Konrad Siek

Distributed Pessimistic Transactional Memory: Algorithms and Properties

Doctoral Dissertation

Submitted to the Council
of the Faculty of Computing Science
of Poznań University of Technology

Advisor: Paweł T. Wojciechowski, Ph. D., Dr. Habil.

Poznań · 2016

Konrad Siek

Rozproszona Pesymistyczna Pamięć Transakcyjna: Algorytmy i Własności

Rozprawa doktorska

Przedłożono Radzie
Wydziału Informatyki
Politechniki Poznańskiej

Promotor: dr hab. inż. Paweł T. Wojciechowski

Poznań · 2016

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy in Computing Science.

Konrad Siek
Distributed Systems Research Group
Faculty of Computing Science
Institute of Computing Science
Poznań University of Technology
konrad.siek@cs.put.edu.pl

Typeset by the author in LaTeX.

This dissertation and associated materials can be downloaded from:
http://www.cs.put.poznan.pl/ksiek/research

Institute of Computing Science
Poznań University of Technology
Piotrowo 2, 60–965 Poznań, Poland
http://www.cs.put.poznan.pl

The research presented in this dissertation was partially funded from National Science Centre funds granted by decision No. DEC-2012/06/M/ST6/00463, from National Science Centre funds granted by decision No. DEC-2012/07/B/ST6/01230, and from the Polish Ministry of Science and Higher Education grant no. POIG.01.03.01-00-008/08.

Acknowledgements

I would like to thank,

My parents and family, Danuta and Marek Siek, Nataniel Siek, and Honorata Tatarska,

My advisor, Dr. Paweł T. Wojciechowski,

Prof. Jerzy Brzeziński and all the members of the Distributed Systems Research Group,

My long-graduated friends, Magdalena Deckert, Bartosz Alchimowicz, and Mirosław Ochodek,

The many past and present inhabitants of rooms 2.7.2, 2.7.5, and 2.7.8, Jan Kończak, Wojciech Wojciechowicz, Marcin Szajek, Andrzej Stroiński, Dariusz Dwornikowski, Marcin Bazydło, Piotr Zierhoffer, Mateusz Hołenko, Maciej Piernik, Dariusz Brzeziński, Sylwia Kopczyńska, Michał Maćkowiak, Tadeusz Kobus, Maciej Kokociński, Paweł Kobyliński, Kalina Jasinska, Krzysztof Ciomek, Mansureh Aghabeig, Małgorzata Trzcielińska, Krystyna Napierała, Konrad Szałkowski, Szymon Francuzik, and Adrian Jaroszewicz,

My older colleagues with whom I shared research interests, Mariusz Mamoński, Piotr Kryger, and Wojciech Mruczkiewicz,

And all of the undergraduate students with whom I had the pleasure to work,

For their support and for putting up with me.

Abstract

In a world dominated by multicore processors and distributed applications, concurrent programming is becoming the norm rather than the exception. However, concurrent programming is also notoriously difficult, because concurrent processes, whether running on independent cores or network nodes, require synchronization. However, a correct and effective application of the existing low-level synchronization mechanisms like locks, barriers, and semaphores, requires skill and careful analysis of the interdependencies among all the processes in the system. Worse still, an incorrect application will lead to catastrophic problems like deadlocks, livelocks, race conditions, or priority inversion.

Hence distributed and multicore computing needs abstractions. Transactional Memory (TM) is an approach aiming to simplify concurrent programming by automating synchronization while maintaining efficiency. This is accomplished by providing the programmer with the transaction abstraction. When using TM the programmer need not understand the underlying concurrency control mechanisms, only the guarantees it provides, as expressed by liveness, progress, and safety properties. TM's extension into distributed systems is called Distributed Transactional Memory (DTM).

Both TM and DTM usually employ the optimistic concurrency control approach, which relies on transactions aborting and restarting if conflicts occur. This is a fairly universal approach which only requires that transactions clean up after their own executions. However, in practice, some aborted transactions can have effects that simply cannot be cleaned up: the effects of system calls, network communication, locking, or I/O operations. Such irrevocable operations are particularly likely to occur in distributed systems where transactions tend to be more complex.

In such cases the pessimistic approach is a more appropriate solution. Pessimistic transactions do not abort on conflict, but defer operations so that conflicts never materialize. Since pessimistic TM need not abort, the problem of irrevocable operations is solved. However, in existing research pessimistic TM was shown to be less efficient than its optimistic counterpart. In this dissertation we aim to show that pessimistic TM can be equally as performant as optimistic TM while solving optimistic TM's problems with irrevocable operations. We do this by employing the early release mechanism, which allows conflicting transactions to nevertheless commit correctly.

If a TM transaction reads a stale value it may and execute an unanticipated dangerous operation, like dividing by zero, accessing an illegal memory address, or entering an infinite loop. Thus, TM safety properties must restrict or eliminate the ability of transactions to view inconsistent state. To that end, opacity, the TM property commonly used

for TM systems, includes the condition that transactions not read values written by other live (not completed) transactions alongside serializability and real-time order conditions. However, if reading from live transactions is not allowed, then opacity precludes early release, regardless of whether dangerous effects actually occur. Thus, TM with early release requires more nuanced safety properties, that limit inconsistent views but nevertheless provide strong guarantees. Hence, we formally analyze the existing TM safety poperties as well as database consistency conditions to determine whether they allow early release and what other guarantees they provide. We also introduce last-use opacity and strong last-use opacity, two strong TM safety properties that enforce practical correctness guarantees and apply to TM with early release.

We also analyze existing pessimistic and distributed TM systems, and TM systems that employ early release. The analysis allows us to select a DTM concurrency control algorithm called SVA, which we extend to eliminate requirements for central coordination, and to lift it to a more general system model, producing the SVA+R algorithm and its variants. We then use these extended algorithms as a basis for new highly parallel pessimistic DTM concurrency control algorithms: OptSVA+R and OptSVA-CF+R (and their variants). We show through formal analysis that these algorithms allow more parallel schedules than their predecessors. We also introduce new proof techniques that allow us to demonstrate that SVA executes as if it were opaque, and that SVA+R, OptSVA+R, and OptSVA-CF+R satisfy last-use opacity. Finally, we implement the new algorithms and show experimentally that OptSVA-CF+R outperforms a state-of-the-art optimistic DTM, but does so without aborting transactions.

We also introduce a precompiler that improves the practicality of the described implementations by deriving the information required *a priori* by some of the TM algorithms from the source code of transactions, so that the information does not need to be provided by the programmer.

In aggregate, the results presented in this dissertation show that it is possible to propose a pessimistic TM concurrency control algorithm for distributed transactional memory, whose implementation achieves high performance, applies practically within general system models, provides strong liveness and progress guarantees, as well as strong correctness guarantees (encapsulated within new safety properties), and guarantees correct execution for irrevocable operations.

Contents

List of Publications

Journal Papers

K. Siek and P. T. Wojciechowski. Proving opacity of transactional memory with early release. *Foundations of Computing and Decision Sciences*, Volume 40, Issue 4. December 2015.

K. Siek and P. T. Wojciechowski. Atomic RMI: A Distributed Transactional Memory Framework. *International Journal of Parallel Programming*, Volume 44, Issue 3. June 2015.

In Preparation

P. T. Wojciechowski and K. Siek. The optimal pessimistic transactional memory algorithm, May 2016. ArXiv:1605.01361 [cs.DC]. (In submission.)

K. Siek and P. T. Wojciechowski. Atomic RMI 2: Highly parallel pessimistic distributed transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, April 2016. ArXiv:1606.03928 [cs.DC]. (Submitted.)

J. Baranowski, P. Kobyliński, K. Siek, and P. T. Wojciechowski. Helenos: A realistic benchmark for distributed transactional memory. *Journal of Systems and Software*, March 2016. ArXiv:1603.07899 [cs.DC]. (In revision.)

K. Siek and P. T. Wojciechowski. Transactions scheduled while you wait. *Journal of Grid Computing*, October 2015. (Submitted.)

K. Siek and P. T. Wojciechowski. Last-use opacity: A strong safety property for transactional memory with early release support. *ACM Transactions on Programming Languages and Systems*, June 2015. ArXiv:1506.06275 [cs.DC]. (Submitted.)

Book Chapters

K. Siek, P. T. Wojciechowski, P. Kujawa, A. Perek, J. Richter, and S. Staszyński. Source-level static analysis and instrumentation. In T. Biały, C. Sobaniec, M. Sobczak, B. Walter, and W. Wróblewski, editors, *Information Technology and its Applications*. Nakom, December 2011.

Conference and Workshop Papers

K. Siek and P. T. Wojciechowski. Brief announcement: Relaxing opacity in pessimistic transactional memory. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, October 2014.

K. Siek and P. T. Wojciechowski. Atomic RMI: a distributed transactional memory framework. In *Proceedings of HLPP'14: the 7th International Symposium on High-level Parallel Programming and Applications*, July 2014.

K. Siek and P. T. Wojciechowski. Zen and the art of concurrency control: An exploration of TM safety property space with early release in mind. In *Proceedings of WTTM'14: the 6th Workshop on the Theory of Transactional Memory (co-located with ACM PODC'14)*, July 2014.

P. T. Wojciechowski and K. Siek. Having your cake and eating it too: Combining strong and eventual consistency. In *Proceedings of PaPEC'14: the 1st Workshop on the Principles and Practice of Eventual Consistency (co-located with ACM SIGOPS EuroSys 2014)*, April 2014.

K. Siek and P. T. Wojciechowski. Brief announcement: Towards a fully-articulated pessimistic distributed transactional memory. In *Proceedings of SPAA'13: the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, July 2013.

K. Siek and P. T. Wojciechowski. A formal design of a tool for static analysis of upper bounds on object calls in Java. In *Proceedings of FMICS'12: the 17th International Workshop on Formal Methods for Industrial Critical Systems (co-located with FM'12)*, number 7437 in Lecture Notes in Computer Science, August 2012.

P. T. Wojciechowski and K. Siek. Transaction concurrency control via dynamic scheduling based on static analysis. In *Proceedings of WTM'12: the Euro-TM Workshop on Transactional Memory (co-located with ACM SIGOPS EuroSys 2012)*, April 2012.

K. Siek and P. T. Wojciechowski. Statically computing upper bounds on object calls for pessimistic concurrency control. In *Proceedings of EC^2'10: the Workshop on Exploiting Concurrency Efficiently and Correctly (co-located with CAV'10, part of FLoC'10)*, July 2010.

Table of Symbols

Symbol	Description
$p_k \in \Pi$	process,
$x, y, z \in \mathit{Var}$	variables,
$\lceil x \rfloor, \lceil y \rfloor, \lceil z \rfloor \in \mathit{Obj}$	objects,
$\mathtt{M}_{\lceil x \rfloor}$	interface of object $\lceil x \rfloor$,
$\mathcal{S}$	state of all objects,
$\mathcal{S}_0$	initial state of all objects,
$\mathtt{S}_{\lceil x \rfloor}$	state of object $\lceil x \rfloor$,
$\mathtt{m} \in \mathtt{M}_{\lceil x \rfloor}$	operation
$\mathtt{buf}_i(x), \mathtt{st}_i(x)$	copy buffers for variable x in transaction T_i,
$\mathtt{log}_i(x)$	log buffer for variable x in transaction T_i,
$\mathtt{buf}_i(\lceil x \rfloor), \mathtt{st}_i(\lceil x \rfloor)$	copy buffers for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{log}_i(\lceil x \rfloor)$	log buffer for object $\lceil x \rfloor$ in transaction T_i,
start_i	initialization operation for transaction T_i,
tryC_i	commit operation for transaction T_i,
tryA_i	abort operation for transaction T_i,
$r_i(x)$	read operation to be executed by transaction T_i on variable x,
$w_i(x)v$	write operation to be executed by transaction T_i writing value v to variable x,
$\mathtt{m}_i(\lceil x \rfloor)$	operation $\mathtt{m}$ to be executed by transaction T_i on $\lceil x \rfloor$,
$r_i(\lceil x \rfloor)$	read operation to be executed by transaction T_i on object $\lceil x \rfloor$,
$w_i(\lceil x \rfloor)v$	write operation to be executed by transaction T_i writing value v to object x,
$\mathit{inv}_i\big[\mathtt{m}\big]$	invocation event by transaction T_i,
$\mathit{res}_i\big[u\big]$	response event by transaction T_i,
op	operation execution,
$\mathit{start}_i \rightarrow \mathit{ok}_i$	initialization operation execution by transaction T_i,
$\mathit{tryC}_i \rightarrow C_i$	commit operation execution by transaction T_i returning value C_i,
$\mathit{tryC}_i \rightarrow A_i$	commit operation execution by transaction T_i returning value A_i,
$\mathit{tryA}_i \rightarrow A_i$	abort operation execution in transaction T_i,
$\mathtt{m}_i(x) \rightarrow v$	execution of operation $\mathtt{m}$ on variable x by transaction T_i returning value u,
$r_i(x) \rightarrow v$	read operation execution on variable x by transaction T_i returning value v,
$w_i(x)v \rightarrow u$	write operation execution on variable x by transaction T_i writing v and returning value u,
$\mathtt{m}_i(\lceil x \rfloor) \rightarrow v$	execution of operation $\mathtt{m}$ on object $\lceil x \rfloor$ by transaction T_i returning value u,

Symbol	Description
$r_i(\lceil x \rfloor) \rightarrow v$	read operation execution on object $\lceil x \rfloor$ by transaction T_i returning value v,
$w_i(\lceil x \rfloor) v \rightarrow u$	write operation execution on object $\lceil x \rfloor$ by transaction T_i writing v and returning value u,
v, u	values,
ok_i	succesful execution return value for transaction T_i,
A_i	abort return value for transaction T_i,
C_i	succesful commit return value for transaction T_i,
$\square$	placeholder value,
v_0	initial value,
ω	unknown value,
$\perp$	initial state of copy buffers,
$T_i, T_j, T_k, T_l \in \mathbb{T}$	transactions,
$\mathbb{I} \subseteq \mathbb{T}$	irrevocable transactions,
$\mathbb{R} \subseteq \mathbb{T}$	reluctant transactions,
$\mathtt{ASet}_i$	transaction T_i's access set,
$\mathtt{RSet}_i$	transaction T_i's read set,
$\mathtt{WSet}_i$	transaction T_i's write set,
$\mathfrak{P}$	property,
$\mathbb{L}$	language,
$s \in \mathbb{L}$	statement,
$\mathbb{L}(s)$	evaluation of statement s according to language $\mathbb{L}$,
$\mathbb{P}$	program,
$\mathtt{P} \in \mathbb{P}$	subprogram,
$\mathcal{E}(\mathbb{P}, \Pi)$	execution of program $\mathbb{P}$ by processes Π,
$\mathscr{T}$	trace,
$H \in \mathbb{H}$	history,
$S \in \mathbb{H}$	a sequential history,
$\hat{S}_H \in \mathbb{H}$	a sequential witness history to history H,
$\hat{\mathbb{T}}^x \subseteq \mathbb{H}$	transactions decided on variable x,
$\hat{\underset{\vee}{\mathbb{T}}}^x \subseteq \mathbb{H}$	transactions strongly decided on variable x,
$\psi_{\mathscr{T}}(T_i, x) \subseteq \mathbb{T}$	transactions reverting the state of x as viewed by transaction T_i,
$\mathbb{H}^{\mathbb{P},\Pi} \subseteq \mathbb{H}$	all possbile histories from execution of program $\mathbb{P}$ by processes Π,
$H \models \mathcal{E}(\mathbb{P}, \Pi)$	history H resulting from execution of program $\mathbb{P}$ by processes Π,
$\mathbb{H}_{\mathfrak{P}} \subseteq \mathbb{H}$	all histories that satisfy property $\mathfrak{P}$,
$\mathit{Hist}(\mathscr{T})$	history created from trace $\mathscr{T}$,
$\mathit{Vis}(H, T_i)$	visible operation history, a subhistory of H as viewed by transaction T_i,
$\mathit{LVis}(H, T_i)$	last-use visible operation history, a subhistory of H as viewed by transaction T_i,
$\mathit{SLVis}(H, T_i)$	strongly last-use visible operation history, a subhistory of H as viewed by transaction T_i,
$\mathit{Decomp}(H)$	a decomposition of history H,
$\mathit{Compl}(H)$	a completion of history H,
$\mathcal{C}_i(H)$	a cut of transaction T_i in history H,
$H \vert T_i$	subhistory of history H containing only operation executions from transaction T_i,
$H \vert \lceil x \rfloor$	subhistory of history H containing only complete operation executions on object $\lceil x \rfloor$,
$H \vert x$	subhistory of history H containing only complete operation executions on variable x,
$H{\uparrow}e$	subhistory of history H containing only event e and events preceding e,
$H \hat{\vert} T_i$	decided transaction history, a subhistory of H for transaction T_i,
$H \overset{\diamond}{\vert} T_i$	decided transaction history completion, a subhistory of H for transaction T_i,
$H \hat{\underset{\vee}{\vert}} T_i$	strongly decided transaction history, a subhistory of H for transaction T_i,

Symbol	**Description**
$H\overset{\diamond}{\underset{\diamond}{\vert}}T_i$	strongly decided transaction history completion, a subhistory of H for transaction T_i,
$H' \subseteq H$	history H' is a subsequence (subhistory) of H,
$Seq(x)$	sequential specification of object x,
$Seq(\lceil x \rfloor)$	sequential specification of object $\lceil x \rfloor$,
$\mathtt{lk}^g$	global lock,
$\mathtt{lk}(x)$	lock for variable x,
$\mathtt{lk}(\lceil x \rfloor)$	lock for object $\lceil x \rfloor$,
$\mathtt{owner}(\mathtt{lk}(x))$	owner of lock $\mathtt{lk}(x)$,
$\mathtt{mode}(\mathtt{lk}(x))$	state of lock $\mathtt{lk}(x)$,
$\bot, R, W$	lock states: unlocked, locked in read mode, write mode,
$\mathtt{lock}\ \mathtt{lk}(x) \rightarrow R$	acquire lock $\mathtt{lk}(x)$ in read mode,
$\mathtt{lock}\ \mathtt{lk}(x) \rightarrow W$	acquire lock $\mathtt{lk}(x)$ in write mode,
$\mathtt{convert}\ \mathtt{lk}(x) \rightarrow W$	escalate lock $\mathtt{lk}(x)$ from read mode to write mode,
$\mathtt{convert}\ \mathtt{lk}(x) \rightarrow R$	de-escalate lock $\mathtt{lk}(x)$ from write mode to read mode,
$\mathtt{unlock}\ \mathtt{lk}(x)$	release lock $\mathtt{lk}(x)$,
$\mathtt{lv}(\lceil x \rfloor)$	local version counter for object $\lceil x \rfloor$,
$\mathtt{ltv}(\lceil x \rfloor)$	local terminal version counter for object $\lceil x \rfloor$,
$\mathtt{pv}_i(\lceil x \rfloor)$	private version counter for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{gv}(\lceil x \rfloor)$	global version counter for object $\lceil x \rfloor$,
$\mathtt{cv}(\lceil x \rfloor)$	current version counter for object $\lceil x \rfloor$,
$\mathtt{rv}_i(\lceil x \rfloor)$	recovery version counter for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{rc}_i(\lceil x \rfloor)$	read counter for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{wc}_i(\lceil x \rfloor)$	write counte for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{uc}_i(\lceil x \rfloor)$	update counter for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{ac}_i(\lceil x \rfloor)$	access counter for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{supr}_i(\lceil x \rfloor)$	supremum for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{rub}_i(\lceil x \rfloor)$	read supremum for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{wub}_i(\lceil x \rfloor)$	write supremum for object $\lceil x \rfloor$ in transaction T_i,
$\mathtt{wub}_i(\lceil x \rfloor)$	update supremum for object $\lceil x \rfloor$ in transaction T_i,
$g_i(x)v$	view event on variable x in transaction T_i returning value v,
$s_i(x)v$	update event on variable x in transaction T_i setting value v,
$\circ s_i(x)v$	routine update event on variable x in transaction T_i setting value v,
$\sqsupset s_i(x)v$	recovery update event on variable x in transaction T_i setting value v,
$ESet_i$	event set for transaction T_i,
$\xi(\mathscr{T}, T_i, T_j)$	view chain between transactions T_i and T_j in trace $\mathscr{T}$,
$\prec_{\mathscr{T}}$	precedence relation in trace $\mathscr{T}$,
$\nprec_{\mathscr{T}}$	direct precedence relation in trace $\mathscr{T}$,
$\prec_{\mathscr{T}}$	preface relation in trace $\mathscr{T}$,
$\rightsquigarrow$	instigation relation,
$\curvearrowleft$	dependence relation,
$\circlearrowleft$	view relation,
$\overset{\cdots}{\circlearrowleft}$	virtual view relation,
$\dot{\prec}$	isolation order relation,
$\ddot{\prec}$	directly isolation order relation,
$\tau_{\mathscr{T}}$	execution time of trace $\mathscr{T}$,
$\tau_{\mathscr{T}}(e)$	point in time at which event e is executed in trace $\mathscr{T}$,
$\tau^{\vdash}_{\mathscr{T}}(op)$	point in time at which operation execution op starts executing,
$\tau^{\dashv}_{\mathscr{T}}(op)$	point in time at which operation execution op finished executing,
$\tau^{r}_{\mathscr{T}}(T_i, x)$	release time of variable x by transaction T_i in trace $\mathscr{T}$,
$\tau^{c}_{\mathscr{T}}(T_i, x)$	completion time of variable x by transaction T_i in trace $\mathscr{T}$,
$\mathtt{location}(\lceil x \rfloor)$	location of object $\lceil x \rfloor$,
$\mathtt{location}(x)$	location of variable x,
$\mathtt{location}(T_i)$	location of transaction T_i,
$\mathcal{R}, \mathcal{Q}, \mathcal{L}$	locations.

1 Introduction

Concurrent programming is well known to be difficult (see e.g., [21, 39, 40, 68]). The source of its difficulty lays in the fact that concurrent execution can cause operations on separate processors to interleave in ways that produce anomalous results, especially when memory accesses are concerned. Thus, it becomes necessary for the programmer to predict and eliminate such interleavings by synchronizing the execution of specific operations. Yet implementing synchronization correctly is notoriously difficult too, since the programmer must reason about interactions among seemingly unrelated parts of the system code. Furthermore, low-level synchronization mechanisms like barriers, monitors, semaphores, and locks are easily misused so that performance, consistency, and progress fall prey to simple bugs and design flaws. Worse still, the misuse may result in deadlocks, livelocks, races, priority inversion, or viewing an inconsistent state of the system. Errors like these are difficult to catch, difficult to track down, non-deterministic, and often catastrophic in their effects.

However, concurrent programming is inescapable. In a world dominated by multicore processors even the rank-and-file programmer is increasingly likely to have to turn to it to take full advantage of various multiprocessor architectures. Moreso, the advent and continuous evolution of large scale (geo-)distributed applications over the past decade caused service-oriented architecture and cloud computing to become ubiquitous, and anything from text editing, through data storage, to big data processing is delegated to remote services. Thus, since distributed computing is concurrent in nature, developing most practical applications requires that the programmer be aware of problems that may stem from concurrency.

Hence, it is necessary to aid programmers in writing concurrent applications and to shield them from the perils of concurrency. In order to do this, concurrent programming needs to follow other programming domains in introducing encapsulating abstractions. For instance, since application programmers have enough to worry about without delving into the details of distributed computing and networking, these details are comprehensively abstracted away and hidden from them within opaque libraries (e.g., Netty, JGroups, Java Message Service, or the Java Remote Method Invocation mechanism), to the point where such programmers rarely resort to directly using low-level mechanisms like sockets. Similarly, the problem of keeping concurrent execution correct should be so hidden away under an abstraction, rather than expecting the programmer to implement each manually using synchronization primitives and to repeatedly avoid their various pitfalls.

Transactional Memory

Transactional memory (*TM*) [44, 71] serves as such an abstraction. TM transplants the idea of transactions from database systems which automate concurrent execution and obscure the details of synchronization from the programmer (see e.g., [12, 16, 91]). Specifically, the programmer annotates the fragments of the code where synchronization should be applied as transactions, and the TM system executes them using some underlying concurrency control algorithm. The algorithm interleaves concurrent transactions to improve performance, while simultaneously applying synchronization as needed to ensure that the execution provides an illusion of transaction atomicity and isolation. There is no need, however, for the programmer to know the details of the employed algorithm, only to be aware of the specific algorithm's correctness guarantees expressed by its properties (e.g., *serializability* [60] or *opacity* [33]). Therefore, such an abstraction makes it is easier for conventionally-trained programmers to reason about and implement correct and efficient concurrent programs.

Distributed transactional memory (*DTM*) [14, 49, 18, 68, 86, 10] extends the idea of TM into distributed systems. This introduces new problems to be addressed by the DTM system, such as asynchrony and partial failures, but also opens up new possibilities. The thing that most starkly differentiates TM from its database predecessors is that apart from executing read and write operations on shared memory (shared variables or shared objects), the TM system can provide interfaces for other or different operations. Classically, this can be an operation like increment, or stack operations like push and pop, all of which atomically read and write the state of a shared object. The operations can also be more complex, computation-intensive, programmer-specified procedures that execute just about any code, and can include code with side effects. In DTM this idea can be taken further: a DTM system can execute the code of an operation on the same network node as the transaction that executes the operation, or on the same node as the object on which the operation is being executed. This choice gives rise to various DTM system models. The *data flow* (*DF*) model entails shared objects being migrated to the client that uses them (while maintaining only a single copy of the object in the system). In such a case the computation and side effects will be performed on the host onto which the object is migrated to execute an operation. In the *control flow* (*CF*) model, shared objects are bound to individual hosts and do not migrate, so the execution of their operations is performed always on the object's "home" host. Both models have their advantages and disadvantages, but a unique feature of CF is that it allows to delegate computation to remote hosts. This allows client transactions to "borrow" computational power from remote object servers. In effect shared resources can act as both shared memory and web services. This provides greater flexibility in designing and implementing distributed applications.

Both of those models assume a distributed system where each network node is distinct and each is a host to a number of discrete shared objects. Yet another model is to use *replicated transactions*, where each transaction executes all of its operations so that its effects are applied to all of the network nodes, so that the nodes are effectively replicas of one another. We concentrate on non-replicated DTM in the remainder of this dissertation.

Optimistic Concurrency Control

Typically, TM (and DTM) systems employ the *optimistic* approach to concurrency control. Generally speaking, a client transaction executes its code speculatively, regardless of other transactions running in parallel. If the transaction manages to perform all of its code without interference it finishes successfully—*commits*. However if two transactions

try to access the same shared variable or object, and one of them writes to it, they *conflict* and the TM system forces at least one of the transactions to *abort* and subsequently *restart*. There are various ways of doing this, but, generally, optimistic TM systems attempt to detect conflicts as soon as possible, to waste as little work as possible. They also tend to buffer the values they write and update the original variable or object at *commit-time*, rather than whenever each operations is executed, at *encounter-time*.

The optimistic approach is a fairly universal solution, however, it has two major drawbacks. First, optimistic TM experiences problems stemming from speculative execution in environments with *high contention*—i.e., where very many transactions attempt to access the same shared variable or object simultaneously. Specifically, high contention makes it more likely that most transactions will conflict and be forced to abort and re-execute (at least partially). It also becomes more likely that each transaction will execute several times before it eventually manages to commit. This causes a large portion of the computational work done by the system as a whole to be wasted. In addition, it means transactions that do manage to commit are effectively executed in sequence. Both of these issues can have a significant impact on the performance of the system. A number of mechanisms were proposed to mitigate this problem by managing conflicting transactions, so that they are prevented from conflicting again. These range from simple mechanisms like exponential back-off [43], through serialization of execution of conflicting transactions, to a dispatcher avoiding collisions based on a probability of conflict [24, 105] and other advanced contention management techniques [25, 70]. These solutions defer the point at which specific transactions start (or re-start after a conflict occurs) which reduces the number of transactions executing at the same time. However, such solutions typically operate based on various threshold parameters, which means they must either be tuned manually or the system must derive such parameters during execution. It also means the system needs to re-tune in reaction to changing workloads. In addition, these solutions tend to use central coordination, which prevents them from being used in distributed systems.

The second issue with optimistic TM is the problem of *irrevocable* operations. These are operations that cannot be aborted and should not be repeated, such as I/O operations, network communication, or acquiring locks. These are typically part of any complex code and are often difficult to locate when the application uses libraries, or when it composes calls to remote services (in DTM). However, if these operations appear inside a transaction and the transaction is forced to re-execute, they will cause side effects to be visible (e.g., there may be stray network messages or a non–re-entrant lock may be re-acquired and cause a deadlock). However, the *modus operandi* of optimistic transactions depends on aborted transactions cleaning up after themselves. Fixing the problem in the optimistic approach leads to complicated or cumbersome solutions that prevent speculative execution in certain transactions. For instance, irrevocable transactions are introduced in [92]. Such transactions always win conflicts with other transaction, so they are never forced to abort. However, in order to prevent the paradoxical situations of two conflicting irrevocable transactions, only one such transaction can execute at a time, which causes this solution to limit parallelism. A different solution is proposed in [9, 62], where the TM maintains multiple versions of each variable or object, so that a transaction can read an older version of a variable instead of aborting. However, this solution introduces complexity and overhead to the concurrency control algorithms. Another solution, proposed in [38], is to move the irrevocable operations to commit, however this requires instrumentation and cannot be applied to all types of irrevocable operations (e.g. locking). Hence, the problem of irrevocable operations is often ignored or they are simply forbidden in transactions (e.g., in Haskell [41]). Other research suggests that a form of compensation can be used to fix the computations, so that conflicting transactions do not

abort [13]. However, solutions like these limit the practicality of TM, especially in complex service-oriented DTM systems, where irrevocable operations are often unavoidable, and compensation may be impossible. For instance, if the the execution of an operation on a shared object representing a service causes a material effect (e.g. printing a book), then the irrevocable operation is a part of the semantics of the operation, and cannot be compensated for at all (without cost).

Pessimistic Concurrency Control

A simpler method of dealing with both the problems described above is using a *pessimistic* concurrency control algorithm. The pessimistic approach originates from database transactions (e.g., two-phase locking [12, 91]) and was brought to TM in [56, 1, 10] as well as the work in [96, 97]. The general idea behind pessimistic TM is that it does not execute transactional operations speculatively, but delays them until they no longer conflict. This means, that forced aborts do not occur on conflict, so they are much less common or even impossible, and, therefore, high contention or irrevocability do not cause abort-related problems.

However, the authors of [56] show that the pessimistic approach can have negative impact on performance in high contention, since it depends on serializing transactions that perform write operations to prevent aborts, which inherently limits parallelism. The goal of this dissertation is to show that this penalty on parallelism is not inherent in the pessimistic approach and can be overcome with the application of specific optimization techniques.

To that end, we consider the technique of *early release*. Early release is an optimization technique for TM, where certain pairs of transactions technically conflict but nevertheless both are allowed to commit correctly [65], if they nevertheless produce a history that is correct. This is particularly useful with pessimistic concurrency control, where transactions, as a rule, do not abort. If they do not abort, then viewing the final state of a variable does not have to lead to inconsistencies, even if the value is read from a live transaction. On the other hand, TM systems employing early release (e.g., [43, 65, 28, 13, 75]) show that this yields a significant and worthwhile performance benefit. For this reason we intend to use it as the core of our optimizations aiming to present a practical, safe, and well-performing pessimistic TM.

Safety

Since TM allows transactional code to be mixed with non-transactional code and to contain virtually any operation, rather than just reads and writes like in its database counterparts, greater attention must be paid to the state of shared variables at any given time. For instance, *serializability* [60] dictates that as long as the execution of committed transactions reflects some sequential execution, then the entire concurrent execution is correct. Hence, if a database transaction reads a stale value, it must simply abort and retry, and no harm is done. However, if a TM transaction reads a stale value it may break some presupposed invariant and execute an unanticipated dangerous operation, like dividing by zero, accessing an illegal memory address, or entering an infinite loop. Thus, it is insufficient for TM systems to use traditional database consistency conditions like serializability to describe the guarantees they ensure. Instead, TMs must restrict or eliminate the ability of transactions to view inconsistent state. To that end, the safety property called *opacity* [33] was introduced, which includes the condition that transactions do not read values written by other live (not completed) transactions alongside

serializability and real-time order conditions. Opacity became the gold standard of TM safety properties, and most TM systems found in the literature are, in fact, opaque.

However, if reading from live transactions is not allowed, then opacity precludes early release, regardless of whether dangerous effects actually occur. Thus, before presenting a practical and efficient pessimistic TM, we must first determine what correctness guarantees should be provided by such a system and find or devise TM safety properties that are comparably strong to opacity, but allow the application of early release. Since opacity is a very restrictive property, a number of more relaxed properties were introduced that tweaked opacity's various aspects to achieve a more practical property. These properties include *virtual world consistency* (*VWC*) [48], *transactional memory specification* (*TMS1* and *TMS2*) [22], *elastic opacity* [28], *live opacity* [26], and others. A significant part of this dissertation is dedicated to examining these properties in order to determine whether or not they allow the use of early release in TM, and, if so, what compromises they make with respect to consistency, and what additional assumptions they require. On the basis of that analysis, we also introduce new properties to enforce specific practical correctness guarantees that apply to TM with early release in general if they are not given by existing properties.

System Model

TM can be applied to various system models that dictate for which assumptions particular concurrency control algorithms must account. Depending on which model a given TM (or DTM) operates on, it is more or less suitable for specific applications.

For instance, TMs can be designed specifically to operate on shared variables, which are defined by a single value that can be either read or overwritten by operations executed by the transaction. Such a model is typical for non-distributed TM (e.g. [21, 39, 65]), but in DTM it is more typical to see a model where shared objects are used instead of variables (e.g. [68, 86]). Shared objects can have complex state consisting of several variables, and can specify arbitrary interfaces. Among these models we differentiate between a *homogeneous* and *heterogeneous object model*. In the former model all objects are the same and relatively simple: they represent structures such as counters or stacks. These objects have a single read and write operation in their interface, whose semantics are known. In the heterogeneous model objects are assumed to each have their own interface defining arbitrary methods with arbitrary, possibly hidden semantics operating on hermetic, complex state. Different models have different applications, with variables finding uses in high performance local and parallel systems as well as data stores, while the object models find uses in complex cloud-computing applications and service-oriented architectures, where objects can represent entire services.

In addition, TM systems can vary in the interface that each transaction provides to the programmer. Many pessimistic as well as optimistic TM systems are *commit-only*, meaning that each running transaction strives to execute all of its code and eventually commit (e.g. [96, 6, 56]). On the other hand, *arbitrary abort* TMs allow transactions to execute a programmatic abort operation from within the transactional code, that will withdraw the transaction's effects. The addition of an abort operation to the transactional interface makes the TM more expressive and provides a vital feature for an efficient implementation of partial-failure resistant distributed systems.

Note that a TM system operating in the variable model can make many more safe assumptions about the state of the system than a TM system operating in either object model, so if we compare their performance in the variable model the former TM is likely to be more efficient, whereas if we compare their performance in the object model, the latter will execute correctly, while the former might not. Similarly, a TM operating

in the arbitrary abort model can be used as a commit-only TM, although likely less efficient one, while a commit-only TM cannot operate correctly or as efficiently wherever manually-issued aborts are required. We stipulate that in order for a pessimistic DTM to be practical, it must be able to span a range of these system models, being able to both perform correctly in the more general model, but also provide variants that operate more efficiently in the more specialized models. Hence, we consider the implication of applying the algorithms presented in this dissertation in various system models, striving for generality.

We also add that in order for a distributed TM to be practically applicable, it must not depend on centralized data structures and introduce a single point of failure to a distributed system, since that compromises its scalability and its ability to function despite partial failures.

Liveness and Progress

Apart from correctness, we note that TM should also ensure that transactions make progress with their assigned computations by allowing individual operations to execute (guaranteed by *liveness* properties), and by making sure transactions are eventually given the chance to commit (described by *progress* properties). *Deadlock-freedom* is a rudimentary TM liveness property which requires that the transactions in a TM system never enter a wait-cycle from which they never leave. *Strong progressiveness* [33] is a common progress property which stipulates that a conflict should never lead all of the conflicting transactions to be forced to abort. A practical TM algorithm that does not meet these conditions is useless in practice, since it can lead the concurrent application to "get stuck" or to transactions treading water while infinitely restarting.

Thesis

Given our goals and stipulations presented above, we formulate our main thesis as follows:

It is possible to propose a pessimistic TM concurrency control algorithm for distributed transactional memory that simultaneously:

a) achieves high performance,

b) satisfies strong safety, liveness, and progress properties,

c) guarantees correct execution for irrevocable operations, and

d) applies practically within general system models.

Contributions

We demonstrate the veracity of this thesis through the aggregate of the contributions that we state briefly below, and that we describe in detail in the following chapters of this dissertation.

I **An analysis of existing properties and TM and DTM algorithms.**
We formally examine the existing TM safety properties and database consistency conditions and determine whether or not they can be applied to TM systems that employ early release. Specifically, we resolve whether or not they allow early release at all, what classes of inconsistent views they admit, and what restrictions they put on transactions. We then survey selected existing pessimistic and distributed TM systems, and TM systems that employ early release and determine their parameters, as well as their safety guarantees. This allows us to draw further conclusions about

the applicability of existing TM safety properties to systems with early release. We also use the analysis to determine which algorithms and techniques can be used for implementing a pessimistic DTM. The analyses are presented in Chapters 3 and 4, and extend the research presented in [77] and [79].

II **Novel strong safety properties for TM and DTM with early release.**

We introduce two new TM safety properties called *last-use opacity* and *strong last-use opacity* which give strong consistency guarantees and preclude most classes of inconsistent views, while allowing early release. We demonstrate them and discuss their characteristics in Chapter 5. They were first presented in [76, 79].

III **Novel pessimistic TM and DTM concurrency control algorithms.**

We introduce novel pessimistic TM concurrency control algorithms designed for distributed systems. We start by extending existing versioning algorithms [96, 97] to relieve them of a single point of failure and lift them into the arbitrary abort model, producing *BVA+R*, *SVA+R*, and *RSVA+R*. We then employ a number of far reaching modifications with respect to operation types to produce new highly parallel concurrency control algorithms: *OptSVA+R* and *OptSVA-CF+R* (and their variants). We show that these algorithms allow more parallel schedules than their predecessors and demonstrate their properties. The algorithms are presented in Chapter 6 and the correctness proofs for a selection of them are given in Chapter 7. This is an extension of our research in [74, 75, 78, 82].

IV **Safety proofs and proof techniques.**

We introduce proof techniques that allow to demonstrate the safety properties (opacity and last-use opacity) of algorithms with early release. We then use the proof techniques to prove the properties of selected algorithms. This is presented in Chapter 7 and extends the work presented in [79, 80, 102].

V **Implementations of the new algorithms.**

We provide CF DTM system implementations for two of the presented concurrency control algorithms and show that OptSVA-CF+R outperforms a state-of-the-art optimistic distributed TM. We show this in Chapter 8 and it follows the research in [75, 78, 82].

VI **Static analysis and precompiler.**

We introduce a precompiler for the Java language that can derive the information required *a priori* by some of the TM algorithms from the source code of transactions. This is shown in Chapter 9 and represents the research published in [72, 73].

2
Preliminaries

This chapter introduces the basic concepts pertaining to further discussion, including basic definitions describing transactional memory and execution of programs within it, as well as various system models, and properties. We also explain the convention we use for diagrams showing transactional executions.

2.1 Basic Definitions

In this section we introduce basic definitions relevant to further discussion and the notation used throughout the dissertation.

2.1.1 Processes

The system is composed of processes $\Pi = \{p_1, p_2, ..., p_n\}$ concurrently executing program $\mathbb{P}$ which constitutes a set of sequential programs $\mathbb{P} = \{\mathtt{P}_1, \mathtt{P}_2, ..., \mathtt{P}_n\}$, where process p_i executes $\mathtt{P}_i$. Each subprogram $\mathtt{P}_k \in \mathbb{P}$ is a finite sequence of statements $\mathtt{P}_k = s_1, s_2, ..., s_m$ in some language $\mathbb{L}$. The definition of $\mathbb{L}$ can be whatsoever, as long as it provides constructs to execute transactional operations in accordance with the interface and assumptions described further in Section 2.1.2.

Given program $\mathbb{P}$ and a set of processes Π, we denote an execution of $\mathbb{P}$ by Π as $\mathcal{E}(\mathbb{P}, \Pi)$. An execution entails each process $p_k \in \Pi$ evaluating some prefix of subprogram $\mathtt{P}_k \in \mathbb{P}$. The evaluation of each statement by a process is deterministic and follows the semantics of $\mathbb{L}$. This evaluation produces a (possibly empty) sequence of events (steps) which we denote $\mathbb{L}(s)$.

Furthermore by $\mathbb{L}(\mathtt{P}_k)$ we denote a sequence s.t. given $\mathtt{P}_k = s_1, s_2, ..., s_m$, $\mathbb{L}(\mathtt{P}_k) = \mathbb{L}(s_1) \cdot \mathbb{L}(s_2) \cdot ... \cdot \mathbb{L}(s_m)$. By extension, $\mathcal{E}(\mathbb{P}, \Pi)$ produces a sequence of events, which we call a trace $\mathscr{T}$. $\mathscr{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$ iff $\forall p_k \in \Pi, \mathtt{P}_k \in \mathbb{P}, \mathbb{L}(\mathtt{P}_k) \subseteq \mathscr{T}$. $\mathcal{E}(\mathbb{P}, \Pi)$ is concurrent, i.e. while the statements in subprogram $\mathtt{P}_k$ are evaluated sequentially by a single process, the evaluation of statements by different processes can be arbitrarily interleaved. Hence, given $\mathscr{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$ and $\mathscr{T}' \vdash \mathcal{E}(\mathbb{P}, \Pi)$, it is possible that $\mathscr{T} \neq \mathscr{T}'$. We call $\mathcal{E}(\mathbb{P}, \Pi)$ a *complete* execution if each process p_k in Π evaluates all of the statements in $\mathtt{P}_k$. Otherwise, we call $\mathcal{E}(\mathbb{P}, \Pi)$ a *partial* execution. By extension, if $\mathcal{E}(\mathbb{P}, \Pi)$ is a complete execution, then $\mathscr{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$ is a *complete* trace, and otherwise $\mathscr{T}$ is a *partial* trace.

2.1.2 Objects and Variables

The system contains a set of *shared objects* (or just *objects*) $Obj = \{\lceil x \rfloor, \lceil y \rfloor, \lceil z \rfloor, ...\}$. An object $\lceil x \rfloor$ is an entity that has *state* $\mathtt{S}_{\lceil x \rfloor}$ and a specified *interface* $\mathtt{M}_{\lceil x \rfloor}$. The state $\mathtt{S}_{\lceil x \rfloor}$ can be defined however. Interface $\mathtt{M}_{\lceil x \rfloor}$ constitutes a set of *operations* (also called *methods*) $\mathtt{M}_{\lceil x \rfloor} = \{\mathtt{m}(\lceil x \rfloor)_1, \mathtt{m}(\lceil x \rfloor)_2, ..., \mathtt{m}(\lceil x \rfloor)_o\}$ that can be executed on $\lceil x \rfloor$ to modify or read elements of the state of $\lceil x \rfloor$. The interfaces of objects can be *heterogeneous*, so given any two objects $\lceil x \rfloor, \lceil y \rfloor$ it is possible that $\mathtt{M}_{\lceil x \rfloor} \neq \mathtt{M}_{\lceil y \rfloor}$.

Any process $p_k \in \Pi$ can execute any operation $\mathtt{m}(\lceil x \rfloor) \in \mathtt{M}_{\lceil x \rfloor}$ on $\lceil x \rfloor$ as part of subprogram $\mathtt{P}_k$. This results in the evaluation of some arbitrary sequence of statements (as part of $\mathtt{P}_k$) in language $\mathbb{L}$ which have the ability to return and modify $\mathtt{S}_{\lceil x \rfloor}$. Objects are *hermetic*, meaning that the state $\mathtt{S}_{\lceil x \rfloor}$ of $\lceil x \rfloor$ can only be modified or read by executing some operation $\mathtt{m}(\lceil x \rfloor) \in \mathtt{M}_{\lceil x \rfloor}$ on $\lceil x \rfloor$.

Among the set of all *Obj* we distinguish a subset we call *shared variables* (or just *variables*). Variables $x, y, z \in Var$ are such objects that $Var \subseteq Obj$, whose state is defined as a single value and whose interface consist of the following two operations:

a) *write* operation $w(x)v$ that sets $\mathtt{S}_{\lceil x \rfloor}$ to value v; the operation's *return value* is the constant *ok* (indicating correct execution),

b) *read* operation $r(x)$ whose *return value* is the current value of $\mathtt{S}_{\lceil x \rfloor}$.

Language $\mathbb{L}$ provides statements that allow operations to be executed within the code of the program. Whenever process $p_k \in \Pi$ executes some operation $\mathtt{m}(\lceil x \rfloor)$ on variable x (for any $\mathtt{m}(\in)\mathtt{M}_{\lceil x \rfloor}$) as part of $\mathtt{P}_k$, this causes an *invocation event* $inv^k\big[\mathtt{m}(\lceil x \rfloor)\big]$ and a subsequent *response event* $res^k\big[v\big]$ to be issued, where v is the return value of $\mathtt{m}(\lceil x \rfloor)$. The pair of these events $(inv^k\big[\mathtt{m}(\lceil x \rfloor)\big], res^k\big[v\big])$ is called a *complete operation execution* and it is denoted $\mathtt{m}^k(\lceil x \rfloor) \rightarrow v$ in shorthand. For the sake of analogy we refer to an invocation event $inv^k\big[o\big]$ without the corresponding response event as a *pending operation execution*.

We distinguish three system models with respect to how shared objects are defined: The *variable object model* (or just *variable model*) describes TMs that only use variables. The *homogeneous object model* describes TMs that operate on simple objects like counters or stacks. These object that share a common interface containing a single read operation and a write operation. The semantics of those operations are known. The read operation may view but not modify the state of the object, while the write operation may both view and modify the state of the object. The *heterogeneous object model* describes TMs that operate on arbitrary or complex objects. In the heterogeneous model each object may define a different interface containing arbitrary operations with arbitrary semantics. The semantics operations may not be known *a priori*. Note that the variable model is a special case of the homogeneous object mode, and the homogeneous object model is a special case of the heterogeneous object model.

2.1.3 Transactions

Transactional memory (*TM*) is a programming paradigm that uses transactions to control concurrent execution of operations on shared variables by parallel processes.

A *transaction* $T_i \in \mathbb{T}$ is some piece of code executed by process p_k, as part of subprogram $\mathtt{P}_k$. Hence, we say that p_k executes T_i. Any transaction T_i is executed by exactly one process p_k and that each process executes transactions sequentially. Process p_k can execute local computations as well as operations on shared objects as part of the transaction. That is, given $\lceil x \rfloor \in Obj$, the process can execute:

a) operation $\mathtt{m}(\lceil x \rfloor) \in \mathtt{M}_{\lceil x \rfloor}$ as part of some transaction T_i, which causes $\mathtt{m}(\lceil x \rfloor)$ to be executed under concurrency control and return either the operation's return value

or the constant A_i; the latter signifies an unsuccessful execution resulting in the transaction aborting.

In the specific case of a variable $x \in Var$, the process can execute the following two operations on x:

a') *write* (denoted $w_i(x)v$, where i indicates transaction T_i) which sets $\mathtt{S}_{\lceil x \rfloor}$ to v and returns ok_i if the operation is successful, and A_i otherwise,

a'') *read* (denoted $r_i(x)$) which returns the value of $\mathtt{S}_{\lceil x \rfloor}$ if the execution is successful, or A_i otherwise.

In addition, the processes can execute the following transactional operations:

b) *start* (denoted $start_i$) which initializes transaction T_i, and whose return value is the constant ok_i,

c) *commit* (denoted $tryC_i$) which attempts to commit T_i and returns either the constant C_i, which signifies a successful commitment of the transaction or the constant A_i in case of a forced abort,

Finally, there is also another operation allowed in some TM system models and not in others, and we wish to discuss it separately. Namely, some TMs allow for a transaction to programmatically roll back by executing the operation:

d) *abort* (denoted $tryA_i$) which aborts T_i and returns A_i.

We call a TM system model where the abort operation is allowed (in addition to other operations) the *arbitrary abort* system model, as opposed to the *commit-only* model.

The operations a–d defined above are part of the so-called *transactional interface* (or *transactional API*). They can only be invoked within a transaction. Specifically, processes execute operations on shared objects only as part of a transaction.

Even though transactions are subprograms evaluated by processes, it is convenient to talk about them as separate and independent entities. Thus, rather than saying p_k executes some operation as part of transaction T_i, we will simply say that T_i executes (or performs) some operation. Hence we will also forgo the distinction of processes in transactional operation executions, and write simply: $start_i \rightarrow ok_i$, $r_i(x) \rightarrow v$, $w_i(x)v \rightarrow ok_i$, $tryC_i \rightarrow C_i$, etc. By analogy, we also drop the superscript indicating processes in the notation of invocation and response events, unless the distinction is needed.

2.1.4 Sequential Specification

Given object $\lceil x \rfloor$, let *sequential specification* of $\lceil x \rfloor$, denoted $Seq(\lceil x \rfloor)$, be a prefix-closed set of sequences containing invocation events and response events which specify the semantics of shared variables. (A set Q of sequences is *prefix-closed* if, whenever a sequence S is in Q, every prefix of S is also in Q.) Intuitively, a sequential specification enumerates all possible correct sequences of operations that can be performed on a variable in a sequential execution.

Specifically, in the case of any variable $x \in Var$, given that D is the domain of $\mathtt{S}_{\lceil x \rfloor}$, and assuming initially $\mathtt{S}_{\lceil x \rfloor} = v_0$ for some $v_0 \in D$, the sequential specification of x s.t., $Seq(x)$ is a set of sequences of the form $[\alpha_1 \rightarrow v_1, \alpha_2 \rightarrow v_2, ..., \alpha_m \rightarrow v_m]$, where each $\alpha_j \rightarrow v_j$ $(j = 1, 2, ..., m)$ is either:

a) $w_i(x)v_j \rightarrow ok_i$, where $v_j \in D$,

b) $r_i(x) \rightarrow v_0$ and there are no preceding writes, or

c) $r_i(x) \rightarrow v_j$ and the most recent preceding write operation is $w_l(x)v_j \rightarrow ok_l$ $(l < i)$.

From this point on, unless stated otherwise, we assume that the domain D of all transactional variables is the set of natural numbers $\mathbb{N}_0$ and that the initial value v_0 of each variable is 0.

2.1.5 Histories

Given a trace $\mathscr{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$, a TM *history* H is a subsequence of trace $\mathscr{T}$ consisting only of executions of transactional operations s.t. for every event e, $e \in H$ iff $e \in \mathscr{T}$ and e is either an invocation or a response event specified by the transactional interface (given in Section 2.1.3). We denote this transformation from a trace $\mathscr{T}$ to a history as $Hist(\mathscr{T})$. If $H \subset \mathscr{T}$ we say $\mathscr{T}$ produces H. Some *subhistory* H' of a history H is a subsequence of H which we denote $H' \subseteq H$.

The sequence of events in a history H_j can be denoted as $H_j = [e_1, e_2, ..., e_m]$. For instance, some history H_1 below is a history of a run of some program that executes transactions T_i and T_j:

$$H_1 = \Big[\ inv_i\big[start_i\big], res_i\big[ok_i\big], inv_j\big[start_j\big], res_j\big[ok_j\big],$$
$$inv_i\big[w_i(x)v\big], inv_j\big[r_j(x)\big], res_i\big[ok_i\big], res_j\big[v\big],$$
$$inv_i\big[tryC_i\big], res_i\big[C_i\big], inv_j\big[tryC_j\big], res_j\big[C_j\big]\ \Big].$$

Given any history H, let $H|T_i$ be the longest subhistory of H consisting only of invocations and responses executed by transaction T_i. For example, $H_1|T_j$ is defined as:

$$H_1|T_j = \Big[\ inv_j\big[start_j\big], res_j\big[ok_j\big], inv_j\big[r_j(x)\big], res_j\big[v\big], inv_j\big[tryC_j\big], res_j\big[C_j\big]\ \Big].$$

We say transaction T_i *is in* H, which we denote $T_i \in H$, iff $H|T_i \neq \varnothing$.

Let $H|\lceil x \rfloor$ be the longest subhistory of H consisting only of invocations and responses executed on object $\lceil x \rfloor$, but only those that form complete operation executions. Let $H|x$ be defined by analogy.

Given a complete operation execution op that consists of an invocation event e^i and a response event e^r, we say op *is in* H ($op \in H$) iff $e^i \in H$ and $e^r \in H$. Given a pending operation execution op consisting of an invocation e^i, we say op *is in* H ($op \in H$) iff $e^i \in H$ and there is no other operation execution op' consisting of an invocation event e^i and a response event e^r s.t. $op' \in H$.

Given two complete operation executions op' and op'' in some history H, where op' contains the response event res' and op'' contains the invocation event inv'', we say op' *precedes* op'' in H if res' precedes inv'' in H. We denote this $op' \prec_H op''$. For operations $op', op'' \in H$, we say op' directly precedes op'', denoted $op' \prec\!\!\!\cdot_H op''$ iff $op' \prec_H op''$ and $\nexists op''' \in H$ s.t. $op' \prec_H op''' \prec_H op''$.

A history whose all operation executions are complete is a *complete* history.

Most of the time it will be convenient to denote any two adjoining events in a history that represent the invocation and response of a complete execution of an operation as that operation execution, using the syntax $e \to e'$. Then, an alternative representation of $H_1|T_j$ is denoted as follows:

$$H_1|T_j = \Big[\ start_j \to ok_j,\ r_j(x) \to v,\ tryC_j \to C_j\ \Big].$$

In addition, sometimes the values written by particular operations, or returned by them will not be relevant to the discussion at hand. In those situations use the placeholder

value $\square$ to indicate that whatever value was passed or returned. For instance, when the value returned by the read operation is irrelevant in $H_1|T_j$, we denote it as follows:

$$H_1|T_j = \Big[\ start_j \to ok_j,\ r_j(x) \to \square,\ tryC_j \to C_j\ \Big].$$

Well-formedness

History H is *well-formed* if, for every transaction T_i in H, $H|T_i$ is an alternating sequence of invocations and responses s.t.,

a) $H|T_i$ starts with an invocation $inv_i\big[start_i\big]$,

b) no events in $H|T_i$ follow $res_i\big[C_i\big]$ or $res_i\big[A_i\big]$,

c) no invocation event in $H|T_i$ follows $inv_i\big[tryC_i\big]$ or $inv_i\big[tryA_i\big]$,

d) for any two transactions T_i and T_j s.t., T_i and T_j are executed by the same process p_k, the last event of $H|T_i$ precedes the first event of $H|T_j$ in H or *vice versa*.

In the remainder of the dissertation we assume that all histories are well-formed.

Unique Writes

History H has *unique writes* if, given transactions T_i and T_j (where $i \neq j$ or $i = j$), for any two write operation executions $w_i(x)v' \to ok_i$ and $w_j(x)v'' \to ok_j$ it is true that $v' \neq v''$ and neither $v' = v_0$ nor $v'' = v_0$.

Completion

Given history H and transaction T_i, T_i is *committed* if $H|T_i$ contains operation execution $tryC_i \to C_i$. Transaction T_i is *aborted* if $H|T_i$ contains response $res_i\big[A_i\big]$ to any invocation. Transaction T_i is *commit-pending* if $H|T_i$ contains invocation $tryC_i$ but it does not contain $res_i\big[A_i\big]$ nor $res_i\big[C_i\big]$. Finally, T_i is *live* if it is neither committed, aborted, nor commit-pending. We say a transaction is *forcibly aborted* if T_i is aborted and $H|T_i$ does not contain an invocation $inv_i\big[tryA_i\big]$.

Given two histories $H' = \big[e'_1, e'_2, ..., e'_m\big]$ and $H'' = [e''_1, e''_2, ..., e''_m]$, we define their concatenation as $H' \cdot H'' = [e'_1, e'_2, ..., e'_m, e''_1, e''_2, ..., e''_m]$. We say P is a prefix of H if $H = P \cdot H'$. Then, let a *completion* $Compl(H)$ of history H be any complete history s.t., H is a prefix of $Compl(H)$ and for every transaction $T_i \in H$ subhistory $Compl(H)|T_i$ equals one of the following:

a) $H|T_i$, if T_i finished committing or aborting,

b) $H|T_i \cdot \Big[res_i\big[C_i\big]\Big]$, if T_i is live and contains a pending $tryC_i$,

c) $H|T_i \cdot \Big[res_i\big[A_i\big]\Big]$, if T_i is live and contains some pending operation,

d) $H|T_i \cdot \Big[tryC_i \to A_i\Big]$, if T_i is live and contains no pending operations.

Note that, if all transactions in H are committed or aborted then $Compl(H)$ and H are identical.

Equivalency

Two histories H' and H'' are *equivalent* (denoted $H' \equiv H''$) if for every $T_i \in \mathbb{T}$ it is true that $H'|T_i = H''|T_i$. When we say H' is equivalent to H'' we mean that H' and H'' are

equivalent.

Real-time Order

A *real-time order* $\prec_H$ is an order over history H s.t., given two transactions $T_i, T_j \in H$, if the last event in $H|T_i$ precedes in H the first event of $H|T_j$, then T_i *precedes* T_j in H, denoted $T_i \prec_H T_j$. We then say that two transactions $T_i, T_j \in H$ are *concurrent* if neither $T_i \prec_H T_j$ nor $T_j \prec_H T_i$. We say that history H' *preserves the real-time order* of H if $\prec_H \subseteq \prec_{H'}$.

Sequential Histories

A *sequential history* S is a history, s.t. no two transactions in S are concurrent in S. Some sequential history S is a *sequential witness history* of H if S is equivalent to H and S preserves the real time order of H. We usually denote such a history $\hat{S}_H$.

Accesses

Given a history H and a transaction T_i in H, we say that T_i *accesses* some object $\lceil x \rfloor$ in H iff there exists some invocation by T_i on any $\lceil x \rfloor$ of any operation $\mathtt{m}(\lceil x \rfloor) \in \mathtt{M}_{\lceil x \rfloor}$ in $H|T_i$. In addition, let T_i's *access set*, denoted $\mathtt{ASet}_i$, in H be a set that contains every object $\lceil x \rfloor$ such that T_i accesses $\lceil x \rfloor$ in H.

With respect to variables specifically, T_i *reads* variable x in H if there exists an invocation $inv_i\big[r_i(x)\big]$ in $H|T_i$. By analogy, we say that T_i *writes* to x in H if there exists an invocation $inv_i\big[w_i(x)v\big]$ in $H|T_i$. If T_i reads x or writes to x in H, we say T_i *accesses* x in H. In addition, let T_i's *read set* be a set that contains every variable x such that T_i reads x. By analogy, T_i's *write set* contains every x such that T_i writes to x. A transaction's *access set*, denoted $\mathtt{ASet}_i$, is the union of its read set and its write set.

Given a history H and a pair of transactions $T_i, T_j \in H$, we say T_i *reads from* T_j if there is some variable x, for which there is a complete operation execution $w_j(x)v \to ok_j$ in $H|T_j$ and another complete operation execution $r_i(x) \to u$ in $H|T_i$, s.t. $v = u$.

Locality

Given any transaction T_i in some history H, any operation execution on a variable x within $H|T_i$ is either *local* or *non-local*. Read operation execution $r_i(x) \to v$ in $H|T_i$ is local if it is preceded in $H|T_i$ by a write operation execution on x, and it is non-local otherwise. Write operation execution $w_i(x)v \to ok_i$ in $H|T_i$ is local if it is followed in $H|T_i$ by a write operation execution on x, and non-local otherwise.

Conflicts

Following [33], transaction conflicts are defined for variables as follows. Given a history H and a pair of transactions $T_i, T_j \in H$, we say T_i and T_j *conflict* on variable x in H if T_i and T_j are concurrent, both T_i and T_j access x, and one or both of T_i and T_j write to x. We call any two operation executions on x that cause two transactions T_i, T_j $(i \neq j)$ to conflict on some x *conflicting* operation executions.

We lift the definition of conflict to any shared object as follows. Given a shared object $\lceil x \rfloor$, any two operations $\mathtt{m}'(\lceil x \rfloor), \mathtt{m}''(\lceil x \rfloor) \in \mathtt{M}_{\lceil x \rfloor}$, $\mathtt{m}''(\lceil x \rfloor)$ *depends on* $\mathtt{m}'(x)$ iff $\mathtt{m}'(\lceil x \rfloor)$ modifies $\mathtt{S}_{\lceil x \rfloor}$ in a way that can impact the execution of $\mathtt{m}''(\lceil x \rfloor)$. The impact can, for instance, amount to any modification of $\mathtt{S}_{\lceil x \rfloor}$ in $\mathtt{m}''(\lceil x \rfloor)$, change the return value of $\mathtt{m}''(\lceil x \rfloor)$ or impact the execution of side-effects in $\mathtt{m}''(\lceil x \rfloor)$. Then, given a history H and a pair of transactions $T_i, T_j \in H$, we say T_i and T_j *conflict* on object $\lceil x \rfloor$ in H if T_i and

T_j are concurrent, both T_i and T_j access $\lceil x \rfloor$, and any operation in $H|T_i|\lceil x \rfloor$ depends on some operation in $H|T_j|\lceil x \rfloor$ or vice versa (or both). If the precise semantics of operations of $\lceil x \rfloor$ are not know, we must conservatively assume that the dependency relation exists between any two operations in $\mathtt{M}_{\lceil x \rfloor}$. Hence, any two concurrent transactions conflict if they access such $\lceil x \rfloor$.

2.1.6 Transaction Legality

The definitions given above allow us to formulate the central concept that defines consistency of transactional execution: *transaction legality*. Intuitively, using variables as an example, we can say a transaction is legal in a sequential history if it only reads values of variables that were written by committed transactions or by itself.

More formally, let S be a sequential history that only contains committed transactions, with the possible exception of the last transaction, which can be aborted. We say that sequential history S is *legal* if for every $\lceil x \rfloor \in Obj$, $S|\lceil x \rfloor \in Seq(\lceil x \rfloor)$.

In addition, given any sequential history S and transaction $T_i \in S$, let *visible history* $Vis(S, T_i)$ be the longest subhistory of S s.t., for every transaction $T_j \in Vis(S, T_i)$, either $i = j$ or T_j is committed in S and $T_j \prec_S T_i$. Then, given a sequential history S and a transaction $T_i \in S$, we say that T_i is *legal in* S if $Vis(S, T_i)$ is legal.

2.1.7 Safety Properties

A *property* $\mathfrak{P}$ is a condition that stipulates correct behavior. In relation to histories, a given history satisfies $\mathfrak{P}$ if the condition is met for that history. Given property $\mathfrak{P}$, we call $\mathbb{H}_{\mathfrak{P}}$ the set of all $\mathfrak{P}$-histories, defined such that $H \in \mathbb{H}_{\mathfrak{P}}$ if, and only if H satisfies $\mathfrak{P}$. In relation to programs, program $\mathbb{P}$ satisfies $\mathfrak{P}$ if all histories produced by $\mathbb{P}$ satisfy $\mathfrak{P}$.

Safety properties [50] are properties which guarantee that "something [bad] will not happen." In the case of TM this means that, transactions will not observe concurrency of other transactions. Property $\mathfrak{P}$ is a safety property if it meets the following definition (adapted from [8]):

Definition 1. *A property $\mathfrak{P}$ is a* safety property *if, given the set $\mathbb{H}_{\mathfrak{P}}$ of all histories that satisfy $\mathfrak{P}$:*

a) Prefix-closure*: every prefix H' of a history $H \in \mathbb{H}_{\mathfrak{P}}$ is also in $\mathbb{H}_{\mathfrak{P}}$,*

b) Limit-closure*: for any infinite sequence of finite histories $H_0, H_1, ...$, s.t. for every $h = 0, 1, ...$, $H_h \in \mathbb{H}_{\mathfrak{P}}$ and H_h is a prefix of H_{h+1}, the infinite history that is the* limit *of the sequence is also in $\mathbb{H}_{\mathfrak{P}}$.*

For distinction, we use the term *consistency condition* to refer to properties that are not safety properties.

We compare properties with respect to their relative strength. Given two properties $\mathfrak{P}'$ and $\mathfrak{P}''$ we say $\mathfrak{P}'$ is *stronger than* $\mathfrak{P}''$ if $\mathbb{H}_{\mathfrak{P}'} \subset \mathbb{H}_{\mathfrak{P}''}$ (so $\mathfrak{P}''$ *is weaker than* $\mathfrak{P}'$). If neither $\mathbb{H}_{\mathfrak{P}'} \subset \mathbb{H}_{\mathfrak{P}''}$ nor $\mathbb{H}_{\mathfrak{P}'} \supset \mathbb{H}_{\mathfrak{P}''}$, then the properties are *incomparable*, which we denote $\mathbb{H}_{\mathfrak{P}'} \| \mathbb{H}_{\mathfrak{P}''}$.

2.1.8 Early Release

Early release pertains to a situation where conflicting transactions execute partially in parallel while accessing the same variable. The implied intent is for all such transactions to access these variables without losing consistency and thus for them all to finally commit.

Our definitions are based on the observed effects of the release without reference to the actions of a concurrency control algorithm. That is a variable is considered to be released

early by some transaction only when another transaction views the modifications applied to the variable by the first transaction. We define the concept of early release as follows:

Definition 2 (Early Release)**.** *Given history H (with unique writes), transaction $T_i \in H$* releases variable x early *in H iff there is some prefix P of H, such that T_i is live in P and there exists some transaction $T_j \in P$ such that there is a complete non-local read operation execution $op_j = r_j(x) \to v$ in $P|T_j$ and a complete write operation execution $op_i = w_i(x)v \to ok_i$ in $P|T_i$ such that $op_i \prec_P op_j$.*

For the most part, we are concerned with early release to the extent it is used with variables, but the definition can nevertheless be extended to shared objects with unknown semantics:

Definition 3 (Strong Early Release)**.** *Given history H (with unique writes), transaction $T_i \in H$* releases object $\lceil x \rfloor$ early *in H iff there is some prefix P of H, such that T_i is live in P and there exists some transaction $T_j \in P$ such that $P|T_i$ and $P|T_j$ each contain a complete operation execution on $\lceil x \rfloor$, op_i and op_j respectively, such that $op_i \prec_P op_j$.*

Since both concepts are analogous but pertain to different models, we will shorten strong early release to early release without confusion.

2.1.9 Locks

Locks are used to block progress of a process as part of a TM algorithm. They are shared objects but are not accessed transactionally. We denote locks as either global $\texttt{lk}^g$ or associated with some variable $\texttt{lk}(x)$ or object $\texttt{lk}(\lceil x \rfloor)$. Here, we discuss locks associated with variables as examples, but the behavior and notation are analogous for global locks and locks associated with objects.

Each lock $\texttt{lk}(x)$ has read-write semantics. That is, each lock is initially *unlocked* and it can be acquired in *write mode* (also called *exclusive mode*) or in *read mode* (also called *shared mode*). Only one process can acquire the lock in write mode at a time. If a lock is already owned, then other processes trying to acquire it in write mode wait until it is released. However, if the lock is acquired in read mode, then other processes can also simultaneously successfully acquire it in read mode (although processes attempting to acquire the lock in write mode at the same time, still wait until the lock is unlocked.) Because several processes can acquire a lock in read mode simultaneously, but only one process can acquire a lock in write mode, we say read and write modes are *conflicting*. Given a lock $\texttt{lk}(x)$ we denote its state as $\texttt{mode}(\texttt{lk}(x))$, whose value can be either W to indicate the lock is acquired in write mode, R for read mode, or $\perp$ if the lock is unlocked.

Locks support the following operations:

a) *acquire* operation *in write mode*, $\texttt{lock lk}(x) \to W$,

b) *acquire* operation *in read mode*, $\texttt{lock lk}(x) \to R$,

c) *release* operation, $\texttt{unlock lk}(x)$,

d) *convert* operations, $\texttt{convert lk}(x) \to W$ and $\texttt{convert lk}(x) \to R$.

Convert operations are used to change (escalate or de-escalate) the state of a lock that is already owned by a given process. If other processes share the ownership of the lock with the current process, a convert operation may block the invoking process. Locks can also be used with try-lock semantics by using the following operations:

e) *try-lock* operation *in write mode*, $\texttt{try lock lk}(x) \to W$,

f) *try-lock* operation *in read mode*, $\texttt{try lock lk}(x) \to R$.

If a process attempts to try-lock a lock, but the lock cannot be acquired instantly, then, instead of waiting, a try-lock operation immediately returns the Boolean value of `false`. If the lock is acquired successfully without waiting, the lock is acquired and the operation returns the value of `true`.

The lock's current owner is denoted $\texttt{owner}(\texttt{lk}(x))$. We will attribute lock ownership to specific transactions (rather than processes that execute them) that successfully executed the acquire operation. That is, if some transaction T_i acquired lock $\texttt{lk}(x)$ (but not yet released it), we write $\texttt{owner}(\texttt{lk}(x)) = T_i$.

We say T_i is waiting for T_j to release $\texttt{lk}(x)$ if T_i is in the process of executing $\texttt{lock lk}(x) \rightarrow W$ and the lock is owned by T_j or if T_i is in the process of executing $\texttt{lock lk}(x) \rightarrow R$ and the lock is owned by T_j in a conflicting mode. Given a set of transactions $\mathbb{T}_d$ we say a *deadlock occurs* for $\mathbb{T}_d$ if for some set of transactions $\mathbb{T}'_d \subseteq \mathbb{T}_d$ defined as $\mathbb{T}'_d = \{T_1, T_2, T_3, ..., T_n\}$, T_1 is waiting for T_2 to release some lock $\texttt{lk}(x)$, tr_2 is waiting for T_3 to release some lock $\texttt{lk}(y)$, ..., and T_n is waiting for T_1 to release some lock $\texttt{lk}(z)$. We say that some history H *contains a deadlock* if, given the set $\mathbb{T}_H$ of all transactions in H, a deadlock occurs for $\mathbb{T}_H$. We say that a TM system is *deadlock-free* if there does not exist any history H produced by the TM, s.t. H contains a deadlock.

2.1.10 Buffers

Buffers are transaction-local structures used to commute the effects of operation execution on shared objects. We use two types of buffers: *copy buffers* and *log buffers* (also known as *redo log buffers*). We give examples using buffers for objects; buffers for variables are defined analogously.

For some transaction T_i a copy buffer for object $\lceil x \rfloor$ is a transaction-local object denoted $\texttt{buf}_i(\lceil x \rfloor)$ or $\texttt{st}_i(\lceil x \rfloor)$ that has the interface $\texttt{M}_{\lceil x \rfloor}$, and whose state is defined by analogy to state $\texttt{S}_{\lceil x \rfloor}$. We discuss the buffer denoted $\texttt{buf}_i(\lceil x \rfloor)$ below, but $\texttt{st}_i(\lceil x \rfloor)$ and $\texttt{buf}_i(\lceil x \rfloor)$ are analogous. Initially $\texttt{buf}_i(\lceil x \rfloor) = \perp$. Object $\lceil x \rfloor$ can be *copied to* $\texttt{buf}_i(\lceil x \rfloor)$, which means that the state of $\texttt{buf}_i(\lceil x \rfloor)$ becomes equal to $\texttt{S}_{\lceil x \rfloor}$. Similarly, $\texttt{buf}_i(\lceil x \rfloor)$ can be *copied from* $\lceil x \rfloor$ (we also say $\lceil x \rfloor$ is *restored from* $\texttt{buf}_i(\lceil x \rfloor)$) which means that $\texttt{S}_{\lceil x \rfloor}$ becomes equal to the state of $\texttt{buf}_i(\lceil x \rfloor)$. Once $\lceil x \rfloor$ is copied to $\texttt{buf}_i(\lceil x \rfloor)$, transaction T_i can execute operations from $\texttt{M}_{\lceil x \rfloor}$ on $\texttt{buf}_i(\lceil x \rfloor)$, which causes the operations' code to execute and view and modify the state of $\texttt{buf}_i(\lceil x \rfloor)$ in accordance to the semantics of the executed operations.

A log buffer for object $\lceil x \rfloor$ is a transaction-local object denoted $\texttt{log}_i(\lceil x \rfloor)$ for some transaction T_i that has the same interface as $\lceil x \rfloor$. The state of $\texttt{log}_i(\lceil x \rfloor)$ is a sequence of operations. At any time transaction T_i can execute operations from $\texttt{M}_{\lceil x \rfloor}$ on $\texttt{log}_i(\lceil x \rfloor)$, which causes the operation to be appended to the sequence of operations $\texttt{log}_i(\lceil x \rfloor)$. Log buffer $\texttt{log}_i(\lceil x \rfloor)$ can be *applied to* $\lceil x \rfloor$, which means each operation in the sequence of $\texttt{log}_i(\lceil x \rfloor)$ is executed on $\lceil x \rfloor$. This is done sequentially and preserving the order in which the operations were executed on $\texttt{log}_i(\lceil x \rfloor)$.

We distinguish two approaches to the use of buffers by TM. *Encounter-time* TM algorithms apply the effects of operation executions on shared objects immediately at the point where the operation is executed. That is, when a transaction finishes executing some operation `m` on some object, any side effects resulting from the execution of `m` already manifested, and all modifications to the object resulting from the execution are already reflected in the state of the variable. In *commit-time* TM algorithms transactions generally perform operation executions using local buffers, so that the effects of the execution may be deferred to some later point during the transaction execution (typically to transaction commit).

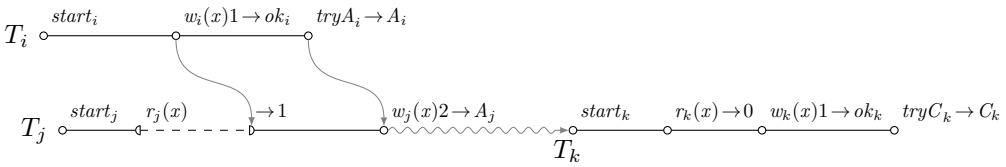


Figure 2.1: Transaction diagram for H_2.

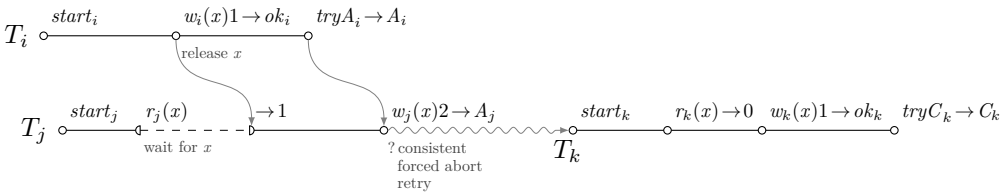


Figure 2.2: Transaction diagram for H_2 with implementation notes.

2.1.11 Approach to Concurrency Control

TM algorithms can either employ *pessimistic* or *optimistic* concurrency control (sometimes referred to as *aggressive* and *conservative* [12, 91]). In pessimistic TM operations on shared variables tend to be delayed in order to prevent situations where their executions can cause inconsistencies within the system state. In optimistic TM operations on shared variables or objects tend to be executed speculatively: optimistic TM systems generally avoid delaying operations, but instead they execute operations as soon as they are invoked. Since this can cause inconsistencies in the system, optimistic transactions perform validation (at some point) before committing. If the validation fails, the transaction is aborted and retried.

2.1.12 Strong Progressiveness

Let $\mathbb{T}^c_H$ is be the set of all subsets Q of all transactions in a history H, such that Q is not empty and no transaction in Q conflicts with any transaction not in Q. Given transaction T_i, let $Obj^c_H(T_i)$ be a set of shared objects, such that object $\lceil x \rfloor \in Obj^c_H(T_i)$ iff there exists a transaction T_j $(i \neq j)$ in history H that (strongly) conflicts with transaction T_i on shared object $\lceil x \rfloor$. Given a set of transactions Q, $Obj^c_H(Q)$ is a union $\bigcup_{T_k \in Q} Obj^c_H(T_k)$.

Given these sets, history H is *strongly progressive* iff, for every set $Q \in \mathbb{T}^c_H$ such that $|Obj^c_H(Q)| \leqslant 1$, at least one transaction in Q is not forcibly aborted in H.

2.2 Transaction Diagrams

When talking about examples of histories, it is easier to understand the relationships between various events if the history is depicted using diagrams. For example, the following history is represented in the diagram in Fig. 2.1:

$$H_2 = \Big[\ start_i \to ok_i, start_j \to ok_j, inv_j\big[r_j(x)\big],$$
$$w_i(x)1 \to ok_i, res_j\big[v\big], tryA_i \to A_i, w_j(x)2 \to A_j,$$
$$start_k \to ok_k, r_k(x) \to 0, w_k(x)1 \to ok_k, tryC_k \to C_k\ \Big].$$

This and other diagrams each depict a history consisting of operations executed by transactions on a time axis. Every line depicts the operations executed by a particular transaction. The symbol –○– denotes a complete operation execution. The inscriptions above operation executions denote operations executed by the transactions, e.g. $r_i(x) \to 0$ denotes that a read operation on variable x is executed by transaction T_i and returns 0, and $w_i(x)1 \to ok_i$ denotes that a write operation writing 1 to x is executed by T_i, and $tryC_i \to C_i$ indicates that T_i attempts to commit and succeeds because it returns C_i, whereas $tryA_i \to A_i$ indicates that the transaction attempts to abort and succeeds, etc. On the other hand, the symbol ⊸ ⊳ denotes an operation execution split into the invocation and the response event to indicate waiting, or that the execution takes a long time. In that case the inscription above is split between the events, e.g. a read operation execution would show $r_i(x)$ above the invocation, and $\to 1$ over the response.

The diagram also adds additional information to the history to emphasize relationships between events. If waiting is involved, the arrow ⤷ is used to emphasize a happens before relation between two events. The same is used to indicate causality, e.g. whenever an abort event forces another operation to abort. Furthermore, ⇝ denotes that the preceding transaction aborts (here, T_j) and a new transaction (T_k) is spawned. These elements are used as necessary to indicate particular scenarios and may be omitted.

In addition, annotations below events may be used to indicate computations performed within a given TM implementation as part of the concurrency control algorithm. We show an example in Fig. 2.2. The notation within these comments follows from the convention used for pseudocode, with the exception that conditions will be marked by a preceding ? mark (to save space).

3
Existing Properties

In this chapter we discuss a number of prominent TM safety properties, as well as some applicable database consistency conditions, that can be used to determine the correctness of the algorithms' behavior, with a strong focus on how the properties in question regulate behavior of TM algorithms that use the early release technique. Specifically, we examine the properties to find out whether or not they allow transactions to use the early release technique, and, if so, to what extent. For this purpose, we first define a set of tools which we use for the examination, and then proceed to employ them on each property. We summarize our findings at the end of the chapter. This chapter extends our previous analysis in [77] and [79].

3.1 Analysis Parameters

The aim of the analysis is to find properties that describe the guarantees of TM systems with early release that can be applied in practice. We seek a safety property that allows early release, but, nevertheless, reduces or eliminates undesired behaviors.

Early Release Support

We begin our analysis by defining its key questions. The first and the most obvious is whether a particular property supports early release at all. Early release pertains to a situation where conflicting transactions execute partially in parallel while accessing the same variable. The implied intent is for all such transactions to access these variables without losing consistency and thus for them all to finally commit. We define early release formally in Def. 2. Then, the ability for a property to support early release is defined as follows:

Definition 4 (Early Release Support)**.** *Property $\mathfrak{P}$ supports early release iff given some history H that satisfies $\mathfrak{P}$ there exists some transaction $T_i \in H$, s.t. T_i releases some variable x early in H.*

If a property allows early release, it allows a significant performance boost (see e.g. [65, 75]) as transactions are executed with a higher degree of parallelism.

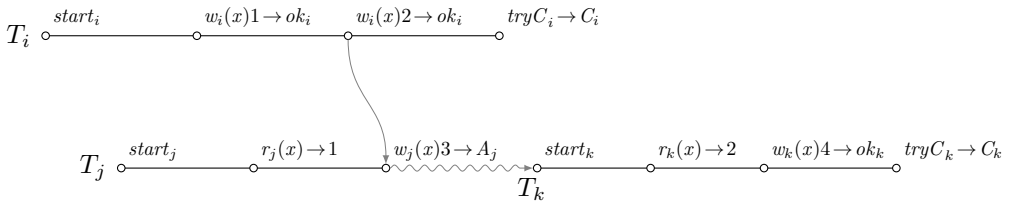


Figure 3.1: A history with early release and overwriting.

Overwriting Support

Early release can give rise to some unwanted or unintuitive scenarios with respect to consistency. The most egregious of these is *overwriting*, where one transaction releases some variable early, but proceeds to modify it afterward. In that case, any transaction that started executing operations on the released variable will observe an intermediate value with respect to the execution of the other transaction, ie., *view inconsistent state*.

An example of overwriting is shown in Fig. 3.1, where transaction T_i releases variable x early but continues to write to x afterward. As a consequence, T_j first reads the value of x that is later modified. When T_j detects it is in conflict while executing a write operation it is aborted. This is a way for the TM to attempt to mitigate the consequences of viewing inconsistent state. The transaction is then restarted as a new transaction T_k.

However, as argued in [33], simply aborting a transaction that views inconsistent state is not enough, since the transaction can potentially act in an unpredictable way on the basis of using an inconsistent value to perform local operations. For instance, if the value is used in pointer arithmetic it is possible for the transaction to access an unexpected memory location and crash the process. Alternatively, if the transaction uses the value within a loop condition, it can enter an infinite loop and become parasitic. Other dangerous behaviors are possible, including division by zero precluded by invariants that assume atomicity of transactions.

Thus, in our analysis of existing properties we ask the question whether, apart from allowing early release, the properties also forbid overwriting. In the light of the potential dangerous behaviors that can be caused by it, we consider properties that allow overwriting to be too weak to be practical.

Definition 5 (Overwriting Support)**.** *Property $\mathfrak{P}$ supports overwriting iff $\mathfrak{P}$ supports early release, and given some history H (with unique writes) that satisfies $\mathfrak{P}$, for some pair of transactions $T_i, T_j \in H$ s.t.,*

a) *T_i releases some variable x early,*

b) *$H|T_i$ contains two write operation executions: $w_i(x)v \to ok_i$ and $w_i(x)v' \to ok_i$, s.t. the former precedes the latter in $H|T_i$,*

c) *$H|T_j$ contains a read operation execution $r_j(x) \to v$ that precedes $w_i(x)v' \to ok_i$ in H.*

Aborting Early Release Support

In addition, we look at whether or not a particular property forbids a transaction that releases some variable early to abort. This is a precaution taken by many properties to prevent *cascading aborts*, another type of scenario leading to inconsistent views. An example of this is shown in Fig. 3.2. In such a case a transaction, here T_i, releases a variable early and subsequently aborts. This can cause another transaction T_j that

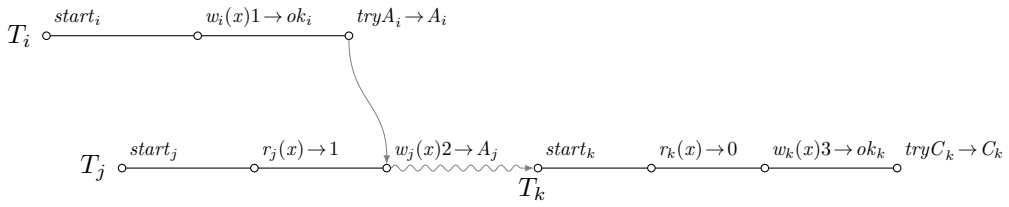


Figure 3.2: A history with early release and cascading abort.

executed operations on that variable in the meantime to observe inconsistent state. In order to maintain consistency, a TM will then typically force T_j to abort and restart as a result.

However, while the condition that no transaction that releases early can abort, solves the problem of cascading aborts, it significantly limits the usefulness of any TM that satisfies it, since TM systems typically cannot predict whether any particular transaction eventually commits or aborts. In particular, there are important applications for TM, where a transaction can arbitrarily and uncontrollably abort at any time. Such applications include distributed TM and hardware TM, where aborts can be caused by outside stimuli, such as machine crashes.

An exception to this may be found in systems making special provisions to ensure that irrevocable transactions eventually commit (see e.g., [92]). In such systems, early release transactions could be ensured never to abort. However, case in point, these take drastic measures to ensure that, e.g., at most a single irrevocable transaction is present in the system at one time. Therefore, the requirement is too strict in the general case.

Definition 6 (Aborting Early Release Support)**.** *Property $\mathfrak{P}$ supports aborting early release iff $\mathfrak{P}$ supports early release, and given some history H that satisfies $\mathfrak{P}$, for some transaction $T_i \in H$ that releases some variable x early, $H|T_i$ contains A_i.*

3.2 Properties

In this section we analyze the extent to which various properties support early release, and what restrictions they apply to transactions that release variables early. The properties under consideration are the typical TM safety properties: serializability, opacity, markability, virtual world consistency, transactional memory specification, live opacity, and elastic opacity. We also consider some strong database consistency conditions that pertain to transactional processing: recoverability, commitment order preservation, cascadelessness, strictness, and rigorousness.

3.2.1 Serializability

The first property we discuss is serializability, a database property which can be regarded as a baseline TM safety property. It can be considered the minimal strong property acceptable in TM. It is also a property that can be grasped intuitively: a history is serializable if there is some sequential execution that would reflect the same behavior as shown in that history.

Serializability is defined formally in [60] in three variants: *conflict serializability*, *view serializability*, and *final-state serializability*. We follow a more general version of serializ-

ability defined in [90] (as *global atomicity*), which we adjust to account for non-atomicity of commits in our model.

Definition 7 (Serializability)**.** *History H is serializable iff there exists some sequential history S equivalent to a completion $Compl(H)$ such that any committed transaction in S is legal in S.*

Intuitively, the definition does not preclude early release, as long as illegal transactions are aborted. Serializability also makes no further stipulations on aborting transactions, so it permits both overwriting and cascading aborts.

Lemma 1. *Serializability supports early release.*

Proof. Let H be a transactional history as shown in Fig. 3.1. Note that since all transactions in H are committed or aborted then $H = Compl(H)$. Then, let there be a sequential history $\hat{S}_H = H|T_i \cdot H|T_j \cdot H|T_k$. Note that $\hat{S}_H \equiv H$. Trivially, all the committed transactions, T_i and T_k, in $\hat{S}_H$ are legal in $\hat{S}_H$, so H is serializable. Since, by Def. 2, T_i releases early in H, then, by Def. 4, serializability supports early release. □

Lemma 2. *Serializability supports overwriting.*

Proof. Let H be a serializable history as in the proof of Lemma 1 above. Transaction T_i writes 1 to x in H prior to T_j reading 1 from x, and subsequently T_i writes 2 to x. Thus, according to Def. 5, serializability supports overwriting. □

Lemma 3. *Serializability supports aborting early release.*

Proof. Let H be a history such as the one in Fig. 3.2. Since all transactions in H are committed or aborted then $H = Compl(H)$. Then, let $\hat{S}_H$ be a sequential history equivalent to H such that $\hat{S}_H = H|T_i \cdot H|T_j \cdot H|T_k$. $\hat{S}_H$ contains only one committed transaction T_k, which is trivially legal in $\hat{S}_H$. Thus H is serializable. In addition, transaction T_i in $\hat{S}_H$ both releases x early (Def. 2) and contains an abort ($A_i \in H|T_i$). Thus, by Def. 6, serializability supports aborting early release. □

There is also a variant of serializability called *strict serializability* that adds the condition that the witness history S which justifies the serializability of history H must also preserve the real-time order of H. The results above trivially extend to this variant.

3.2.2 Commitment Order Preservation

Commitment order preservation (*CO*) is a database consistency condition, which requires that transactions commit in the same order as the order in which the transactions accessed variables. It is often used as an additional condition to serializability. Formally, CO is defined as follows (adapted from [91]):

Definition 8 (Commitment Order Preservation)**.** *History H* preserves commitment order *iff for any two committed conflicting transactions $T_i, T_j \in H$ s.t. $i \neq j$ given any pair of conflicting operation executions $op_i \in H|T_i$ and $op_j \in H|T_j$, either $op_i \prec_H op_j$ and $res_i[C_i] \prec_H res_j[C_j]$, or $op_j \prec_H op_i$ and $res_j[C_j] \prec_H res_i[C_i]$.*

CO maintains the order of their commits with respect to the order in which they access operations, but it makes no stipulations regarding aborted transactions, which allows them to read from live transactions. Thus, early release is generally allowed under commit ordering.

Lemma 4. *Commitment order supports early release.*

Proof. Let H be a transactional history as shown in Fig. 3.1. Here, all operations in T_i conflict with all operations in T_j, and all operations in T_i conflict with all operations in T_k. In addition, transactions T_i and T_k commit. Since T_k performs its operations on the shared variable after T_i, then T_k must commit after T_i. Since this is the case, H preserve commitment order (Def. 8). Since, by Def. 2, T_i releases early in H, then, by Def. 4, commit ordering supports early release. □

Lemma 5. *Commitment order supports overwriting.*

Proof. By analogy to Lemma 4. □

Lemma 6. *Commitment order supports aborting early release.*

Proof. Let H be a history such as the one in Fig. 3.2. Here, only transaction T_k commits, so trivially, the history preserver commitment order by Def. 8. Transaction T_i in H releases x early (Def. 2) and contains an abort ($A_i \in H|T_i$). Thus, by Def. 6, CO supports aborting early release. □

Note that, a composition of CO with either serializability or recoverability (see below) trivially also allows early release, overwriting, and aborting early release.

3.2.3 Recoverability

Recoverability is another database consistency condition used in conjunction with serializability. Recoverability requires that transactions only commit after other transactions whose changes they read commit. It is defined as below (following [36]):

Definition 9 (Recoverability)**.** *History H is* recoverable *iff for any $T_i, T_j \in H$, s.t. $i \neq j$ and T_j reads from T_i, T_i commits in H before T_j commits.*

Recoverability requires that transactions only commit after other transactions whose changes they read commit, which does not impinge on the ability to release early.

Lemma 7. *Recoverability supports early release.*

Proof. Let H be a transactional history as shown in Fig. 3.1. Here, transaction T_j reads from T_i and T_k reads from T_i, and no other transactions are in the reads-from relation. If H is recoverable, then, by Def. 9, T_i must commit before T_j commits and before T_k commits. This condition is true for T_i and T_j, since T_j never commits. The condition is trivially true for T_i and T_k. Hence, H is recoverable. Since, by Def. 2, T_i releases early in H, then, by Def. 4, recoverability supports early release. □

Lemma 8. *Recoverability supports overwriting.*

Proof. By analogy to Lemma 7. □

Lemma 9. *Recoverability supports aborting early release.*

Proof. Let H be a history such as the one in Fig. 3.2. Here, transaction T_j reads from T_i and no other transactions are in the reads-from relation. Since T_i and T_j both abort, then, the condition in Def. 9, is trivially true for H. Hence, H is recoverable. Transaction T_i in H releases x early (Def. 2) and contains an abort ($A_i \in H|T_i$). Thus, by Def. 6, recoverability supports aborting early release. □

Note that, a composition of recoverability and serializability or commitment order preservation trivially also allows early release, overwriting, and aborting early release.

3.2.4 Cascadelessness

Cascadelessness (also known as *avoiding cascading aborts* or *rollbacks—ACA*, *ACR*) is a database consistency condition that is used to exclude scenarios where one aborting transaction T_i forces another transaction T_j to abort, because T_j read from T_i before T_i aborted. It is used to impose additional requirements on serializable executions. It is defined as follows (after [12]):

Definition 10 (Cascadelessness)**.** *History H is* cascadeless *iff for any $T_i, T_j \in H$ s.t. $i \neq j$ and T_j reads from T_i, T_i commits before the read.*

Cascadelessness restricts reading from live transactions. Therefore, cascadelessness clearly removes all the scenarios encompassed by Def. 2. Since this is the only provision of cascadelessness, the property forbids early release without giving any additional guarantees. Hence, it also does not support overwriting nor aborting early release.

Lemma 10. *Cascadelessness does not support early release.*

Proof. By contradiction let us assume that cascadelessness supports early release. Then, from Def. 4, there exists some history H, s.t. H is cascadeless and there exists some transaction $T_i \in H$ that releases some variable x early in H. From Def. 2, this implies that there exists some prefix P of H s.t.

a) there is an operation execution $op_i = w_i(x)v \to ok_i$ and $op_i \in P|T_i$,

b) there exists a transaction $T_j \in P$ $(i \neq j)$ and an operation execution $op_j = r_j(x) \to v$, s.t. $op_j \in P|T_j$ and op_i precedes op_j in P,

c) T_i is live in P.

These imply that op_j follows op_i in P in such a way that there does not exist in P an operation $op_c = tryC_i \to C_i$ in P s.t. $op_c \prec_P op_j$. Therefore, such op_c does not exist in H either. This contradicts Def. 10, which dictates that T_i must commit before T_j reads from T_i, so H is not cascadeless, which is a contradiction. □

Since both Def. 5 and Def. 6 require early release support, then:

Corollary 1. *Cascadelessness does not support overwriting.*

Corollary 2. *Cascadelessness does not support aborting early release.*

3.2.5 Strictness

Strictness [12] is a strong database property that, given a write in one transaction, and some other following operation in another transaction, that second operation can only be executed if the transaction executing the write already committed or aborted. Formally:

Definition 11 (Strictness)**.** *History H is* strict *iff for any $T_i, T_j \in H$ $(i \neq j)$ and given any operation execution $op_i = r_i(x) \to v$ or $op_i = w_i(x)v' \to ok_i$ in $H|T_i$, and any operation execution $op_j = w_j(x)v \to ok_j$ in $H|T_j$, if op_i follows op_j, then T_j commits or aborts before op_i.*

The definition unequivocally states that a transaction cannot read from another transaction, until the latter is committed or aborted. Thus, strictness precludes early release altogether.

Lemma 11. *Strictness does not support early release.*

Proof. By contradiction let us assume that strictness supports early release. Then, from Def. 4, there exists some history H, s.t. H is strict and there exists some transaction $T_i \in H$ that releases some variable x early in H. From Def. 2, this implies that there exists some prefix P of H s.t.

a) there is an operation execution $op_i = w_i(x)v \rightarrow ok_j$ and $op_i \in P|T_i$,

b) there exists a transaction $T_j \in P$ $(i \neq j)$ and an operation execution $op_j = r_j(x) \rightarrow v$, s.t. $op_j \in P|T_j$ and op_i precedes op_j in P,

c) T_i is live in P.

These imply that op_j follows op_i in P in such a way that there does not exist in P an operation op_c that returns either A_i or C_i and $op_i \prec_P op_c \prec_P op_j$. Therefore, there does not exist an operation op_c in H that returns either A_i or C_i and $op_i \prec_H op_c \prec_H op_j$. This contradicts Def. 11, so H is not strict, which is a contradiction. □

Since both Def. 5 and Def. 6 require early release support, then:

Corollary 3. *Strictness does not support overwriting.*

Corollary 4. *Strictness does not support aborting early release.*

Note, that while strictness does not allow early release as defined by Def. 2, it allows for parallel execution of reads by live transactions which can be considered a limited form of early release (e.g. [43]).

3.2.6 Opacity

Opacity [32, 33] can be considered *the* standard TM safety property that guarantees serializability and preservation of real-time order, and prevents reading from live transactions. It is defined by the following two definitions. The first definition specifies *final state opacity* that ensures the appropriate guarantees for a complete transactional history. The second definition uses final state opacity to define a safety property that is prefix-closed. Both definitions follow those in [33].

Definition 12 (Final state opacity)**.** *A finite TM history H is final-state opaque if, and only if, there exists a sequential history S equivalent to any completion of H s.t.,*

a) S preserves the real-time order of H,

b) every transaction T_i in S is legal in S.

Definition 13 (Opacity)**.** *A TM history H is opaque if, and only if, every finite prefix of H is final-state opaque.*

This definition of opacity forbids reading from live transactions, so it precludes any use of early release whatsoever.

Lemma 12. *Opacity does not support early release.*

Proof. By contradiction let us assume that opacity supports early release. Then, from Def. 4, there exists some history H (with unique writes), s.t. H is opaque and there exists some transaction $T_i \in H$ that releases some variable x early in H.

From Def. 2, this implies that there exists some prefix P of H s.t.

a) there is an operation execution $op_i = w_i(x)v \rightarrow ok_i$ and $op_i \in P|T_i$,

b) there exists a transaction $T_j \in P$ $(i \neq j)$ and an operation execution $op_j = r_j(x) \rightarrow v$, s.t. $op_j \in P|T_j$ and op_i precedes op_j in P,

c) T_i is live in P.

Let P_c be any completion of P. Since T_i is live in P, by definition of completion, it is necessarily aborted in P_c (ie. $A_i \in P_c|T_i$). Given any sequential history $\hat{S}_H$ equivalent to P_c, since T_i is aborted in P_c and $Vis(\hat{S}_H, T_j)$ only contains operations of committed transactions, then $P_c|T_i \not\subseteq Vis(\hat{S}_H, T_j)$. This means that $op_j \in Vis(\hat{S}_H, T_j)$ but $op_i \notin Vis(\hat{S}_H, T_j)$, so $Vis(S, T_j) \not\subseteq Seq(x)$ and therefore $Vis(\hat{S}_H, T_j)$ is not legal.

On the other hand, Def. 13 implies that any prefix P of H is final state opaque, which, by Def. 12, implies that there exists some completion P_c of P for which there exists an equivalent sequential history $\hat{S}_H$ s.t., any T_j in $\hat{S}_H$ is legal in $\hat{S}_H$. Since any T_j is legal then for any T_j, $Vis(\hat{S}_H, T_j)$ is also legal. But this is a contradiction with the paragraph above. Thus, there cannot exist a history like H that is both opaque and contains a transaction that releases some variable early. □

Since both Def. 5 and Def. 6 require early release support, then:

Corollary 5. *Opacity does not support overwriting.*

Corollary 6. *Opacity does not support aborting early release.*

It is worth noting that opacity precludes reading from live transactions regardless of whether the transactions in question ever abort, even given a transactional model where transaction aborts are outright impossible.

3.2.7 Markability

Markability [52] is a TM safety property equivalent to opacity (i.e. every opaque history is markable, and every markable history is opaque) introduced as a simpler way to prove opacity. Since every markable history is opaque, then it follows from Lemma 12, and Corollaries 5 and 6, that:

Corollary 7. *Markability does not support early release.*

Corollary 8. *Markability does not support overwriting.*

Corollary 9. *Markability does not support aborting early release.*

3.2.8 Rigorousness

Rigorousness is a strong database property which requires that given any two transaction executing operations on the same variable, the latter of them cannot execute any operations until the former commits or aborts. It is defined as a condition added onto strictness, as follows (following [15]):

Definition 14 (Rigorousness)**.** *History H is* rigorous *iff it is strict and for any $T_i, T_j \in H$ $(i \neq j)$ such that T_i writes to variable x, i.e., $op_i = w_i(x)v \rightarrow ok_i \in H|T_i$ after T_j reads x, then T_j commits or aborts before op_i.*

Since in [7] the authors demonstrate that rigorous histories are opaque, and since we show in Lemma 12 and Corollaries 5 and 6 that opaque histories do not support early release, then neither does rigorousness.

Corollary 10. *Rigorousness does not support overwriting.*

Corollary 11. *Rigorousness does not support overwriting.*

Corollary 12. *Rigorousness does not support aborting early release.*

3.2.9 Transactional Memory Specification

In [22] the authors argue that some scenarios, such as sharing variables between transactional and non-transactional code, require additional safety properties. Thus, they propose and rigorously define two consistency conditions for TM: *transactional memory specification 1 (TMS1)* and *transactional memory specification 2 (TMS2)*.

TMS1 follows a set of design principles including a requirement for observing consistent behavior that can be justified by some serialization. Among others, TMS1 also requires that partial effects of transactions are hidden from other transactions. These principles are reflected in the definition of the TMS1 automaton, and we paraphrase only parts relevant to our further discussion, i.e. the condition for the correctness of an operation's response in the following definitions (see the definitions of *extConsPrefix* and *validResp* for TMS1 in [22]).

Given history H and some response event r in H, let $H{\uparrow}r$ denote a subhistory of H, s.t. for every operation execution $op \in H$, $op \in H{\uparrow}r$ iff $op \prec_H r$ and op is complete. This represents all operations executed "thus far," when considering the legality of r.

Let $\mathbb{T}^c_H$ be the set of all such transactions that $T_k \in \mathbb{T}^c_H$ iff $T_k \in H$ and $inv_k\big[tryC_k\big] \in H|T_k$. Given response event r, let $\mathbb{T}^c_H{\uparrow}r$ be the set of all transactions in H s.t. $T_k \in \mathbb{T}^c_H{\uparrow}r$ if $T_k \in \mathbb{T}^c_H$ and $inv_k\big[tryC_k\big] \prec_H r$. These sets represent transactions which committed or aborted (but are not live) and the set of all such transactions that did so before response event r.

Given some history H, let $\mathbb{T}'_H$ by any subset of transactions in H. Let σ be a sequence of transactions. Let $ser(\mathbb{T}'_H, \prec_H)$ be a set of all sequences of transactions s.t. $\sigma \in ser(\mathbb{T}'_H, \prec_H)$ if σ contains every element of $\mathbb{T}'_H$ exactly once, and, for any $T_i, T_j \in \mathbb{T}'_H$, if $T_i \prec_H T_j$ then T_i precedes T_j in σ.

Given a history H and some response event r in H, let $ops(\sigma, r)$ be a sequence of operations s.t. if $\sigma = [T_1, T_2, ..., T_n]$ then $ops(\sigma, r) = H{\uparrow}r|T_1 \cdot H{\uparrow}r|T_2 \cdot ... \cdot H{\uparrow}r|T_n$. This represents the sequential history prior to response event r that respects a specific order of transactions defined by σ.

The most relevant condition in TMS1 with respect to our further discussion is the validity condition of individual response operations. A prerequisite for checking validity is to check whether a response event can be justified by some *externally consistent prefix*. This prefix consists of operations from all transactions that precede the response event and whose effects are visible to other transactions. Specifically, if a transaction precedes another transaction in the real time order, then it must be both committed and included in the prefix, or both not committed and excluded from the prefix. However, if a transaction does not precede another transaction, it can be in the prefix regardless of whether it committed or aborted.

Definition 15 (Externally Consistent Prefix)**.** *Given a history H and a response event r, let the externally consistent prefix $\mathbb{T}^r_H$ be any subset of all transactions in H s.t. for any $T_i, T_j \in \mathbb{T}^r_H$, if $T_i \prec_H T_j$ then T_i is in $\mathbb{T}^r_H$ iff $res_i\big[C_i\big] \in H{\uparrow}r|T_i$.*

TMS1 specifies that each response to an operation invocation in a safe history must be *valid*. Intuitively, a valid response event is one for which there exists a sequential prefix that is both legal and meets the conditions of an externally consistent prefix. More precisely, the following condition must be met:

Definition 16 (Valid Response)**.** *Given a transaction T_i in H, we say the response r to some operation invocation e in $H|T_i$ is valid if there exists set $\mathbb{T}^r_H \subseteq \mathbb{T}^c_H{\uparrow}r$ and sequence $\sigma \in ser(\mathbb{T}^r_H, \prec_H)$ s.t. $\mathbb{T}^r_H$ satisfies Def. 15 and $ops(\sigma \cdot T_i, r) \cdot [e \to r]$ is legal.*

Intuitively, TMS1 specifies that each response to an operation in a safe history must be *valid* (Def. 16), which means it is explained by a sequential prefix that is both legal

and meets the conditions of an externally consistent prefix (Def. 15). Since the externally consistent prefix excludes live transactions, then TMS1 does not allow early release in general. More formally:

Lemma 13. *TMS1 does not support early release.*

Proof. Assume by contradiction that TMS1 supports early release. Then by Def. 4, there exists some TMS1 history H s.t. $T_i, T_j \in H$ and there is a prefix P of H s.t. $op_i = w_i(x)v \to ok_i \in P|T_i$, $op_j = r_j(x) \to v \in P|T_j$, and T_i is live in H. This implies that $inv_i\big[tryC_i\big] \notin P{\uparrow}res_j\big[v\big]|T_i$. This means that $T_i \notin \mathbb{T}^c_H$ and therefore not in any $\mathbb{T}'_H \subseteq \mathbb{T}^c_H$ or, by extension, not in any $\sigma \in ser(\mathbb{T}'_H, \prec_H)$. Therefore, there is no op_i in $ops(\sigma, res_j\big[v\big])$, so, assuming unique writes, op_j is not preceded by a write of v to x in $ops(\sigma \cdot T_j, res_j\big[v\big]) \cdot [r_j(x) \to v]$. Therefore, $ops(\sigma \cdot T_j, res_j\big[v\big]) \cdot [r_j(x) \to v]$ is not legal, which contradicts Def. 16. □

Since both Def. 5 and Def. 6 require early release support, then:

Corollary 13. *TMS1 does not support overwriting.*

Corollary 14. *TMS1 does not support aborting early release.*

TMS2 is a stricter, but more intuitive version of TMS1. Since the authors show in [22] that TMS2 is strictly stronger than TMS1 (TMS2 implements TMS1), the conclusions above equally apply to TMS2. Hence, from Lemma 13:

Corollary 15. *TMS2 does not support early release.*

Corollary 16. *TMS2 does not support overwriting.*

Corollary 17. *TMS2 does not support aborting early release.*

3.2.10 Virtual World Consistency

The requirements of opacity, while very important in the context of TM's ability to execute any operation transactionally, can often be excessively stringent. On the other hand serializability is considered too weak for many TM applications. Thus, a weaker TM consistency condition called *virtual world consistency* (*VWC*) was introduced in [48].

Let us first define a partial order $\prec^{PO}_H$ on the set of all transactions in H. Given two transactions $T_i, T_j \in H$, $T_i \prec^{PO} T_j$ if:

a) T_i and T_j are executed by the same process p_k and $res^k_i\big[C_i\big] \prec_H inv^k_j\big[start_j\big]$, or

b) T_j reads from T_i, or

c) there exists some $T_l \in H$ such that $T_i \prec^{PO}_H T_l$ and $T_l \prec^{PO}_H T_j$.

The authors of [48] remark further that there is no $\prec^{PO}_H$ relation between T_i and T_j if T_i is aborted. We say sequential history S is a linear extension of H if $S \equiv H$ and the order of transactions in S preserves the partial order $\prec^{PO}_H$, i.e., $\prec^{PO}_H \subseteq \prec_S$. Then, the *causal past* $C(H, T_i)$ of some transaction T_i in some history H is such a history that includes T_i (i.e. $H|T_i \subseteq C(H, T_i)$ and includes every T_j (i.e. $H|T_j \subseteq C(H, T_i)$ s.t. $T_j \prec^{PO}_H T_i$.

Definition 17 (Virtual World Consistency)**.** *History H is virtual world consistent iff its subhistory containing all committed transactions is serializable and preserves real-time order, and for each aborted transaction T_i there exists a linear extension S of its causal past $C(H, T_i)$ that is legal.*

VWC allows limited support for early release as follows.

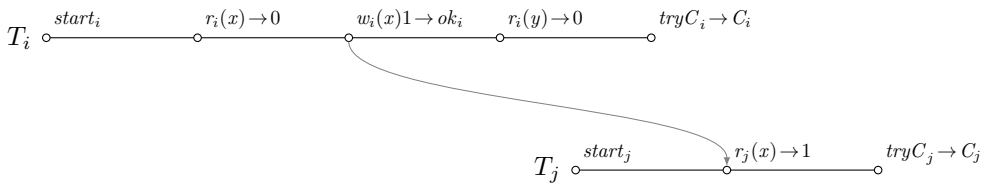


Figure 3.3: VWC history with early release.

Lemma 14. *VWC supports early release.*

Proof. Let H be a transactional history as shown in Fig. 3.3. Here, T_i performs two operations on x and one on y, while T_j reads x. The sequential witness history of H is $S = H|T_i \cdot H|T_j$ wherein both transactions are committed and trivially legal. Thus H is VWC. Since, by Def. 2, T_i releases early in H, then, by Def. 4, VWC supports early release. □

Lemma 15. *VWC does not support overwriting.*

Proof. Since VWC requires that aborting transactions view a legal causal past, then, if a transaction reading x is aborted, it must read a legal (i.e. "final") value of x. Thus, let us consider some history H (with unique writes) where some T_i writes value v to x and releases x early, and some T_j reads v from x (so T_j reads from T_i).

a) If T_i writes some value v' to x after releasing it, and T_j commits, then T_j is not legal in any sequential witness history of H because there is another write operation execution writing v' to x between a write writing v to x and a read on x returning v, and therefore H does not satisfy VWC.

b) If T_i writes some value v' to x after releasing it, and T_j aborts, then the causal past $C(H, T_j)$ contains T_i, and T_j is illegal in $C(H, T_j)$, because there is another write operation execution writing v' to x between a write writing v to x and a read on x returning v, so $C(H, T_j)$ is not legal. Thus, H does not satisfy VWC.

Therefore, any history H containing T_i, such that T_i releases x early and modifies it after release does not satisfy VWC. Hence, by Def. 5, VWC does not support overwriting. □

Lemma 16. *VWC does not support aborting early release*

Proof. Given a history H that satisfies VWC and a transaction $T_i \in H$ that releases variable x in H, let us assume for the sake of contradiction that T_i eventually aborts. By Def. 2, there is some T_j in H that reads from T_i. If T_i eventually aborts, then T_j reads from an aborted transaction.

a) If T_j eventually aborts, then its causal past $C(H, T_j)$ does not contain aborted transaction T_i and is, therefore, trivially illegal. Hence H does not satisfy VWC, which is a contradiction.

b) If T_j eventually commits, then the sequential witness history is also illegal. Hence H does not satisfy VWC, which is a contradiction.

Therefore, if T_i eventually aborts, H does not satisfy VWC, which is a contradiction. Thus, since a VWC history cannot contain an abortable transaction that releases a variable early. Hence, by Def. 6, VWC does not support aborting early release. □

While VWC supports early release, there are severe limitations to this capability. That is, VWC does not allow a transaction that released early to subsequently abort for any reason.

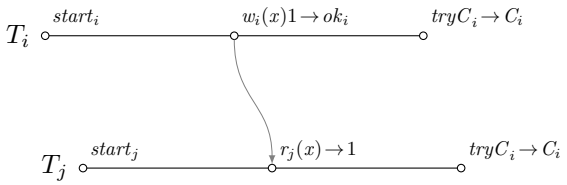


Figure 3.4: A live opaque history with early release.

3.2.11 Live Opacity

Live opacity was introduced in [26] as part of a set of consistency conditions and safety properties that were meant to regulate the ability of transactions to read from live transactions. The work analyzes a number of properties and for each one presents a commit oriented variant that forbids early release and a live variant that allows it. Here, we concentrate on live opacity, since it best fits alongside the other properties presented here, however our conclusions will apply to the remainder of live properties.

Let $H|(T_i, r)$ be the longest subsequence of $H|T_i$ containing only read operation executions (possibly pending), with the exclusion of the last read operation if its response event is A_i. Let $H|(T_i, gr)$ be a subsequence of $H|(T_i, r)$ that contains only non-local operation executions. Let T_i^r be a transaction that invokes the same transactional operations as those invoked in $[start_i \to ok_i] \cdot H|(T_i, r) \cdot [inv_i\big[tryC_i\big]]$ if $H|(T_i, r) \neq \varnothing$, or $\varnothing$ otherwise. Let T_i^{gr} be a transaction that invokes the same transactional operations as those invoked in $[start_i \to ok_i] \cdot H|(T_i, gr) \cdot [tryC_i \to C_i]$ if $H|(T_i, gr) \neq \varnothing$, or $\varnothing$ otherwise.

Given a history H, a transaction $T_i \in H$, and a complete local operation execution $op = r_i(x) \to v$, we say the latter's response event $res_i\big[v\big]$ is *legal* if the last preceding write operation in $H|T_i$ writes v to x. We say sequential history S justifies the serializability of history H when there exists a history H' that is a subsequence of H s.t. H' contains invocation and response events issued and received by transactions committed in H, and S is a legal history equivalent to H'.

Definition 18 (Live Opacity)**.** *A history H is live opaque iff, there exists a sequential history S that preserves the real time order of H and justifies that H is serializable and all of the following hold:*

a) We can extend history S to get a sequential history S' such that:
- *for each transaction $T_i \in H$ s.t. $T_i \notin S$, $T_i^{gr} \in S'$,*
- *if $<$ is a partial order induced by the real time order of S in such a way that for each transaction $T_i \in H$ s.t. $T_i \notin S$ we replace each instance of T_i in the set of pairs of the real time order of H with transaction T_i^{gr}, then S' respects $<$,*
- *S' is legal.*

b) For each transaction $T_i \in H$ s.t. $T_i \notin S$ and for each operation op in T_i^r that is not in T_i^{gr}, the response for op is legal.

Live opacity is a variant of opacity specifically introduced to allow early release, but only for non-aborting transactions. We show this below.

Lemma 17. *Live opacity supports early release.*

Proof. Let history H be that represented in Fig. 3.4. Since there is a transaction $T_i \in H$ that writes 1 to x and a transaction T_j that reads 1 from x before T_i commits, then there is a prefix P of H that meets Def. 2. Therefore T_i releases x early in H.

Let S be a sequential history s.t. $S = H|T_i \cdot H|T_j$. Since the real-time order of H is $\varnothing$, then, trivially, S preserves the real-time order of H. Since $Vis(S, T_i)$ contains only $H|T_i$ and therefore only a single write operation execution and no reads, then it is legal and T_i in S is legal in S. Furthermore, $Vis(S, T_j)$ is such that $Vis(S, T_j) = H|T_i \cdot H|T_j$ and contains a read operation $r_j(x) \to 1$ preceded by the only write operation $w_i(x)1 \to ok_i$, so $Vis(S, T_j)$ is legal, and, consequently, T_j in S is legal in S. Thus, all transactions in S are legal in S, so H is serializable.

Let S' be a sequential history that extends S in accordance to Def. 18. Since there are no transactions in S' that are not in S, then $S' = S$. Thus, since every transaction in S is legal in S, then every transaction in S' is legal in S'. Trivially, S' also preserves the real time order of S. Therefore, the condition Def. 18a is met. Since there are no local read operations in S, then condition Def. 18b is trivially met as well. Therefore, H is live opaque.

Since H is both live opaque and contains a transaction that releases early, then the lemma holds. □

Lemma 18. *Live opacity does not support overwriting.*

Proof. For the sake of contradiction, let us assume that there is a history (with unique writes) H that is live opaque and, from Def. 5, contains some transaction T_i that writes value v to some variable x and releases x early and subsequently executes another write operation writing v' to x where the second write follows a read operation executed by transaction T_j reading v from x.

Since H is live opaque there exists a sequential history S that justifies the serializability of H. There cannot exist a sequential history S where T_j reads from x between two writes to x executed by T_i, because there cannot exist a legal $Vis(S, T_j)$, so T_j would not be legal in S. Therefore, T_j must be aborted in H and therefore T_j is not in any sequential history S that justifies the serializability of H.

Since T_j is in H but not in S, then given any sequential extension S' of S in accordance to Def. 18, T_j is replaced in S' by T_j^{gr} which reads v from x and finally commits. However, since the write operation execution writing v to x in T_i is followed in $S'|T_i$ by another write operation execution that writes v' to x, then there cannot exist a $Vis(S', T_j^{gr})$ that is legal. Thus T_j^{gr} in S' cannot be legal in S', which contradicts Def. 18a. Thus, H is not live opaque, which is a contradiction.

Therefore H cannot simultaneously be live opaque and contain a transaction with early release and overwriting. □

Lemma 19. *Live opacity does not support aborting early release.*

Proof. For the sake of contradiction, let us assume that there is a history (with unique writes) H that is live opaque and, from Def. 6, contains some transaction T_i that writes value v to some variable x and releases x and subsequently aborts in H.

Let S be any sequential history that justifies the serializability of H, and let S' be any sequential extension S' of S in accordance to Def. 18. Since T_i aborts in H, then it is not in S, and therefore it is replaced in S' by T_i^{gr}. Since T_i^{gr} does not contain any write operation executions, there is no write operation execution writing v to x in S'. Since T_i released x early in H there is a transaction T_j in H that executes a read operation reading v from x and the same read operation is in S'. But since there is no write operation execution writing v to x in S', no transaction containing a read operation execution reading v from x can be legal in S'. Thus, H is not live opaque, which is a contradiction.

Therefore H cannot be simultaneously live opaque and contain a transaction with early release that aborts. □

In addition, note that, if some transaction T_i in some history H reads from a live transaction and is itself live, then there cannot be any transaction T_j that reads from T_i according to live opacity. This is because if T_i is replaced in S' with T_i^{gr}, then whatever value T_j reads from T_i will not be written by T_i^{gr}, so the read in T_j (or T_j^{gr}) may not be legal. We consider this to be overstrict, because the requirement that transactions which release early already are not allowed to abort. This means that transactions that read from live transactions will not experience inconsistent views regardless of whether that live transaction reads only from committed transactions or whether it reads from some yet another live transaction—all such live transactions must eventually commit. But if the live transaction reads from yet another live transaction, the history is unnecessarily precluded by live opacity, despite no inconsistent views being introduced by it.

3.2.12 Elastic Opacity

Elastic opacity is a safety property based on opacity, that was introduced to describe the safety guarantees of elastic transactions [28]. The property allows to relax the atomicity requirement of transactions to allow each of them to execute as a series of smaller transactions.

An *elastic transaction* T_i is split into a sequence of subhistories called a *cut* denoted $\mathcal{C}_i(H)$, where each subhistory represents a "subtransaction." In brief, a cut that contains more than one operation execution is *well-formed* if all subhistories are longer than one operation execution, all the write operations are in the same subhistory, and the first operation execution on any variable in every subhistory is not a write operation, except possibly in the first subhistory. A well-formed cut of some transaction T_i is *consistent* in some history H, if given any two operation executions op_i and op_i' on x in any subhistories of the cut, no transaction T_j $(i \neq j)$ executes a write operation op_j on x s.t. $op_i \prec_H op_j \prec_H op_i'$. In addition, given any two operation executions op_i and op_i' on x, y respectively, no two transactions T_k, T_l $(l \neq i,\ k \neq i)$ execute writes op_k on x and op_l on y, s.t. $op_i \prec_H op_k \prec_H op_i'$ and $op_i \prec_H op_l \prec_H op_i'$. A *cutting function* $f_{\mathcal{C}}$ takes a history H as an argument and produces a new history H_f where for each transaction $T_i \in H$ declared as elastic, T_i is replaced in H_f with the transactions resulting from the cut $\mathcal{C}_i(H)$ of T_i. If some transaction is committed (aborted) in H, then all transactions resulting from its cut are committed (aborted) in $f_{\mathcal{C}}(H)$. Then, elastic opacity is defined as follows:

Definition 19 (Elastic Opacity)**.** *History H is elastic opaque iff there exists a cutting function $f_{\mathcal{C}}$ that replaces each elastic transaction T_i in H with its consistent cut $\mathcal{C}_i(H)$, such that history $f_{\mathcal{C}}(H)$ is opaque.*

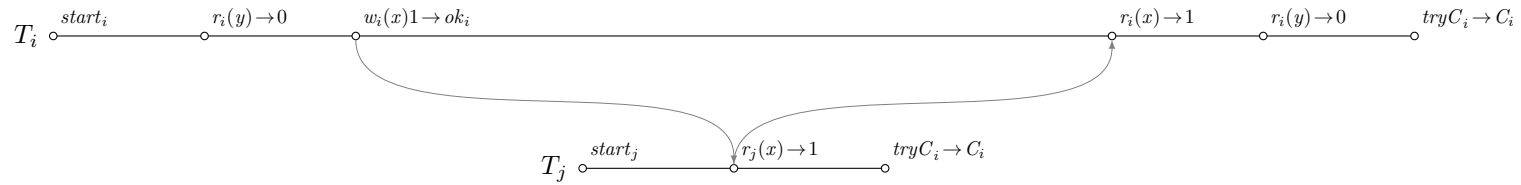
Note that the authors demonstrate in [28] that elastic opacity is not strictly stronger than serializability.

Elastic opacity (Def. 19) checks the validity of a history, not by validating the consistency of transactions, but of fragments of transactions. Hence, elastic opacity supports early release. A formal demonstration follows.

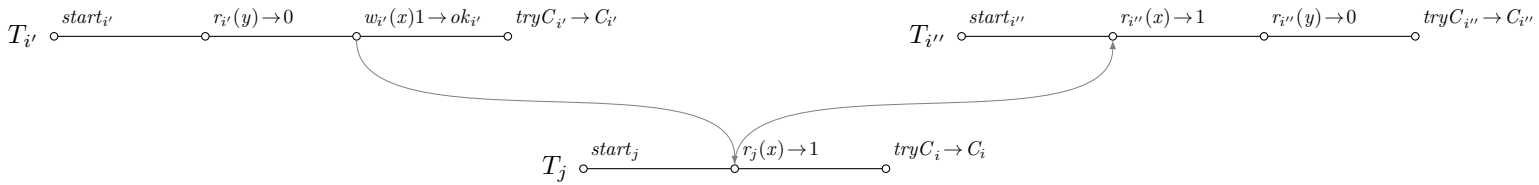
Lemma 20. *Elastic opacity supports early release.*

Proof. Let H be a transactional history with unique writes as shown in Fig. 3.5a. Let T_i be an elastic transaction. Let $\mathcal{C}_i(H)$ be a cut of subhistory $H|T_i$, such that:

$$\mathcal{C}_i(H) = \Big\{ \big[start_{i'} \to ok_{i'}, r_{i'}(y) \to 0, w_{i'}(x)1 \to ok_{i'}, tryC_{i'} \to C_{i'}\big], \\ \big[start_{i''} \to ok_{i''}, r_{i''}(x) \to 1, r_{i''}(y) \to 0, tryC_{i''} \to C_{i''}\big] \Big\}.$$



(a) History.



(b) History after applying a cutting function.

Figure 3.5: An elastic opaque history with early release.

All subhistories of $\mathcal{C}_i(H)$ are longer than one operation, all the writes are in the first subhistory, and no subhistory starts with a write, so $\mathcal{C}_i(H)$ is well-formed. Since there are no write operations outside of T_i, then it follows that $\mathcal{C}_i(H)$ is a consistent cut in H. Let $f_{\mathcal{C}}$ be any cutting function such that it cuts T_i according to $\mathcal{C}_i(H)$, in which case $f_{\mathcal{C}}(H)$ is defined as in Fig. 3.5b. Let S be a sequential history s.t. $S = f_{\mathcal{C}}(H)|T_{i'} \cdot f_{\mathcal{C}}(H)|T_j \cdot f_{\mathcal{C}}(H)|T_{i''}$. Since $T_{i'}$ precedes $T_{i''}$ in S as well as in $f_{\mathcal{C}}(H)$, and all other transactions are not real time ordered, S preserves the real time order of $f_{\mathcal{C}}(H)$. Trivially, each transaction in S is legal in S. Thus, $f_{\mathcal{C}}(H)$ is opaque by Def. 13, and in effect H is elastic opaque by Def. 19. Since in H transaction T_j reads x from T_i while T_i is live, then, by Def. 2, T_i releases x early in x. Hence, since H is elastic opaque, elastic opacity supports early release, by Def. 4. □

Lemma 21. *Elastic opacity does not support overwriting.*

Proof. For the sake of contradiction, let us assume that there is an elastic opaque history H, s.t. transaction T_i writes value v to some variable x and releases it early in H. Furthermore, let us assume that there is overwriting, so after some transaction T_j reads v from x, T_i writes u to x. Since only elastic transactions can release early in elastic opaque histories, and T_i releases early, T_i is necessarily elastic. Thus, in any $f_{\mathcal{C}}(H)$ T_i is replaced by a cut $\mathcal{C}_i(H)$.

The two writes on x in T_i are either a) in two different subhistories in $\mathcal{C}_i(H)$, or b) in the same subhistory in $\mathcal{C}_i(H)$. Since the definition of a consistent cut requires all writes on a single variable to be within one subhistory of the cut, then in case (a), $\mathcal{C}_i(H)$ is inconsistent. Since by Def. 19 elastic opaque histories are created using consistent cuts, then H is not elastic opaque, which is a contradiction.

In the case of (b), let us say that both writes are in a subhistory that is converted into transaction $T_{i'}$ in $f_{\mathcal{C}}(H)$. Since T_i releases x early, then by Def. 2, there is a transaction $T_{j'}$ in $f_{\mathcal{C}}(H)$ which executes a read on x reading the value written by $T_{i'}$ in $f_{\mathcal{C}}(H)$. Since we assume overwriting, the read operation on x in $T_{j'}$ reads the value written by the first of the two writes in $T_{i'}$ and does so before the other write on x is performed within $\mathcal{C}_H(i)$. Then, in any sequential history S equivalent to $f_{\mathcal{C}}(H)$ either $T_{j'} \prec_S T_{i'}$ or $T_{i'} \prec_S T_{j'}$. In the former case $T_{j'}$ in S is not legal in S, since the read on x that yields value v will not be preceded by any operation that writes v to x in any possible $\mathit{Vis}(S, T_{j'})$. In the latter case $T_{j'}$ in S is also not legal in S, since there will be a write operation writing u to x between the read on x that yields value v and any operation that writes v to x in $\mathit{Vis}(S, T_{j'})$. Since $T_{j'}$ in S is not legal in any S equivalent to $f_{\mathcal{C}}(H)$, then, by Def. 12, $f_{\mathcal{C}}(H)$ is not final-state opaque, and hence, by Def. 13, not opaque. In effect, by Def. 19, H is not opaque, which is a contradiction.

Thus, there cannot be an elastic opaque history H with overwriting. □

Lemma 22. *Elastic opacity does not support aborting early release.*

Proof. For the sake of contradiction, let us assume that there is an elastic opaque history H s.t. transaction T_i releases some variable x early in H and aborts. Since T_i releases early then it writes v to x, and there is another T_j that executes a read on x that returns v before T_i aborts. Since only elastic transactions can release early in elastic opaque histories, and T_i releases early, T_i is necessarily elastic. If T_i aborts in H, then all of the transactions resulting from its cut $\mathcal{C}_i(H)$ in $f_{\mathcal{C}}(H)$ also abort (by construction of $f_{\mathcal{C}}(H)$). Therefore, for any sequential history S equivalent to $f_{\mathcal{C}}(H)$, there is no subhistory $H' \in \mathcal{C}_i(H)$ s.t. $H' \subseteq \mathit{Vis}(S, T_j)$, and in effect the read operation in T_j on x reading v is not preceded by a write operation writing v to x. Therefore, $\mathit{Vis}(S, T_j)$ is illegal, so T_j in S is not legal in S, and thus, by Def. 13 $f_{\mathcal{C}}(H)$ is not opaque. Since $f_{\mathcal{C}}(H)$ is not opaque, then by Def. 19, H is not elastic opaque, which is a contradiction. □

Property	Application	Def. 4	Def. 5	Def. 6	⊆Serializable
Serializability	database, TM	✓	✓	✓	✓
CO	database	✓	✓	✓	×
Recoverability	database	✓	✓	✓	×
Cascadelessness	database	×	×	×	×
Strictness	database	×	×	×	✓
Rigorousness	database	×	×	×	✓
Opacity	TM	×	×	×	✓
Markability	TM	×	×	×	✓
TMS1	TM	×	×	×	✓
TMS2	TM	×	×	×	✓
VWC	TM	✓	×	×	✓
Live opacity	TM	✓	×	×	✓
Elastic opacity	TM	✓	×	×	×

Table 3.1: Summary of property early release support: Def. 4 is early release support, Def. 5 is overwriting support, and Def. 6 is aborting early release support.

Elastic opacity supports early release, but, since it does not guarantee serializability (as shown in [28]), we consider it to be a relatively weak property. This is contrary to our premise of finding a property that allows early release and provides stronger guarantees than serializability. It also makes elastic opacity unintuitive to programmers, and therefore less than practical.

In addition, elastic transactions, i.e. transactions described by elastic opacity, were proposed as an alternative to traditional transactions for implementing search structures. However, we submit that the restrictions placed on the composition of elastic transactions and the need for transactions with early release to be non-aborting put an unnecessary burden on general-purpose TM. In particular, for a cut to be well-formed, it is necessary that all writes are executed in the same subtransaction, and that no subtransaction starts with a write, which severely limits how early release can be used and precludes scenarios that are nevertheless intuitively correct.

3.3 Summary

In Table 3.1 we present a summary of the properties discussed in this chapter. The table informs whether a particular property is a database property or a TM property, and whether each of the properties satisfies the definitions for early release support, overwriting support, and aborting early release support. Finally, the last column informs whether each property is at least as strong as serializability.

It is worth noting, that only a small fraction of TM safety properties allow early release at all. Those that do allow early release fall into two groups. The first group are weak properties, which allow aborted transactions to view whatever inconsistent state—they do not provide sufficient consistency guarantees and may lead to the dangerous errors stemming from inconsistent views described in [33]. The second group are properties that include the requirement that transactions which release early not abort. This

requirement is too restrictive and excludes many practical (D)TM applications. In addition, particular properties within that group are also not suitable for TM with early release for other reasons. For instance, elastic opacity cannot be employed as a general purpose safety property, since it allows non-serializable histories, which amounts to accepting histories that are not intuitively correct, while putting constraints on TMs that exclude some histories that are intuitively correct. Another example is live opacity, which arbitrarily forbids transactions that read from live transactions to be read from by other transactions, even though such executions do not inherently cause inconsistent views.

4
Existing Algorithms

The purpose of this chapter is to present and examine existing TM concurrency control algorithms with a focus on systems employing pessimistic concurrency control, distributed TM, and systems using the technique of early release. Specifically, in the following consecutive sections we examine examples of distributed pessimistic TM algorithms, distributed optimistic TM algorithms, non-distributed pessimistic TM algorithms, and optimistic TM algorithms with early release. In each group we present a broad overview of the class and proceed to introduce specific representative examples in depth. For each example we attempt to determine the safety properties of each algorithm (indicating sources or uncertainty as applicable), as well as placing them within system models defined in Chapter 2. In addition, in the final section we provide a comparative summary of the characteristics of algorithms which we examined and a brief discussion.

4.1 Distributed Pessimistic TM

Pessimistic distributed TM systems are a fairly unexplored part of the TM system spectrum. Nevertheless, two-phase locking algorithms have successfully been used in distributed database systems, and port well into the TM model. In addition, a family of versioning algorithms designed for use in protocol stacks for communicating services (or processes) is similarly well suited for use within the TM model. We present the details of the former in Section 4.1.1, and of the later in Section 4.1.2.

4.1.1 Two-Phase Locking Algorithms

Two-phase locking (*2PL*) [12, 91] is a lock-based concurrency control algorithm whose variants are used for database transactions in local as well as distributed databases. The algorithms translate to variable-based transactional memory and distributed transactional memory. Given the large number of applicable 2PL variants, we will refer to the entire family of algorithms as the 2PL family. In order to avoid ambiguity, we will refer to the original 2PL algorithm as *Basic 2PL* (*B2PL*).

```
1  proc start(Transaction T_i) {
2    ASet_i ← ∅
3  }
4  proc read(Transaction T_i, Var x) {
5    if owner(lk(x)) ≠ T_i
6      lock lk(x) → R
7    ASet_i ← ASet_i ∪ {x}
8    v ← x
9    if shrinking
10     for y ∈ ASet_i: finished(y)
11       unlock lk(y)
12   return v
13 }
14 proc write(Transaction T_i, Var x, Value v) {
15   if owner(lk(x)) ≠ T_i
16     lock lk(x) → W
17   if owner(lk(x)) = T_i and mode(lk(x)) ≠ W
18     convert lk(x) → W
19   if st_i(x) = ⊥
20     x ← st_i(x)
21   ASet_i ← ASet_i ∪ {x}
22   x ← v
23   if shrinking {
24     for y ∈ ASet_i: finished(y)
25       unlock lk(y)
26     for y ∈ ASet_i: finished_writing(y)
27       convert lk(x) → R
28   }
29   return ok_i
30 }
31 proc commit(Transaction T_i) {
32   for x ∈ ASet_i
33     if owner(lk(x)) = T_i
34       unlock lk(x)
35   return C_i
36 }
37 proc abort(Transaction T_i) {
38   for x ∈ ASet_i
39     if owner(lk(x)) = T_i {
40       if st_i(x) ≠ ⊥
41         x ← st_i(x)
42       unlock lk(x)
43     }
44   return A_i
45 }
```

Figure 4.1: B2PL.

```
1  proc start(Transaction T_i) {}
2  proc read(Transaction T_i, Var x) {
3    if first operation on x
4      acquire_all(T_i)
5    v ← x
6    if shrinking {
7      for y ∈ ASet_i: finished(y)
8        unlock lk(y)
9      if unlocked something
10       notify all
11   }
12   return v
13 }
14 proc write(Transaction T_i, Var x, Value v) {
15   if first operation on x
16     acquire_all(T_i)
17   x ← v
18   if shrinking {
19     for y ∈ ASet_i: finished(y)
20       unlock lk(y)
21     for y ∈ ASet_i: finished_writing(y)
22       convert lk(x) → R
23     if unlocked something
24       notify all
25   }
26   return ok_i
27 }
28 proc commit(Transaction T_i) {
29   for x ∈ ASet_i
30     if owner(lk(x)) = T_i
31       unlock lk(x)
32   if unlocked something
33     notify all
34   return C_i
35 }
36 proc acquire_all(Transaction T_i) {
37   atomic {
38     while ∃x ∈ RSet_i: mode(lk(x)) = R or
39           ∃x ∈ WSet_i: mode(lk(x)) = W
40       wait
41     for x ∈ RSet_i
42       lock lk(x) → R
43     for x ∈ WSet_i:
44       lock lk(x) → W
45   }
46 }
```

Figure 4.2: C2PL.

Basic 2PL

The general idea of B2PL is that for every variable $x \in \mathit{Var}$ there is a read-write lock $\texttt{lk}(x)$. The algorithm uses these locks to prevent two concurrent transactions from accessing a data item in conflicting modes. Each transaction acquires locks in the *growing phase* and releases them in the following *shrinking phase*. Once a transaction enters the shrinking phase (i.e., releases a lock) it cannot acquire further locks.

Fundamentally, in the B2PL algorithm, when transaction T_i invokes an operation on some variable x it must have acquired $\texttt{lk}(x)$ in the appropriate mode before doing so. Specifically:

a) before executing any read operation on x, T_i must acquire $\texttt{lk}(x)$ in either read or write mode (the former if there were no preceding writes), and

b) before executing any write operation on x, T_i must acquire $\texttt{lk}(x)$ in write mode (which means if it previously acquired $\texttt{lk}(x)$ in read mode, it must escalate it to write mode).

We give the pseudocode of B2PL in Fig. 4.1 defined as a TM consistency control algorithm that responds to the invocations of operations on shared variables and transactional operations following the rules above. The way in which locks are acquired and released by B2PL is not strictly specified, so we model the B2PL TM implementation

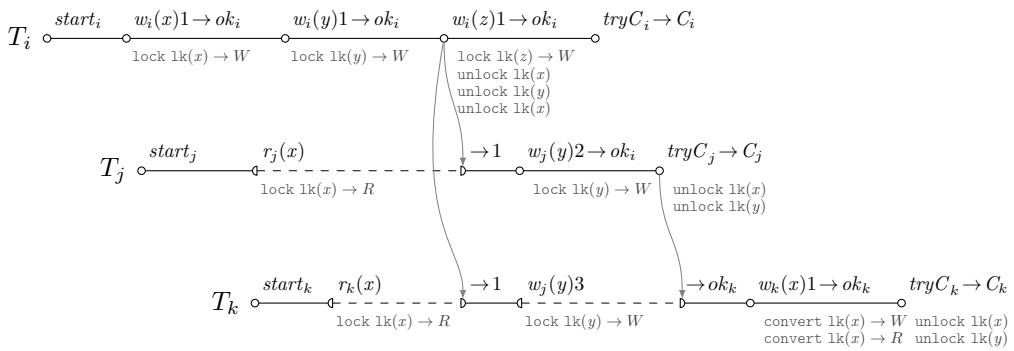


Figure 4.3: An example B2PL history, adapted from [91].

under the assumption that locks should be acquired as late as possible and released as soon as possible. Hence, locks are acquired in the appropriate mode during the first operation of a given type on a given variable. A lock on x can be released once a transaction executed all of its operations on x (indicated `finished(`x`)`) and the transaction is in the shrinking phase. In addition, a transaction can convert from a write lock to a read lock if it will perform no writes on x in the future (indicated `finished_writing(`x`)`).

We show an example of a B2PL history in Fig. 4.3. Here, transaction T_i executes three consecutive writes to variables x, y, and z. It acquires new locks with each write, and finally releases them at the end of the last write (having somehow discovered that no new locks will be acquired and no further operations will be executed). Meanwhile, transactions T_j and T_k start and attempt to read x, which causes both of them to wait until T_i releases $\texttt{lk}(x)$. T_j and T_k are finally able to acquire $\texttt{lk}(x)$ after T_i executes its last write, and since both read from x, they can lock it simultaneously in read mode. Next, both transactions attempt to write to y, but only one of them (T_j) can simultaneously lock $\texttt{lk}(y)$ in write mode. Hence T_k must then wait until T_j releases locks. For whatever reason though, T_j is not able to confirm that no further operations will be executed or locks acquired in its lifetime, so it only releases its locks on commit. Hence T_k may only execute its write once T_j finishes committing. Finally, T_k executes one more write on z. It must first convert the lock to write mode, because it only previously acquired $\texttt{lk}(x)$ in read mode. At this point T_k enters the shrinking phase and also recognizes that this was the last write T_k will execute on x. However, (for some reason) it is not certain whether T_k will execute further read operations on x. Hence, T_i only converts the lock back from write mode to read mode after the write. Finally, T_k eventually commits, at which point it releases all of the locks it retained until this point.

We intentionally leave vague the mechanism in which transactions obtain the knowledge as to when no future operations will follow and when the transaction enters the shrinking phase, since it is not given as part of the specification of B2PL. A simple example mechanism would add operations to the transactional API that the programmer could use to indicate that a given variable is no longer going to be used, or that no new variables are going to be added to the access set. Note, though, that if no such mechanism exists, simply releasing all locks on commit also fits the general specification of B2PL.

As demonstrated in [12], B2PL is (strict) serializable. However, since B2PL allows committing out of order as well as reading from live transactions, it does not satisfy stronger properties.

Correct B2PL executions may result in deadlocks. Take, for instance, the B2PL history in Fig. 4.4. Here transactions T_i and T_j concurrently access variables x and y. T_i successfully acquires $\texttt{lk}(x)$ in read mode and proceeds to execute a read on x. Soon after,

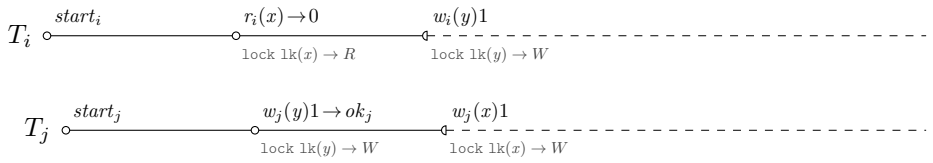


Figure 4.4: An example B2PL history with a deadlock, adapted from [12].

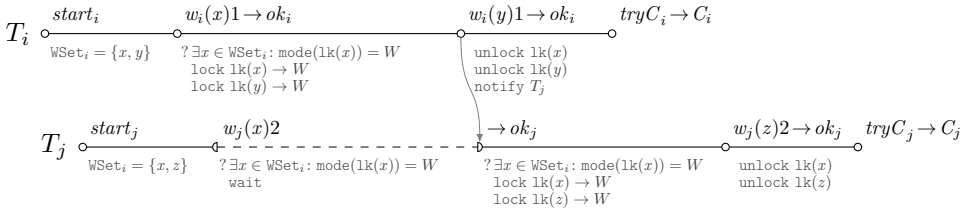


Figure 4.5: An example C2PL history.

T_j acquires $\texttt{lk}(y)$ in write mode and writes to y. Then, as T_i attempts to execute a write operation on y it must acquire $\texttt{lk}(y)$, which means it must wait until T_j releases $\texttt{lk}(y)$. Therefore T_i waits at the invocation of the write on y. Finally, T_j attempts to execute a write on x, which requires it to acquire $\texttt{lk}(x)$, which causes T_j to wait. Since T_i is currently the owner of $\texttt{lk}(x)$, T_j must wait until T_i releases $\texttt{lk}(x)$. In this way, T_i and T_j create a wait cycle and deadlock.

In order to resolve deadlocks, database systems using 2PL employ deadlock detectors (independent of the transactions themselves), which track wait dependencies and detect cycles in the wait dependency graph. Once detected, one of the transactions in the cycle—the *victim*—is forcibly aborted, which causes it to abandon waiting, restore variables it modified to their original values and retry later on. The choice of the victim is open and may be modified according to goals (e.g. to eliminate cyclic restarts) as well as circumstances (to redo the least computations). Other deadlock management techniques exist as well, including executing lock acquisition conservatively, in a way as never to cause a deadlock, and if a deadlock could happen *in potentia*, abort the transaction before locking. Finally, the lock acquisition strategy can be modified to prevent wait cycles in more conservative variants of 2PL.

Conservative 2PL

Conservative 2PL (*C2PL*) (also called Static 2PL) is a variant of 2PL that prevents deadlocks and all transaction aborts by requiring locks to be preclaimed. C2PL assumes the knowledge of the read set and write set *a priori* for each transaction. These sets are then used to acquire all of the locks for all the variables each transaction might access throughout its execution before the transaction invokes the first operation of any kind, on any variable. If all the locks could be acquired, the transaction proceeds to execute its operations. However, if any of the locks could not be acquired, the transaction acquires none of the locks, but instead queues up to wait. Whenever a lock is released, waiting transactions are notified, and they each re-try to acquire all of the locks, following the same procedure and either proceed to execute their operations in case of success, or end up waiting again. We give the pseudocode of an implementation of C2PL in Fig. 4.2.

We give an example of a C2PL history in Fig. 4.5. Here, T_i executes two consecutive writes on x and y. Before executing the first write, it acquires locks for both variables, which is successful, since all locks are initially unlocked. Meanwhile T_j attempts to execute

```
1 proc start(Transaction T_i) {
2   ASet_i ← ∅
3 }
4 proc read(Transaction T_i, Var x) {
5   if owner(lk(x)) ≠ T_i
6     lock lk(x) → R
7   ASet_i ← ASet_i ∪ {x}
8   v ← x
9   if shrinking
10    for y ∈ ASet_i: finished(y)
11      if mode(lk(x)) = R
12        unlock lk(y)
13  return v
14 }
15 proc write(Transaction T_i, Var x, Value v) {
16   if owner(lk(x)) ≠ T_i
17     lock lk(x) → W
18   if owner(lk(x)) = T_i and mode(lk(x)) ≠ W
19     convert lk(x) → W
20   if st_i(x) = ⊥
21     x ← st_i(x)
22   ASet_i ← ASet_i ∪ {x}
23   x ← v
24   return ok_i
25 }
26 proc commit(Transaction T_i) {
27   for x ∈ ASet_i
28     if owner(lk(x)) = T_i
29       unlock lk(x)
30   return C_i
31 }
32 proc abort(Transaction T_i) {
33   for x ∈ ASet_i
34     if owner(lk(x)) = T_i {
35       if st_i(x) ≠ ⊥
36         x ← st_i(x)
37       unlock lk(x)
38     }
39   return A_i
40 }
```

Figure 4.6: S2PL.

```
1 proc start(Transaction T_i) {
2   ASet_i ← ∅
3 }
4 proc read(Transaction T_i, Var x) {
5   if owner(lk(x)) ≠ T_i
6     lock lk(x) → R
7   ASet_i ← ASet_i ∪ {x}
8   v ← x
9   return v
10 }
11 proc write(Transaction T_i, Var x, Value v) {
12   if owner(lk(x)) ≠ T_i
13     lock lk(x) → W
14   if owner(lk(x)) = T_i and mode(lk(x)) ≠ W
15     convert lk(x) → W
16   if st_i(x) = ⊥
17     x ← st_i(x)
18   ASet_i ← ASet_i ∪ {x}
19   x ← v
20   return ok_i
21 }
22 proc commit(Transaction T_i) {
23   for x ∈ ASet_i
24     unlock lk(x)
25   return C_i
26 }
27 proc abort(Transaction T_i) {
28   for x ∈ ASet_i {
29     if st_i(x) ≠ ⊥
30       x ← st_i(x)
31     unlock lk(x)
32   }
33   return A_i
34 }
```

Figure 4.7: R2PL.

a write on x. Hence, it also attempts to acquire locks for all the variables in its write set. This fails, since T_i holds the lock for x, so T_j begins waiting. Then, T_i executes its last write, releases its locks on x and y, and notifies all waiting transactions, i.e. T_j. Then, T_j can attempt to acquire all its locks again. Since this is successful, T_j proceeds to execute its operations.

Since waiting transaction never holds any locks, they do not participate in a wait dependency cycle, which means C2PL transactions never deadlock (see [12]). Since deadlocks are the only cause of aborts in B2PL, C2PL also completely eliminates aborts.

Since C2PL admits a subset of histories admitted by B2PL, C2PL is also (strict) serializable. However, C2PL does not satisfy any stronger properties, since it is possible for transactions to read from live transactions and commit out of order.

Even though C2PL does not involve forced aborts, it can easily be extended to the arbitrary abort model, by providing an abort operation and keeping "clean" backup copies of variables just like in B2PL.

Strict 2PL

Strict 2PL (*S2PL*) is a variant of 2PL that abandons early release in favor of stronger properties. Specifically, S2PL behaves like B2PL, but write locks are never released until the transaction commits or aborts. We give the pseudocode in Fig. 4.6 (compare with Fig. 4.1). As a result no transaction ever reads from another live transaction. Hence, as demonstrated in [91], S2PL satisfies strictness.

We show an example of an S2PL history in Fig. 4.8. Here, transaction T_i executes

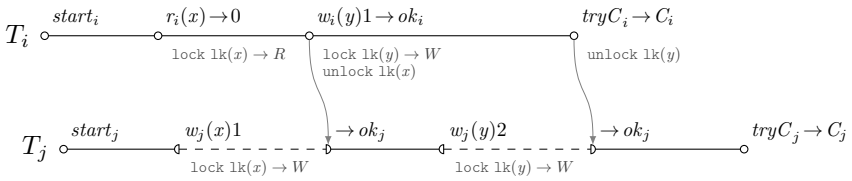


Figure 4.8: An example S2PL history.

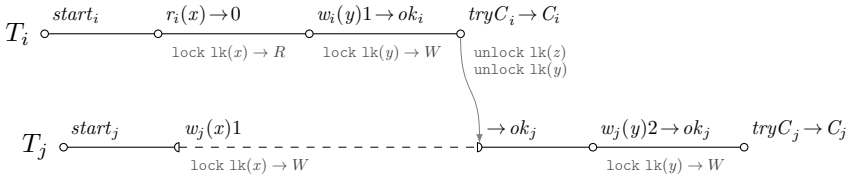


Figure 4.9: An example R2PL history.

a read on x, followed by a write on y. The transaction acquires the appropriate locks before each operation execution. After executing the write operation T_i determines it will not acquire further locks, so it can start releasing locks. The only lock T_i can release immediately is $\texttt{lk}(x)$ which it acquired in read mode. T_i can only release the remaining lock $\texttt{lk}(y)$ on commit, since it was acquired in write mode. In effect, transaction T_j must wait until T_i commits before accessing y, although it may access x much sooner.

It is possible for S2PL transactions to deadlock. In order to prevent deadlocks in strict executions, the S2PL can use the same approach to acquiring locks as C2PL, in effect creating a variant of 2PL we will refer to as *conservative strict 2PL* (*CS2PL*). CS2PL is strict like S2PL, does not deadlock, and, in effect, never aborts, like C2PL. By analogy to CA2PL, we can create *CAS2PL*, a variant of CS2PL that allows arbitrary aborts.

Rigorous 2PL

Rigorous 2PL (*R2PL*, also known as *strong strict 2PL*) is another variant of 2PL which releases all locks at commit time (or during abort) in order to satisfy the property rigorousness. We give the pseudocode in Fig. 4.7 (compare with Fig. 4.6).

Since locks are released only at the end of a transaction, if some transaction T_i executes any operation op_i on x after some other transaction T_j executes its own operation op_j on x, then T_i commits or aborts between op_i and op_j. Hence, trivially, R2PL satisfies rigorousness. However, just as with S2PL, deadlocks are not prevented in R2PL.

We show an example of an R2PL history in Fig. 4.9. The history is analogous to that in Fig. 4.8, with the exception that T_i does not release any locks before it commits. Hence T_j must delay all of its operations until after T_i is already finished.

As with S2PL, R2PL allows deadlocks, but can be extended to prevent them by preclaiming locks. Hence, *conservative rigorous 2PL* (*CR2PL*) and *conservative aborting rigorous 2PL* (*CAR2PL*) variants of R2PL can trivially be derived.

Object-based 2PL

Even though 2PL algorithms are designed specifically for variables, their pessimistic approach and in-place modifications make them suitable for the homogeneous object model as they are. In addition, if object interfaces are known, there is a proposition in [12] of extending the lock system to a more complex one, where each operation in an

object's interface uses a different lock, and a table of lock compatibility is defined. This would allow the TM to be applied in an extended homogeneous object model with more operation types and more complex semantics.

Adapting the 2PL algorithms to the heterogeneous object model requires that all operations are treated as writes and all locks are treated as exclusion locks.

Distributed 2PL

2PL algorithms can be easily used in distributed contexts using the control flow model of transaction execution. No modifications to the algorithm are required for that, although various auxiliary modules, like deadlock detection may require a comprehensive overhaul with respect to their non-distributed counterparts (see e.g. [12]). In that case conservative variants of 2PL are best suited for distributed environments.

4.1.2 Versioning Algorithms

Versioning algorithms [96, 97] are a family of pessimistic distributed transactional concurrency control algorithms. The algorithms were initially designed with communicating processes or network services in mind [100], but apply to transactional memory operating in the heterogeneous object model.

In general, versioning uses two version counters to determine whether a given transaction can be allowed to access a particular shared object, or whether the access should be deferred to avoid conflicts. The intuition behind how these counters work is by analogy to how the teller may manage a queue in a bank: customers who come into the bank retrieve a ticket with a number from a dispenser and wait before approaching the teller until their number is called. Meanwhile the teller increments the number as she finishes serving each consecutive customer. In the analogy, each customer is a transaction, and the teller is an object. The number in the customer's hand is his version for that object, and it is being compared against the number that is currently being served by the teller—the object's version.

Basic Versioning Algorithm

Basic Versioning Algorithm (*BVA*) [100, 96] is a straightforward implementation of versioning concurrency control. This mechanism is key in our further discussion, so we explain it in detail below. We also provide the full pseudocode of BVA in Fig. 4.10.

Whenever a transaction T_i starts, it retrieves a *private version* $\mathrm{pv}_i(\lceil x \rfloor)$ for every object $\lceil x \rfloor$ in its access set $\mathtt{ASet}_i$. The access set is assumed to be known *a priori*. The values of private versions received by consecutive transactions are assigned from a sequence of consecutive positive integers. The sequence is generated using a *global version* counter $\mathrm{gv}(\lceil x \rfloor)$, which is initially 0 and is incremented with each starting transaction that has $\lceil x \rfloor$ in its access set. The assignment is also guarded by a global lock so that it is done atomically. In effect, transactions' private versions have the following properties:

I no two transactions have the same private version for any shared object,

II if transaction T_i started before T_j and they both access $\lceil x \rfloor$, then $\mathrm{pv}_i(\lceil x \rfloor) < \mathrm{pv}_j(\lceil x \rfloor)$,

III given two transactions T_i and T_j, if $\mathrm{pv}_i(\lceil x \rfloor) < \mathrm{pv}_j(\lceil x \rfloor)$ then for any shared object $\lceil y \rfloor$ that both transactions plan to access, $\mathrm{pv}_i(\lceil y \rfloor) < \mathrm{pv}_j(\lceil y \rfloor)$, and

```
1 proc start(Transaction T_i) {
2   lock lk^g → W
3   for ⌈x⌋ ∈ ASet_i {
4     gv(⌈x⌋) ← gv(⌈x⌋) + 1
5     pv_i(⌈x⌋) ← gv(⌈x⌋)
6   }
7   unlock lk^g
8 }
9 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
10  wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
11  execute m on ⌈x⌋ returning v
12  return v
13 }
14 proc commit(Transaction T_i) {
15  for ⌈x⌋ ∈ ASet_i {
16    wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
17    :dismiss(T_i, ⌈x⌋)
18  }
19  return C_i
20 }
21 proc :dismiss(Transaction T_i, Object ⌈x⌋) {
22  lv(⌈x⌋) ← pv_i(⌈x⌋)
23 }
```

Figure 4.10: BVA.

```
1 proc start(Transaction T_i) {
2   lock lk^g → W
3   for ⌈x⌋ ∈ ASet_i {
4     gv(⌈x⌋) ← gv(⌈x⌋) + 1
5     pv_i(⌈x⌋) ← gv(⌈x⌋)
6   }
7   unlock lk^g
8 }
9 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
10  wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
11  execute m on ⌈x⌋ returning v
12  ac_i(⌈x⌋) ← ac_i(⌈x⌋) + 1
13  if ac_i(⌈x⌋) = supr_i(⌈x⌋)
14    :release(T_i, ⌈x⌋)
15  return v
16 }
17 proc commit(Transaction T_i) {
18  for ⌈x⌋ ∈ ASet_i {
19    wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
20    :dismiss(T_i, ⌈x⌋)
21    ltv(⌈x⌋) ← pv_i(⌈x⌋)
22  }
23  return C_i
24 }
25 proc :release(Transaction T_i, Object ⌈x⌋) {
26  lv(⌈x⌋) ← pv_i(⌈x⌋)
27 }
28 proc :dismiss(Transaction T_i, Object ⌈x⌋) {
29  if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
30    lv(⌈x⌋) ← pv_i(⌈x⌋)
31 }
```

Figure 4.11: SVA.

IV if T_i started before T_j and no other transaction started in between the two, and both plan to access $\lceil x \rfloor$, then they have consecutive private versions for $\lceil x \rfloor$, i.e. $\mathtt{pv}_i(\lceil x \rfloor) = \mathtt{pv}_j(\lceil x \rfloor) - 1$.

BVA uses private versions to maintain order when accessing shared objects via objects' *local versions*. That is, each shared object $\lceil x \rfloor$ has its own *local version* counter, denoted $\mathtt{lv}(\lceil x \rfloor)$, which is always equal to the private version of such transaction T_j that most recently finished using the object, i.e. out of all transactions that have $\lceil x \rfloor$ in their access set, T_j is the one that committed most recently. Specifically, when T_j does commit, it writes its own private version for $\lceil x \rfloor$ to the local version counter of $\lceil x \rfloor$. This is enclosed in the dismiss procedure. In the context of versioning algorithms, whenever any transaction writes its private version to a local version counter for some object, we say the transaction *released* the object.

Once T_j releases $\lceil x \rfloor$, T_j will execute no further operations on $\lceil x \rfloor$, so some other transaction can safely start calling methods on the object. BVA determines which transaction gets to access the object next by simply selecting the transaction with the next consecutive private version, i.e. such T_i whose $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{pv}_j(\lceil x \rfloor)$. Thus, invariably $\lceil x \rfloor$ can be accessed by such T_i for which $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$, and no other transaction. Hence, if some transaction T_i wants to access $\lceil x \rfloor$, then it may do so if $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$. We call this condition the *access condition*. On the other hand, if T_i wants to access $\lceil x \rfloor$ and the access condition is not satisfied, then it waits until it is satisfied. In this way, only one transaction is able to access $\lceil x \rfloor$ at any one time. Initially, all local version counters are set to 0, so a transaction with a private version of 1 can access a given variable as the first.

An example of how this mechanism works is shown in Fig. 4.12. Here, T_i and T_j attempt to access shared object $\lceil x \rfloor$ at the same time. Transaction T_i starts first, so $\mathtt{pv}_i(\lceil x \rfloor) = 1$, and T_j starts second, so $\mathtt{pv}_j(\lceil x \rfloor) = 2$. Since initially $\mathtt{lv}(\lceil x \rfloor) = 0$, T_j is not able to pass the access condition and execute an operation on $\lceil x \rfloor$ when it tries to, so it waits. On the other hand, T_i can pass the access condition $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$ and

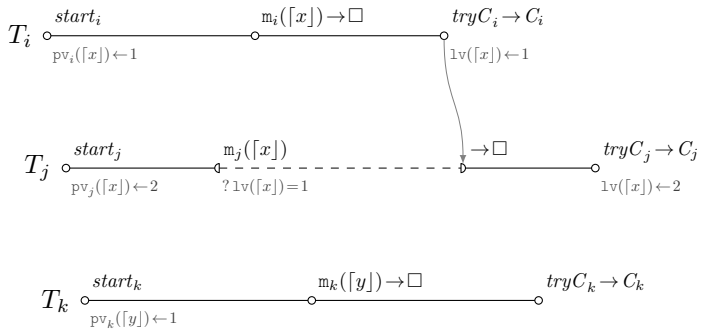


Figure 4.12: An example BVA history.

it executes an operation on $\lceil x \rfloor$ without waiting. Once T_i commits, it sets $\mathtt{lv}(\lceil x \rfloor)$ to 1, so T_j then becomes capable of passing the access condition and finishing executing its operation on $\lceil x \rfloor$. In the mean time, transaction T_k can proceed to access $\lceil y \rfloor$ completely in parallel.

Given that transactions access objects in an order defined by private versions, and since private versions are assigned to transactions both atomically and from a monotonic sequence of integers, BVA avoids deadlocks. In addition, BVA trivially never aborts. In [96, 97] the author shows that BVA is serializable (as *isolation*). In fact, BVA preserves real-time order and executes all potentially conflicting transaction sequentially, so it is straightforward to see that it is rigorous, as well as opaque.

While BVA is a strong algorithm, serializing conflicting transactions makes for a low degree of parallelism in produced histories. Hence, the author of [96, 97] introduces two variants of BVA that execute conflicting transactions partially in parallel: the *Supremum Versioning Algorithm* (*SVA*) and the *Routing Versioning Algorithm* (*RVA*). Out of these, RVA uses a different system model where operations on shared objects are executed completely asynchronously (without reading the result of the operation) and may complete even after the transaction commits. These assumptions do not fit the TM system model, so RVA is not directly applicable. On the other hand, eliminating asynchrony trivially reduces RVA to BVA. Hence, in further discussion we concentrate on SVA alone.

Supremum Versioning Algorithm

Supremum Versioning Algorithm (*SVA*) [100, 97] is a variant of BVA that uses an early release mechanism to execute concurrent conflicting transactions partially in parallel. That is, SVA transactions use the versioning concurrency control mechanism from BVA, but transactions sometimes are able to release shared objects before the transaction commits. After executing an operation on some shared object, a transaction uses additional *a priori* knowledge to decide whether it will perform any further actions on that object. If it can be safely determined that no further operation will occur, the transaction releases the object instantly. We explain the mechanism below and give the pseudocode in Fig. 4.11.

Each SVA transaction has *a priori* knowledge of *suprema*: it knows at most how many times it will attempt to access each object throughout its execution. We denote transaction T_i's supremum for object $\lceil x \rfloor$ as $\mathtt{supr}_i(\lceil x \rfloor)$ and it takes the value of either a positive integer (if supremum were 0, the object would not be in the access set of T_i), or infinity ω. If the supremum is unknown, setting it to ω guarantees correct execution (since only an upper bound is required). However, the supremum must never be lower than the actual number of accesses. Suprema can be derived and specified for each transaction

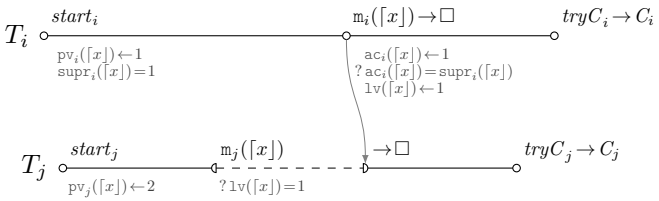


Figure 4.13: An example SVA history (early release).

manually by the programmer, but it can also be derived and automatically supplemented by a type checker [96]. In addition, as part of our research we propose a static analyzer and precompiler that can generate this information (see Chapter 9).

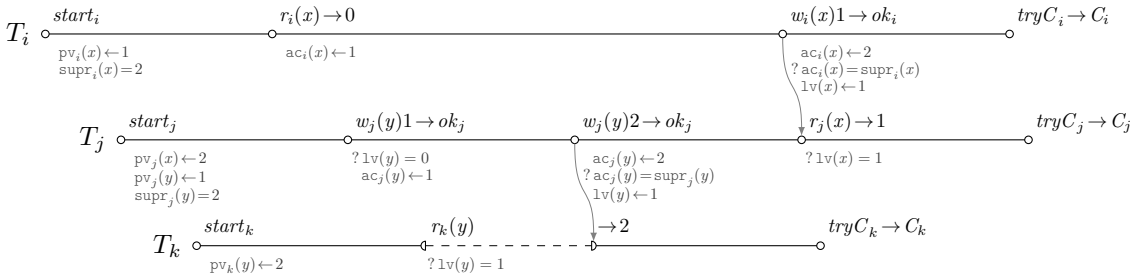
Given specified suprema, each SVA transaction T_i counts operation executions on each variable as they occur using an *access counter* $\mathtt{ac}_i(\lceil x \rfloor)$. When the access counter reaches the supremum for $\lceil x \rfloor$, $\mathtt{ac}_i(\lceil x \rfloor) = \mathtt{supr}_i(\lceil x \rfloor)$, the transaction knows that no further accesses on $\lceil x \rfloor$ will occur afterward as part of T_i. Therefore, the transaction releases the object immediately (by writing its own private version for $\lceil x \rfloor$ to the local version counter of $\lceil x \rfloor$). This allows, another transaction T_j, such that $\mathtt{pv}_j(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor) = \mathtt{pv}_i(\lceil x \rfloor)$ to execute operations on $\lceil x \rfloor$ right away, without waiting for T_i to commit. On the other hand, if a transaction does not reach the supremum for some object $\lceil x \rfloor$, the object will be released during commit, as in BVA.

The early release mechanism is illustrated further in Fig. 4.13. Here, transactions T_i and T_j both try to access $\lceil x \rfloor$. Like in Fig. 4.12, since T_i's private version for $\lceil x \rfloor$ is lower than T_j's, the former manages to access $\lceil x \rfloor$ first, and T_j waits until $\lceil x \rfloor$ is released. T_i has upper bound information: it knows that it will execute at most one operation on $\lceil x \rfloor$ ($\mathtt{supr}_i(\lceil x \rfloor) = 1$). The number of operations executed on $\lceil x \rfloor$ is tracked using counter $\mathtt{ac}_i(\lceil x \rfloor)$. After T_i executes its operation on $\lceil x \rfloor$, it increments $\mathtt{ac}_i(\lceil x \rfloor)$. Since this causes $\mathtt{ac}_i(\lceil x \rfloor)$ to reach the supremum $\mathtt{supr}_i(\lceil x \rfloor)$, i.e. $\mathtt{ac}_i(\lceil x \rfloor) = \mathtt{supr}_i(\lceil x \rfloor) = 1$, T_i releases $\lceil x \rfloor$ immediately after it accesses $\lceil x \rfloor$, rather than waiting to do so until commit. In effect, T_j can access $\lceil x \rfloor$ earlier. T_j is even capable of committing before T_i.

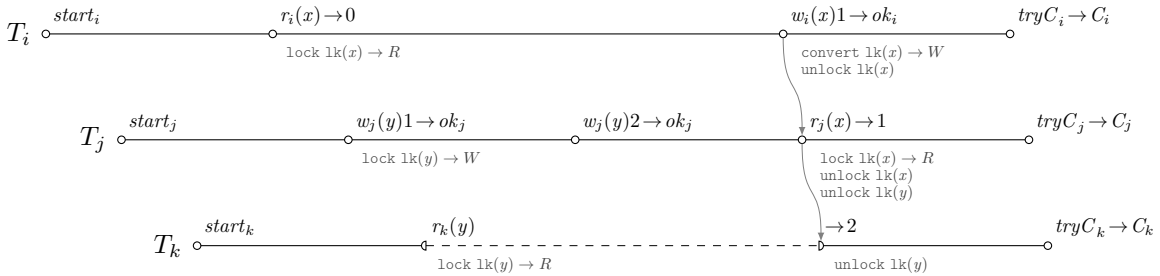
Given that SVA acquires versions during the start of the transaction, and since these versions are used later on to defer operations, SVA is sometimes mistaken for an implementation of a 2PL algorithm with early release: B2PL and C2PL. We contrast the three algorithms in Fig. 4.14. We use the variable model in this comparison for all algorithms (although SVA is still agnostic with respect to operation semantics). The histories shown there contain three transactions each (executing the same transactional code). Transaction T_i starts first and attempt to first read x and then write to x. Transaction T_j starts second and attempts to first update the value of y twice, and then read the value of x. Finally, transaction T_j starts last and attempts to read the value of y. The history in Fig. 4.14a is an SVA execution of those three transactions, the history in Fig. 4.14b is a B2PL execution, and Fig. 4.14c represents a C2PL execution.

The SVA execution is the most parallel of the three, since each variable is "acquired" and released individually, so T_j can first release y and still attempt to gain access to x afterward. On the other hand, B2PL does not allow T_j to release any locks until it entered the shrinking phase. Hence in B2PL T_j cannot release y until the transaction acquires the lock for x. Thus, the time at which T_j releases y is delayed, which causes T_k to execute longer than it does under SVA.

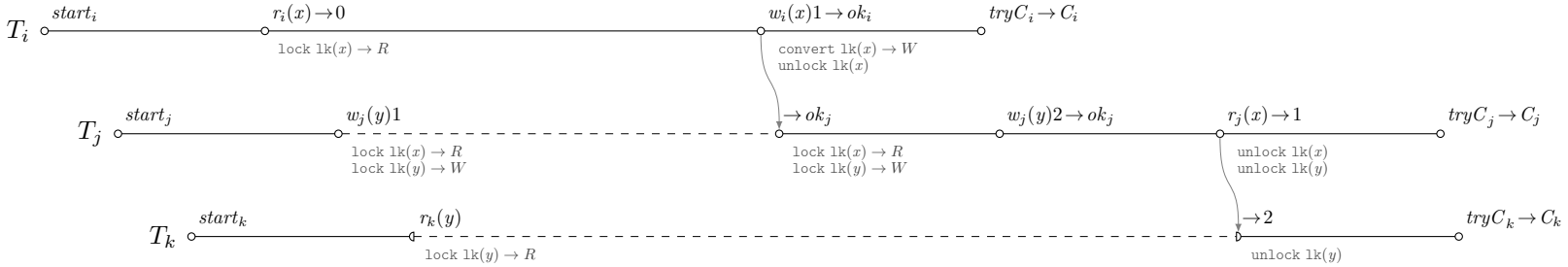
Whereas, C2PL attempts to acquire all locks for its read and write set before the first operation on any shared variable, which causes T_j to wait until T_i releases x before executing operations on y. This introduces additional delays in comparison to SVA. SVA prevents those delays since the access condition to each variable is checked independently



(a) SVA.



(b) B2PL.



(c) C2PL.

Figure 4.14: A comparison between SVA, B2PL, and C2PL histories.

of other variables. The only point at which SVA causes transactions to wait is during the start operation itself.

In contrast to BVA, since two transactions can commit in reverse order to the order in which they accessed some shared object $\lceil x \rfloor$, and since SVA's early release permits a situation where a transaction will read from another live transaction, then SVA cannot satisfy opacity, rigorousness, recoverability, or strictness. Since an SVA transaction can read from a live transaction that reads from another live transaction, SVA is not live opaque. Nevertheless, SVA guarantees strict serializability, as shown in [97]. It also never deadlocks or aborts, just as BVA. We discuss the safety properties of SVA further in Section 7.1.

4.2 Distributed Optimistic TM

Given that distributed systems often face similar concurrency control problems as (non-distributed) multiprocessor, several distributed TM systems were proposed. Broadly, two models of distributed TM are used. Transactional memory can be replicated on several network nodes and locally-scoped transactions can be used by processes to consistently and reliably propagate any changes of shared data to all the replicas (see e.g., [14, 18, 49, 99]). In the other model distributed transactions can be employed to atomically access a subset selected from a larger set of objects. Each object is accessible from a single location only (objects can be replicated, if necessary, but this is an orthogonal issue). This type of TM includes HyFlow2 [86], a state-of-the-art distributed TM system implemented in Scala on top of the Akka library. HyFlow2 implements the optimistic Transactional Forwarding Algorithm (TFA) [67, 69] and operates in the data flow model. HyFlow [68] is an earlier version of HyFlow2, implemented in Java on top of Aleph and DeuceSTM and included control flow and data flow concurrency control algorithms, including TFA [67] and DTL2 (a distributed version of TL2 [21]). HyFlow was compared with HyFlow2 in [86] and was shown to perform worse then its successor. We examine DTL2 and TFA more closely in Sections 4.2.1 and 4.2.2, respectively.

In addition to TM systems, it is worth mentioning distributed transactional database and data store systems. Distributed transactions are successfully used where requirements for strong consistency meet wide-area distribution, e.g., in Google's Percolator [61] and Spanner [17]. Percolator supports multi-row, ACID-compliant, pessimistic database transactions that guarantee snapshot isolation. This is a much weaker guarantee than expected from TM systems. Another drawback in comparison to DTM is that writes must follow reads. Spanner provides semi-relational replicated tables with general purpose distributed transactions. It uses real-time clocks and Paxos to guarantee consistent reads. Spanner requires some *a priori* information about access sets and defers commitment, but aborts on conflict. Irrevocable operations are banned in Spanner. Spanner transactions provide snapshot isolation and external consistency (akin to real-time order), much weaker properties than considered sufficient in DTM.

4.2.1 Distributed Transactional Locking II

Distributed Transactional Locking II (*DTL2*) is an optimistic TM concurrency control algorithm implemented within HyFlow [68]. It is a distributed version of *Transactional Locking II* (*TL2*) [21], a quintessential optimistic (non-distributed) TM concurrency control algorithm that operates in the variable system model and uses locks for synchronization and a global version clock to determine whether operations are valid.

```
1  proc start(Transaction T_i) {
2    RSet_i ← ∅
3    WSet_i ← ∅
4    rv_i ← gv
5  }
6  proc read(Transaction T_i, Var x) {
7    RSet_i ← RSet_i ∪ {x}
8    if mode(lk(x)) = W or version(x)) > rv_i
9      return abort(T_i)
10   if x ∈ WSet_i: // bloom filter
11     return buf_i(x)
12   else
13     return x
14 }
15 proc write(Transaction T_i, Var x, Value v) {
16   WSet_i ← WSet_i ∪ {x}
17   buf_i(x) ← v
18   return ok_i
19 }
20 proc commit(Transaction T_i) {
21   // Lock the write set.
22   for x ∈ WSet_i in order
23     lock lk(x) → W
24   if ∃x ∈ WSet_i: owner(lk(x)) ≠ T_i
25     return abort(T_i)
26
27   // Increment and fetch global version clock.
28   lock lk^g → W
29   gv ← gv + 1
30   wv_i ← gv
31   unlock lk^g
32
33   // Validate the read set.
34   if rw + 1 ≠ wv_i
35     if ∃x ∈ RSet_i:
36         mode(lk(x)) = W or version(x) > wv_i
37       return abort(T_i)
38
39   // Commit and release.
40   for x ∈ WSet_i in order {
41     x ← buf_i(x)
42     version(x) ← wv_i
43     unlock lk(x)
44   }
45   return C_i
46 }
47 proc abort(Transaction T_i) {
48   for x ∈ WSet_i
49     if owner(lk(x)) = T_i
50       unlock lk(x)
51   return A_i
52 }
```

(a) R/W Transactions.

```
53 proc start(Transaction T_i) {
54   RSet_i ← ∅
55   WSet_i ← ∅
56   rv_i ← gv
57 }
58 proc read(Transaction T_i, Var x) {
59   if mode(lk(x)) = W or version(x)) > rv_i
60     return abort(T_i)
61   return x
62 }
63 proc commit(Transaction T_i) {
64   return C_i
65 }
66 proc abort(Transaction T_i) {
67   return A_i
68 }
```

(b) R Transactions.

Figure 4.15: (Distributed) TL2.

We show the full pseudocode of DTL2 in Fig. 4.15 and explain it in detail below. DTL2 allows transactions to run in either a read/write (R/W) mode or a read-only (R) mode. This can be determined *a priori* or all transactions can run as read-only and switch to R/W mode if they attempt to execute a write. The pseudocode in Fig. 4.15a defines the behavior of transactions in R/W mode, and Fig. 4.15b describes the behavior of R transactions. Since the algorithm operates in the CF model, we feel that the details where transactions and variables are located, and how data is moved between nodes can be omitted in the description of the algorithm without confusion.

Each shared variable x in the system has an associated *version* counter *version*(x) and an exclusive lock $\texttt{lk}(x)$. In addition, there is a single *global version clock* gv. The global clock is accompanied by a global lock $\texttt{lk}^g$, that has to be acquired to modify gv.

In both R/W and R mode, as transaction T_i starts, it makes a transaction-local copy of gv and stores it as its *read version* $\texttt{rv}_i$. The read version is later compared against variable versions in order to determine whether the transaction's view is consistent. I.e., when a read-only transaction T_i attempts to read x, it updates its read set and then checks whether *version*(x) $> \texttt{rv}_i$. If that is the case, x was modified since T_i started, so T_i cannot continue without viewing an inconsistent state of the system. Hence T_i aborts in such a case. Similarly, T_i aborts if some other transaction locked $\texttt{lk}(x)$, since that means that some other transaction will commit changes to x and T_i will eventually have to abort anyway.

When a R/W transaction T_i attempts to read x, the procedure is similar, but it may be the case that T_i already wrote to x. DTL2 transactions writes update a transaction-local buffer $\texttt{buf}_i(x)$ (for some T_i, x) and only apply the changes to the actual variable during commit. In the original TL2 specification this buffer is implemented as a redo log instead. Hence, if T_i reads x after previously writing to it, it is sufficient to simply retrieve the value from a buffer. Hence, transactions read their own writes.

Once a read-only transaction reaches commit, it is already consistent, and it has no buffered writes to transfer to variables, so it simply finishes execution. On the other hand, any R/W transaction T_i must apply the changes in its buffers to variables in a consistent manner during commit. Thus, first T_i locks its entire write sets. This is done using some prescribed order to prevent deadlocks. If any lock cannot be instantly obtained, T_i aborts.

Otherwise, the transaction increments the global version clock, to indicate that the state of the system is changed. The incremented value is stored as the transaction's *write version* wv_i. Next, T_i validates its read set. I.e., if any other transaction modified any of the variables in T_i's read set, then T_i is inconsistent and must abort. Hence, the transaction checks whether *version*(x) $> \texttt{rv}_i$ for each x in its read set and acts accordingly. Similarly, it also checks whether such variable's locks are locked, which would indicate that some other transaction is in the process of committing, so T_i also aborts. If $\texttt{rv}_i + 1 = wv_i$, there are no concurrent transactions that could interfere with T_i's read set, in which case read-set validation may be skipped. Once the read-set is deemed valid, T_i proceeds to apply changes from its buffers to the variables in its write set. Once the changes were applied to some variable x, its version is updated to wv_i, and its lock is released. Once this is done for all variables in the write set, the transaction is complete.

We show an example of three concurrent transactions executing under DTL2 in Fig. 4.16. All three transactions read 0 from the global clock as they start, so they all have a read version of 0. Then, transaction T_i executes a read operation on x, which validates correctly and returns 0, the current value of x. Then, transaction T_j executes a write on x. It adds x to its write set and puts the written value 1 into the local buffer $\texttt{buf}_i(x)$. Since this value is stored locally, the next read operation in T_j simply retrieves it from the buffer, without having to retrieve it from the original location. Meanwhile T_i performs its own write on x, writing 2 to its own local buffer—since this only has

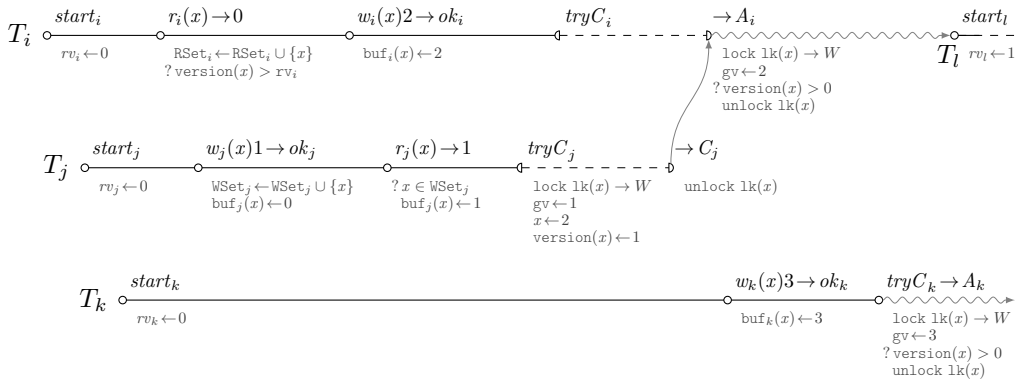


Figure 4.16: An example DTL2 history.

transaction-local effects, the operation does not interfere with concurrent transactions.

Eventually transaction T_j attempts to commit. It locks its write set by locking $\mathtt{lk}(x)$ and increments the global clock to 1. This means that the transaction's write version equals 1. Since the read version and write version are consecutive values, ti means no other transaction attempted to commit since T_j started, so no validation is required. Hence, T_j updates x to 2, and increases its version to its write version: 1. Transaction T_j commit successfully. Meanwhile transaction T_i also attempts to commit, but the commit procedure must wait until the locks are released, so T_i waits until T_k commits. Then, T_i checks the version of all the variables in its read set, i.e. for x. Since T_j set the version of x to 1 during its own commit, and since that is larger than T_i's read version of 0, T_i is forced to abort at this point and restarts later on as T_l (we only show T_l's start operation).

Once T_i finished committing, T_k executes a write. Since that write is performed on the buffer, the transaction continues execution despite T_j committing and invalidating T_k view of the system. However, once T_k attempts to commit a similar situation occurs as with T_i—T_k's read version 0 is lower than the version of x, so T_k must also abort and retry later on (not shown).

In [52] the authors demonstrate that TL2 is opaque, so, by extension, DTL2 is also opaque. We conjecture that DTL2 is rigorous. DTL2 also never deadlocks, but it aborts on conflict, which may cause issues with irrevocable operations.

In its basic form, DTL2 is designed for relatively low contention systems, where conflicts occur only sporadically. Otherwise, it is prone to suffer from high transaction abort rates. Hence modules are typically added to DTL2 to control the execution of transactions. Specifically, a contention manager can be employed that postpones re-execution of transactions that aborted due to conflicts.

4.2.2 Transaction Forwarding Algorithm

Transaction Forwarding Algorithm (*TFA*) [67, 69] is an optimistic distributed TM concurrency control algorithm. TFA was specifically designed for the data flow model, where each shared object is migrated to immobile transactions in order for the transaction to execute operations on the object. In addition, the model specifies that only one copy of each object exists in the system, as opposed to objects being replicated (although caching through buffers is used). TFA can be applied to homogeneous objects and variables—we use the latter model in our discussion. The basic *modus operandi* of TFA resembles that of DTL2, however TFA specifically defines how shared objects are migrated within the

system and uses a mechanism akin to Lamport timestamps [55] in place of DTL2's global version clock. This change introduces a lot of additional complexity into the algorithm, but in return TFA transactions have a lower chance of aborting during presumed conflicts that would not cause inconsistency. For the same reason, TFA also does not have a single point of failure. We describe the algorithm in detail below and provide an adapted pseudocode in Fig. 4.17.

When a TFA transaction starts, it copies the value of the local clock $lc_{\mathcal{L}}$ for node $\mathcal{L}$, the node the transaction starts on, to a transaction-local variable wv_i. This is the transaction's *write version* used to determine whether there were modifications to variables observed by the transactions since it started execution.

When transaction T_i reads a variable x, it first checks whether that variable it wrote to x already. If that is the case, then, just like in DTL2, the transaction uses the copy of the variable made during the write to perform the read. Otherwise, the transaction first loads the variable into the buffer using the procedure `:open`, and then uses that to perform the read. The behavior is similar in the case of writes as well. In order to write to a variable, the transaction first buffers the variable using `:open`, and then performs the write on the buffer.

Apart from loading a variable to the buffer, the `:open` procedure performs all the necessary early validation required to both read and write a variable. These are analogous to DTL2, but use distributed clocks rather than a single global clock. Specifically, given variable x located on node $\mathcal{R}$, before x is buffered, the transaction moves its local clock $lc_{\mathcal{L}}$ forward to match $lc_{\mathcal{R}}$. Then, after x is buffered, if x is a local variable, then the transaction simply checks whether any other transaction updated the variable since the transaction started. If such an interruption occurred, then x has a higher version then the transaction's write version. This forces T_i to abort. Otherwise, the operation can proceed. If the variable is not local to the transaction, then the transaction catches up its local clock $lc_{\mathcal{L}}$ with the variable's $lc_{\mathcal{R}}$. Then, the transaction attempts to catch its write version to match $lc_{\mathcal{R}}$, if necessary. However, before doing so, it checks if any of the variables in its read set were invalidated, i.e. if any of their versions are ahead of their respective local clocks. If this is true for any variable in the read set, then the transaction aborts. If all the steps were successfully performed, the transaction can perform operations on the buffered copy of x.

The commit operation is analogous to DTL2. Initially, the objects in the write set are locked according to some global order. The commit fails, if this cannot be done. When all the locks are acquired, the read set is validated by checking each variable's version against the transaction's write version and making sure that the locks are unlocked. If validation succeeds, then no other transaction wrote to any of the variables in the read set, or is currently committing on them. In the next step, the transaction's local clock is incremented. Then, all the variables in the write set are moved from their original location onto the transaction's node and updated using buffered values. In this way, the "master" copy of each object is located wherever the most recent committed transaction that updated it resides. Once the variable is updated, the transaction's version is updated to the transaction's write version, to reflect that a new fresh value of the variable is present. Finally, each variable's lock is released and the commit is successfully completed.

We show an example execution of TFA in Fig. 4.18. This example is analogous to the DTL2 execution in Fig. 4.16, with a few exceptions. TFA checks version consistency before each write, irrespective of the fact that it is performed locally. This allows TFA to find potential conflicts earlier, and abort a transaction sooner, which wastes less work. More importantly, in the TFA execution T_k is not forced to abort, as opposed to the DTL2 example. That is because using the system of local clocks instead of a global clock allows transactions to catch up to the existing state of the system. Thus, even if

```
1  proc start(Transaction T_i) {
2    RSet_i ← ∅
3    WSet_i ← ∅
4    L ← location(T_i)
5    wv_i ← lc_L
6  }
7  proc read(Transaction T_i, Var x) {
8    RSet_i ← RSet_i ∪ {x}
9    if x ∈ WSet_i
10     return buf_i(x)
11   if :open(T_i, x) = false
12     return abort(T_i)
13   else
14     return buf_i(x)
15 }
16 proc write(Transaction T_i, Var x, Value v) {
17   WSet_i ← WSet_i ∪ {x}
18   if :open(T_i, x) = false
19     return abort(T_i)
20   else {
21     buf_i(x) ← v
22     return ok_i
23   }
24 }
25 proc commit(Transaction T_i) {
26   // Lock the write set.
27   for x ∈ WSet_i in order
28     lock lk(x) → W
29   if ∃x ∈ WSet_i: owner(lk(x)) ≠ T_i
30     return abort(T_i)
31
32   // Validate the read set.
33   if ∃x ∈ RSet_i:
34       mode(lk(x)) = W or version(x) > rv_i
35     return abort(T_i)
36
37   // Increment and fetch global version clock.
38   L ← location(T_i)
39   lc_L ← lc_L + 1
40
41   // Commit and release.
42   for x ∈ WSet_i {
43     R ← location(x)
44     if L ≠ R
45       move x from R to L
46     x ← buf_i(x)
47     version(x) ← lc_L
48     unlock lk(x)
49   }
50   return C_i
51 }
52 proc abort(Transaction T_i) {
53   for x ∈ WSet_i
54     if owner(lk(x)) = T_i
55       unlock lk(x)
56   return A_i
57 }
```

```
58 proc :open(Transaction T_i, Var x) {
59   L ← location(T_i)
60   R ← location(x)
61
62   // Retrieve object.
63   if lc_R < lc_L
64     lc_R ← lc_L
65   copy x from R to L as buf_i(x)
66
67   if L = R {
68     // Open local object.
69     if version(x) > wv_i
70       return false
71     else
72       return true
73   } else {
74     // Open remote object.
75     if lc_L < lc_R
76       lc_L ← lc_R
77     if wv_i < lc_R {
78       if ∃y ∈ RSet_i and Q = location(y):
79           version(y) > lc_Q
80         return false
81       wv_i ← lc_R
82     }
83     return true
84   }
85 }
```

Figure 4.17: TFA.

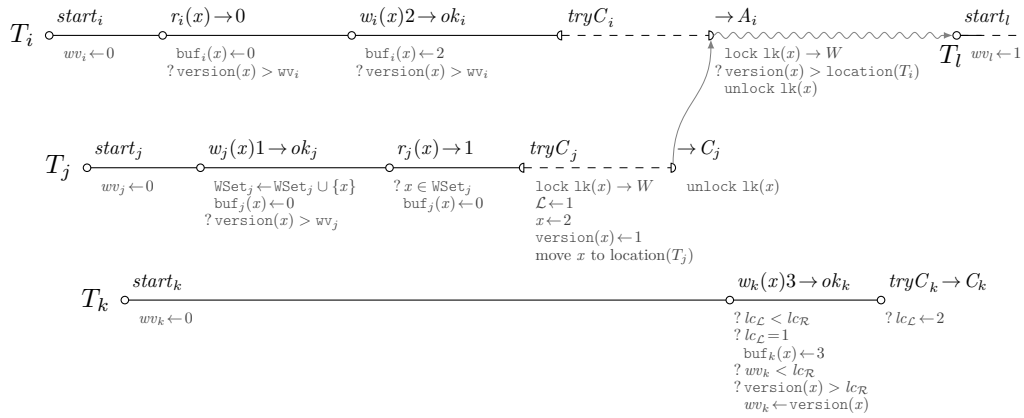


Figure 4.18: An example TFA history.

transaction T_j updated the version of x to 1 during commit, it also adjusted its local clock to 1. Then, when transaction T_k writes to x it can adjust its own local clock to T_j's local clock, since it did not actually view or modify any variable prior to this point. Then, T_k uses T_i's local clock to validate x's version, rather than a stale value read from a global clock. Given these adjustments, T_k is able to validate correctly during commit.

The authors show in [69] that TFA is opaque. In addition, just like DTL2, TFA does not deadlock. On the other hand it also aborts on conflict, which may cause issues with irrevocable operations.

4.3 Non-distributed Pessimistic TM

A great majority of TM concurrency control algorithms are optimistic. Nevertheless, a handful of partially or fully-pessimistic non-distributed TM algorithms were recently introduced. In [92] the authors introduced the idea of irrevocable transactions. The system can execute irrevocable transactions concurrently to ordinary transactions, but irrevocable transactions always execute one at a time and never abort. This makes the execution safe for irrevocable operations within these transactions, and prevents wasted effort due to aborting long-running transactions. However, such transactions severely limit the level of parallelism of which the system is capable. In [62], the authors introduce a partially pessimistic TM where read-only transactions use multiversioning to prevent aborts. However, the maintenance of multiple versions introduces a high overhead, and read/write transactions aborted nevertheless. In [9], the authors propose a system with irrevocable single-version read-only transactions that used locking for each read variable. However, this introduces overhead that makes the system outperformed by optimistic TMs. While not exactly a pessimistic system *per se*, Twilight STM [13] is an interesting system that relaxes isolation to allow conflicting transactions to reconcile using so-called twilight code at the end of the transaction and commit nevertheless. If a transaction reads a value that was modified by another transaction since its start, twilight code can re-read the changed variables and re-write the variables the transaction modified to reflect the new state, allowing the transaction to commit anyway. Even though the operations are re-executed, as per optimistic concurrency control, it means that transactions that execute twilight never need to abort.

In [56] the authors propose pessimistic non-distributed TM that defers read/write

transactions to execute sequentially (as in [92]) but allows parallel read-only transactions. The read/write transactions maintain consistency by waiting for concurrent read-only transactions to complete, before making any updates in memory. This idea was improved upon in *Pessimistic Lock Elision* (*PLE*) [1], where a number of optimizations were introduced, including encounter-time synchronization, rather than commit-time. We examine these algorithms in Sections 4.3.1 and 4.3.2 respectively.

SemanticTM [10, 26] is another pessimistic non-distributed TM. Rather than using versioning or blocking, transactions are scheduled and place their operations in bulk into a producer-consumer queues attached to variables. The instructions are then executed by a pool of non-blocking executor threads that use statically derived access sets and dependencies between operations to ensure the right order of execution. The scheduler ensures that all operations of one transaction are executed consistently and in the right order. We examine the system further in Section 4.3.3.

4.3.1 Matveev and Shavit's Pessimistic TM

In [56] the authors propose a fully-pessimistic TM algorithm for multicores operating in the variable model. The algorithm is loosely based on TL2 [21] and the work on irrevocable transactions in [92], and uses the quiescence mechanism from [20]. The result is a TM algorithm that executes its RW transactions in series and prevents them from writing to location that may be read by read-only transaction. In effect, the authors introduce a fully-pessimistic non-aborting commit-time TM. As far as we can tell the authors never named their algorithm: they refer to it in the original paper as well as later work simply as *pessimistic TM* (*PTM*) and go out of their way to indicate this is not its proper name. We will therefore refer to the algorithm as *Matveev and Shavit's Pessimistic TM*, which we abbreviate as *MS-PTM*. We describe the operation of the MS-PTM algorithm in detail below and give the pseudocode in Fig. 4.19.

Note that the pseudocode indicates memory fences, processor instructions that explicitly order the execution of instructions in a block of code. These are necessary for the original application of MS-PTM in local multi-core processors. We omit such low-level details in the other descriptions of algorithms in this dissertation, but retain them here for verisimilitude with the original proposition in [56].

The synchronization in MS-PTM is based on a global version counter `gv`, transaction-local version counters $\texttt{rv}_i$, and each variable's x version $\texttt{version}(x)$, much like TL2. Also like TL2, MS-PTM specifies two types of transactions: RW and R. R and RW transactions are synchronized by blocking reads and deferring writes to commit via buffering. The mechanism is based on the global version counter. During start each transaction reads the global version and stores it as its read version. Write operations do not block, they simply store the written value in the buffer. A read operation in any transaction will block until the transaction's read version differs from the variables version. If the read version is equals to the variable's version, this means that some RW transaction is committing. During the commit procedure RW transactions increment the global value twice: once at the outset, and once at the finish. R transactions that started after a RW transaction started committing will therefore be blocked until the RW transaction finishes committing. R transactions that started before the RW transaction started committing will not be blocked by the RW transaction. Instead the RW transaction will not start writing from buffers into memory until all such R transactions finish execution. Specifically, RW transactions use the quiescence mechanism which makes them wait until all other transactions whose read versions are lower than the current global version update their read version to be equal or greater than the global version. R transactions can meet this condition when they update their version to infinity during commit.

```
1  proc start(Transaction T_i) {
2    waiting_i ← true
3    memory fence
4    while true {
5      if waiting_i = false {
6        rv_i ← gv
7        memory fence
8        return ok_i
9      }
10     if mode(lk^g) ≠ W {
11       try lock lk^g → W
12       if owner(lk^g) = T_i {
13         waiting_i ← false
14         rv_i ← gv
15         memory fence
16         return ok_i
17       }
18     }
19   }
20 }
21 proc read(Transaction T_i, Var x) {
22   if x ∈ WSet_i:
23     return buf_i(x)
24   if progress_i = false
25     if version(x) = rv_i {
26       wait until rv_i ≠ gv
27       progress_i ← true
28     }
29   return x
30 }
31 proc write(Transaction T_i, Var x, Value v) {
32   WSet_i ← WSet_i ∪ {x}
33   buf_i(x) ← v
34   return ok_i
35 }
36 proc commit(Transaction T_i) {
37   // Synch and update versions.
38   if rv_i is even
39     wait until rv_i ≠ gv
40   for x ∈ WSet_i
41     version(x) ← rv_i + 1
42
43   // First global version increment.
44   gv ← gv + 1
45   rv_i ← gv
46   memory fence
47
48   // Signal next writer.
49   if ∃T_j: waiting_j = true
50     waiting_j ← false
51   else
52     unlock lk^g
53
54   // Quiescence.
55   for ∀T_j: j ≠ i
56     if rv_j < gv
57       wait until rv_j ⩾ gv
58
59   // Write the write set.
60   for x ∈ WSet_i
61     x ← buf_i(x)
62   memory fence
63
64   // Second global version increment.
65   gv ← gv + 1
66   return C_i
67 }
```

(a) RW Transactions.

```
68 proc start(Transaction T_i) {
69   rv_i ← gv
70   memory fence
71 }
72 proc read(Transaction T_i, Var x) {
73   if progress_i = false
74     if version(x) = rv_i {
75       wait until rv_i ≠ gv
76       progress_i ← true
77     }
78   return x
79 }
80 proc commit(Transaction T_i) {
81   rv_i ← ∞
82   return C_i
83 }
```

(b) R Transactions.

Figure 4.19: MS-PTM.

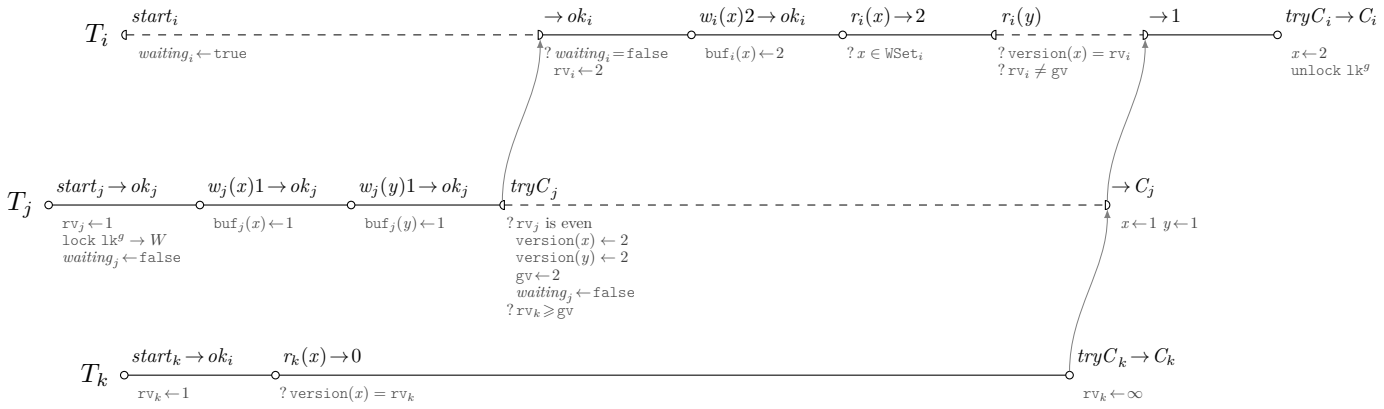


Figure 4.20: An example MS-PTM history.

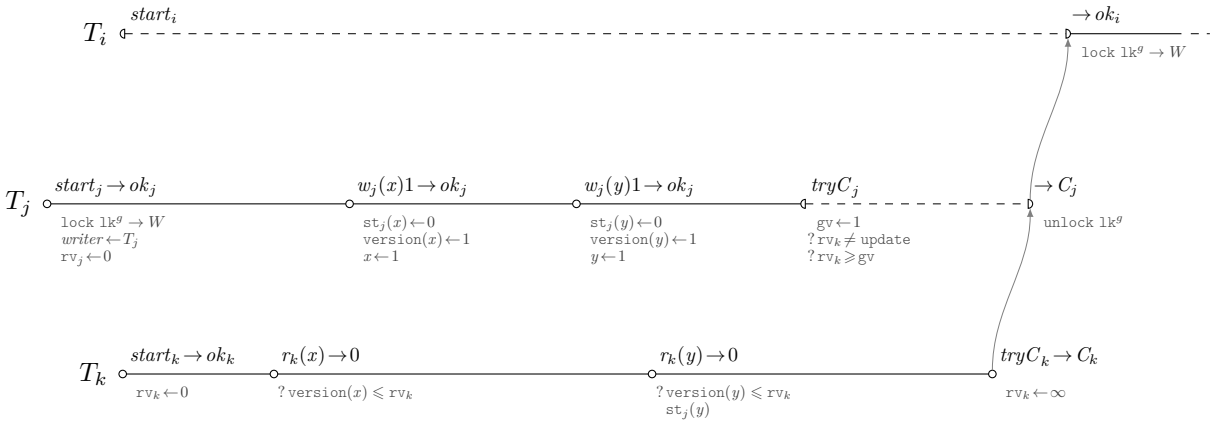


Figure 4.21: An example PLE history.

In addition to this mechanism, once a transaction was blocked and managed to proceed, it cannot be blocked. This is because transactions are blocked during read only if they start during an RW transaction's commit. Thus, if a transaction was blocked and released, the transaction that blocked it had already committed, and any future RW transactions will wait until this transaction commits before modifying the variables. There is no purpose to checking versions in such cases, so a transaction-local variable $progress_i$ is used instead.

Only one RW transaction is allowed to execute operations at a time. Specifically, during transaction start, the first RW transaction acquires a global lock, which causes any other RW transactions to have to wait. However, instead of releasing the lock during commit, the transaction "passes" the lock to the next waiting transaction by signaling it using the transaction-local $waiting_i$ variable. The lock is released only if there are no more waiting RW transactions.

We give an example execution under MS-PTM in Fig. 4.20. Here, two RW transactions T_i, T_j, and one R transaction T_k execute concurrently. T_j starts first, acquires the global lock and receives the read version of 1. Hence, when T_i attempts to start later on, it cannot, since it cannot acquire the global lock, so it waits until one of two things happen: either the lock is released, or the currently running RW transaction releases T_i by falsifying the $waiting_i$ variable. Meanwhile, T_j executes writes on variables x and y. The values are not written to memory, but are stored in the buffer. This does not prevent the R transaction T_k from executing a read on x, which reads the value 0, without taking the RW transaction's operations into account. When T_j finishes executing operations it proceeds to commit. It updates the versions of each variable in its write set and increments the global version. Then it sets $waiting_i$ to false, which releases T_i to act. This means, that soon after T_j invokes $tryC_j$, T_i can execute its own write on x using the buffer. It can also execute a local read on x, since it will simply read the value from said buffer. However, when T_i tries to perform a non-local read on y it encounters a situation where the variables version is equal to its read version, meaning that another RW transaction is in progress, and it must wait until that transaction finishes committing and increments the global version further. However T_j cannot commit yet either. It must first perform quiescence and wait for concurrent R transactions to finish. Thus it waits until T_k sets its read version to infinity. T_k does so during commit. Then, T_j can finish its commit by writing its write set to memory, and incrementing the global version. This, then allows T_i to finish its read and eventually commit. Since T_i is the last RW transaction, it unlocks the global lock.

The authors do not discuss the properties of MS-PTM in [56], but since RW transactions wait for concurrent R transactions, no transactions ever abort, and the algorithm is commit-time, then on intuition MS-PTM is opaque. MS-PTM is non-aborting and does not deadlock.

Given the amount of communication among transactions that MS-PTM requires, the algorithm is not trivial to implement in distributed environments. An implementation would require either that a given client be able to poll other clients for the status of their transactions, or a global structure to coordinate. In the first case, communication among clients is likely to be costly (especially in geo-distributed systems) or even impossible (e.g. due to firewalls). The second solution introduces a major bottleneck and a single point of failure. Thus, a more comprehensive re-tooling is required.

4.3.2 Pessimistic Lock Elision

The *Pessimistic Lock Elision* (*PLE*) [1] is a non-distributed pessimistic TM algorithm. It is an adaptation of MS-PTM to the encounter-time update approach, rather than

```
1 proc start(Transaction T_i) {
2   lock lk^g
3   writer ← T_i
4   rv_i ← gv
5   memory fence
6   return ok_i
7 }
8 proc read(Transaction T_i, Var x) {
9   return x
10 }
11 proc write(Transaction T_i, Var x, Value v) {
12   if st_i(x) = ⊥
13     st_i(x) ← x
14   version(x) ← rv_i + 1
15   x ← v
16   return ok_i
17 }
18 proc commit(Transaction T_i) {
19   gv ← gv + 1
20   memory fence
21
22   // Quiescence.
23   for ∀T_j : i
24     wait until rv_i ≠ update
25     wait until rv_i ⩾ gv
26
27   unlock lk^g
28   return C_i
29 }
```

(a) RW Transactions.

```
30 proc start(Transaction T_i) {
31   rv_i ← update
32   memory fence
33   rv_i ← gv
34   memory fence
35   return ok_i
36 }
37 proc read(Transaction T_i, Var x) {
38   if version(x) ⩽ rv_i
39     return x
40   T_j ← writer
41   if st_j(x) ≠ ⊥
42     return st_j(x)
43   else
44     return x
45 }
46 proc commit(Transaction T_i) {
47   rv_i ← ∞
48   return C_i
49 }
```

(b) R Transactions.

Figure 4.22: PLE.

commit-time. As such, PLE can be used interchangeably with locks in such contexts as hardware lock elision and as a fallback for optimistic hardware transactional memory. It also confers a performance advantage over MS-PTM. We describe the operation of the algorithm below and give the pseudocode in Fig. 4.22.

When RW transactions execute, values are written directly to memory, however, prior to executing the write transaction, T_i will append the old value of the variable x in buffer $\texttt{st}_i(x)$. This will allow concurrent R transactions to access either the current or the old version of the variable. Basically, an R transaction chooses the current version if it is consistent with its read version $\texttt{version}(x) \leqslant \texttt{rv}_i$, or if the RW transaction did not write anything yet. Otherwise, the buffered version is retrieved from the currently running RW transaction. Reads in RW transactions execute unconditionally using the current version, since no other transaction can interfere with the variable. When a RW transaction commits, it increments gv, waits for executing R transactions via quiescence, and releases the global lock. The extra wait step in quiescence is included to ensure the order of operations on the global version and read version counters.

We give an example execution under PLE in Fig. 4.20. Here, two RW transactions T_i, T_j, and one R transaction T_k execute concurrently. T_j starts first, so it acquires the global lock, receives the read version of 0, and names itself the active writer. Transaction T_i attempts to start a moment later, but cannot do so, since the global lock is taken, so it waits. However transaction T_k starts unimpeded and reads x. Since the version of x is consistent, it reads the memory location directly. Then, T_j executes two consecutive writes, one on x and one on y. Both variables are therefore not in version 1, and T_j maintains their old version in its buffer. When T_k executes a read on y next, the current version of y is not consistent with the transaction's read version, so it reads it from T_j's buffer. Next, T_j attempts to commit, so it increments the global version. However it cannot proceed, since there is an extant R transaction, so it waits until T_k commits, which it detects by monitoring T_k's read version. Once T_k commits it sets its read version to infinity, which allows T_j to commit. Once T_j commits it releases the global lock, which

allows T_i to begin execution.

The authors do not discuss the algorithm's safety in [1], but since RW transactions execute sequentially and wait for R transactions on commit, no transactions ever abort, R transactions read a stale but consistent state of the system, then on intuition PLE is opaque. PLE is non-aborting and does not deadlock.

4.3.3 SemanticTM

SemanticTM [10, 23] is a unique wait-free, fully pessimistic (abort-free) TM system with instruction-level parallelism operating in the variable model.

SemanticTM maintains a list associated with each shared variable in the system, which we will refer to as an *execution queue*, where transactions place operations to be executed along with the operation's *dependencies*. Dependencies are relations between operations in a single transaction that determine the order in which the operations must be executed. For instance, given a write operation which writes value v to variable x, the operation cannot proceed until v is computed: if v is computed by executing another operation, let's say a read on y, then the write depends on the read. The authors define a number of dependency types, including ones for arithmetic, conditional expressions, and loops, as well as ones for operations on shared variables.

Given such execution queues, SemanticTM employs a pool of independent *worker threads*. Each worker thread randomly (but uniformly) selects a variable, and tries to execute the first operation in the queue, provided its dependencies allow it. This process is done without blocking, which means workers execute in a wait-free fashion. This may cause several workers to execute the same instruction in parallel, but it does not violate safety.

The consistency of operation executions is assured in SemanticTM due to the order in which operations are placed onto execution queues. First, each operation $\mathtt{m}_i$ on variable x from transaction T_i is placed onto the execution queue for x after all of the operations on x in T_i on which $\mathtt{m}_i$ depends. Second, given any two transactions T_i and T_j, for each variable x, if any of T_i's operations precede any of T_j's operations in the execution queue for x, then all of T_i's operations precede all of T_j's operations in the execution queue for any variable.

SemanticTM does not specify how these conditions are satisfied by the system, indicating rather that the operations are to be loaded onto execution queues statically. This would imply that either the entire code of all transactions in the system, as well as the order in which they execute is known *a priori*. We consider this an impractical assumption for general purpose TM, especially with an outlook towards distributed systems. To our best knowledge, this is also a much stronger assumption than is made by any other TM.

The alternative is to use a run-time scheduler that maintains a proper order among operations from different transactions for each variable's execution queue. In order to do so, the scheduler itself requires some form of synchronization. In the simplest case, a global lock can be used to prevent one transaction from enqueueing operations while another is in progress. This, however, implies that (potentially) conflicting transactions are serialized by the scheduler. A more subtle scheduler may be implemented, effectively using some transactional "front end" to enqueue operations to be executed by the SemanticTM "back end."

The authors of [23] propose that SemanticTM is opaque (although the proof is not given) assuming an appropriate scheduler. Regardless, SemanticTM is trivially serializable. Assuming a static or wait-free scheduler, SemanticTM is also wait-free. Finally, no transaction ever aborts, but irrevocable operations can be executed multiple times if two

or more processors simultaneously access the same execution queue.

4.4 Optimistic TM with Early Release

A number of TM systems employ early release to improve parallelism. One example is Dynamic STM [43], the system that can be credited with introducing the concept of early release in the TM context. Dynamic STM allows transactions that only perform read operations on particular variables to (manually) release them for use by other transactions. However, it left the assurance of safety to the programmer, and, as the authors state, even linearizability cannot be guaranteed by the system. The authors of [83] expanded on the work above and evaluated the concept of early release with respect to read-only variables on several concurrent data structures. The results showed that this form of early release does not provide a significant advantage in most cases, although there are scenarios where it would be advantageous if it were automated.

Dependency Aware STM [65] (DATM) is another noteworthy system with an early release mechanism. DATM is an optimistic multicore-oriented TM based on TL2 [21], augmented with early-release support. It allows a transaction T_i to write to a variable that was accessed by some uncommitted transaction T_j, as long as T_j commits before T_i. DATM also allows transaction T_i to read a speculative value, one written by T_j and accessed by T_i before T_j commits. DATM detects if T_j overwrites the data or aborts, in which case T_i is forced to restart. We examine DATM in detail in Section 4.4.1.

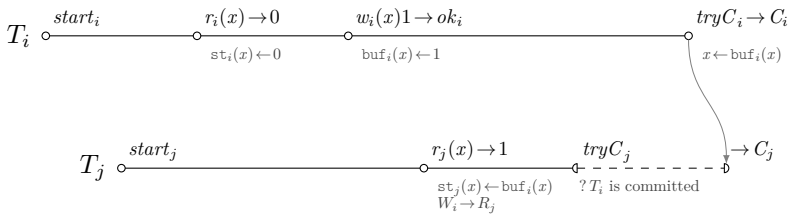
4.4.1 Dependence Aware TM

Dependency Aware TM (*DATM*) [65, 66] is a non-distributed commit-time optimistic TM concurrency control algorithm based on TL2 but extended with an early release mechanism. Specifically, DATM tracks dependencies between transactions and either passes uncommitted data between them, or delays some of them to prevent their conflicts from causing inconsistencies. In effect, conflicting transactions can be committed. We explain the details of the dependency awareness mechanism below.

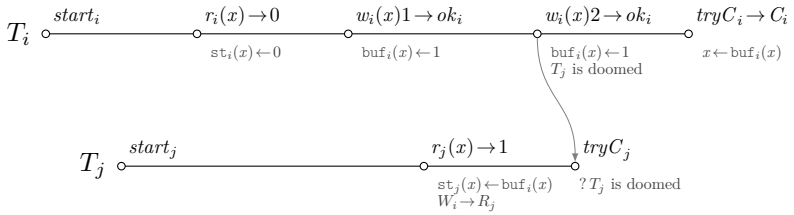
DATM specifies three kinds of dependences between transactions. If transaction T_i writes some variable x and then T_j reads x afterward, then they are in a *write-read dependence*, denoted $W_i \rightarrow R_j$. If transaction T_i reads some variable x and then T_j writes to x afterward, then they are in a *read-write dependence*, denoted $R_i \rightarrow W_j$. If transaction T_i writes to some variable x and then T_j writes to x afterward, then they are in a *write-write dependence*, denoted $W_i \rightarrow W_j$. If there is any dependece from T_i to T_j we say T_j *depends on* T_i.

DATM transactions respond to these dependences using either *forwarding* or *ordering*. If two transactions T_i and T_j are in $W_i \rightarrow R_j$, then the value written by T_i is forwarded to T_j when T_j performs a read. This means that if T_i previously wrote some new value to x (and stored it in buffer $\texttt{buf}_i(x)$, since DATM is commit-time), then when T_j executes a read operation on x it does not read the value of x directly from memory, but instead reads the value of x from $\texttt{buf}_i(x)$. If two transactions T_i and T_j are in any relation $W_i \rightarrow R_j$, $W_i \rightarrow W_j$ or $R_i \rightarrow W_j$, then the transactions are ordered: T_j cannot perform its commit until T_i commits or aborts (in particular it cannot write its write set to memory). In addition, if T_i aborts and $W_i \rightarrow R_j$, then this forces T_j to also abort (thus, a cascading abort occurs).

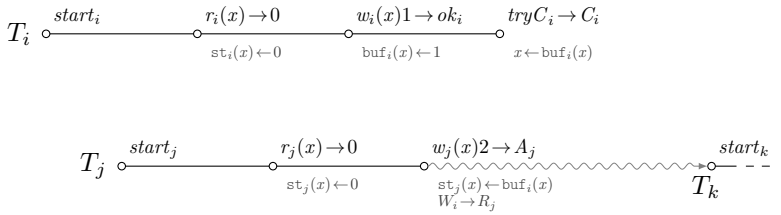
We show an example of overwriting and forwarding in Fig. 4.23. Here transaction T_i executes a read and a consecutive write on variable x. The write causes the value of x



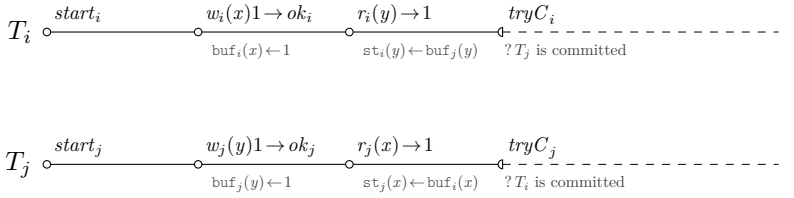
(a) Forwarding an ordering (adapted from [65]).



(b) Forwarding and overwriting.



(c) Abort due to dependence cycle (adapted from [66]).



(d) Deadlock due to dependence cycle.

Figure 4.23: Examples of DATM histories.

to be buffered in $\texttt{buf}_i(x)$. Then, transaction T_j performs a read on x. The transactions use a sequence associated with x to determine that a dependence $W_i \rightarrow R_j$ was created. On the basis of this information, T_i forwards x to T_j, meaning T_j grabs the value of x directly from $\texttt{buf}_i(x)$. Afterward, T_j attempts to commit. However, since the dependence $W_i \rightarrow R_j$ exists, DATM defers T_j's commit until T_i commits. During commit T_i updates the memory using the values it stored in the buffer. Finally, once T_i finishes its commit procedure, T_j can finish to commit as well.

DATM buffers values read by each transaction T_i from each variable x in a separate buffer which we indicate $\texttt{st}_i(x)$, which means that repeated reads do not need to engage in forwarding or re-reading the value from memory. Transactions also read its own written value by moving it from $\texttt{buf}_i(x)$ to $\texttt{st}_i(x)$, making local reads always consistent. On the other hand *overwriting*, repeated writes by T_i to some variable x causes any transaction for which there is a $W_i \rightarrow R_j$ dependece to be forced to abort. A doomed flag is set for the dependent transaction in such cases, which every transaction checks during commit. Since DATM does not have any mechanism to determine *a priori* whether forwarding a variable will cause an abort or not, overwriting cannot be avoided. We show an example of overwriting in Fig. 4.23b.

DATM is able to prevent deadlocks due to dependence cycles on the same variable. If such a cycle appears, the transactions detect it by scanning the sequence of transactions accessing the variable, and one of the transactions aborts. However, two DATM transactions can deadlock during commit if they conflict on multiple variables (they can enter a dependence cycle). We give examples of a prevented and undetected deadlock in Fig. 4.23c and 4.23d, respectively.

The authors demonstrate in [65] that DATM is serializable (conflict serializable). Since transactions can read from live transactions, it should be evident, that DATM does not guarantee properties such as strictness, rigorousness, TMS1 and TMS2, and opacity. Since commits of conflicting transactions are ordered in accordance to the order of accesses of variables, and a $W_i \rightarrow R_j$ dependence induces a rollback in the transaction that reads from an aborting transaction, then DATM guarantees recoverability. Since aborting transactions can cause transactions that depend on them to also abort, DATM is not cascadeless.

4.5 Summary

We present a summary of the characteristics of the examined algorithms in Table 4.1. There, the *approach* column indicates whether a given TM uses the optimistic or pessimistic approach to concurrency control. The *progress* column indicates whether the algorithm is blocking or wait-free. Here we describe SemanticTM as wait-free with an asterisk, to indicate the assumptions placed on the scheduler. The *updates* column specifies whether the algorithm is encounter time or commit-time. The *aborts* column specifies what scenarios can cause a transaction to abort, be it a deadlock, a conflict, a cascading abort, or an arbitrary abort (invoked manually by the programmer). The *a priori* column indicates what information must be known to each transaction before it starts: some TMs require that the read set and the write set be known, while others require only a union of the two, while other still place additional constraints. The *objects* column indicates how the object model must be defined for the TM to operate. Note that algorithms that operate in a heterogeneous model can be used with homogeneous and variable models without modification, but they will not optimize with regards to read operations. Similarly, TMs operating in the homogeneous model can be used in the vari-

Algorithm	**Approach**	**Progress**	**Updates**	**Aborts**	**A priori**	**Objects**	**Deadlock**	**Safety**	**Early release**	**Irrevocable**
B2PL	pessimistic	blocking	encounter-time	on deadlock	∅	any	yes	strict serializable	yes	abortable
C2PL	pessimistic	blocking	encounter-time	abort-free	*RSet*, *WSet*	any	no	strict serializable	yes	correct
S2PL	pessimistic	blocking	encounter-time	on deadlock	∅	any	yes	strict	reads	abortable
R2PL	pessimistic	blocking	encounter-time	on deadlock	∅	any	yes	rigorous	no	abortable
CS2PL	pessimistic	blocking	encounter-time	abort-free	*RSet*, *WSet*	any	no	opaque	reads	correct
CR2PL	pessimistic	blocking	encounter-time	abort-free	*RSet*, *WSet*	any	no	opaque	no	correct
CAS2PL	pessimistic	blocking	encounter-time	arbitrary abort	*RSet*, *WSet*	any	no	opaque	reads	user abortable
CAR2PL	pessimistic	blocking	encounter-time	arbitrary abort	*RSet*, *WSet*	any	no	opaque	no	user abortable
BVA	pessimistic	blocking	encounter-time	abort-free	*ASet*	heterogeneous	no	opaque	no	correct
SVA	pessimistic	blocking	encounter-time	abort-free	*ASet*, suprema	heterogeneous	no	strict serializable	yes	correct
TL2/DTL2	optimistic	blocking	commit-time	on conflict	∅	variable	no	opaque	no	abortable
TFA	optimistic	blocking	commit-time	on conflict	∅	homogeneous	no	opaque	no	abortable
MS-PTM	pessimistic	blocking	commit-time	abort-free	∅	variable	no	opaque	no	correct
PLE	pessimistic	blocking	encounter-time	abort-free	∅	variable	no	opaque	no	correct
SemanticTM	pessimistic	wait-free*	encounter-time	abort-free	*ASet*, dependencies	variable	no	opaque	no	repeatable
DATM	optimistic	blocking	commit-time	on overwriting, deadlock, and cascade	∅	variable	yes	conflict serializable	yes	abortable

Table 4.1: Summary comparison of discussed TM algorithms.

able model. On the other hand, algorithms marked as *any* operate in the variable model, but can be trivially lifted to any of the other models. The *deadlock* column indicates whether a deadlock can occur (at all) in this algorithm. The *safety* column indicates the strongest safety property or consistency condition satisfied by the transaction. The *early release* column indicates whether the algorithm allows conflicting transactions to simultaneously use the same shared object, e.g. by releasing a lock before committing. Some algorithms allow this only with respect to objects that they only read. Finally, the *irrevocable* column indicates how the algorithm handles the execution of irrevocable operations. Such an operation may be subject to aborts or repeated execution without aborts, which is incorrect behavior. We differentiate between aborts caused by the concurrency control algorithm and those caused by the transaction's programmer—if the transaction elects to abort arbitrarily, aborting irrevocable operations is the programmer's wish, and, therefore, correct behavior.

Properties for TM with Early Release

Note that despite there being some strong safety properties that allow early release like VWC, live opacity, and elastic opacity, the TM algorithms that use the early release technique do not satisfy them, but instead satisfy only variants of serializability, which are relatively weak. The stronger properties cannot be satisfied by these TM algorithms because of the stringent and, we submit, impractical requirements they make with respect to abortability of transactions that employ early release. On the other hand, the behavior of these algorithms differs greatly. For instance, 2PL and SVA do not allow overwriting to occur in aborted transactions, while DATM does. Since these differences are not expressed by safety properties the algorithms satisfy, we conclude that there is a lack of adequate TM safety properties that can regulate and describe the behavior of transactions with early release.

Applicability to Distributed TM

Among the algorithms presented, a number can be used in distributed systems. These include two-phase locking algorithms, versioning algorithms, DTL2 and TFA. These algorithms are either designed specifically for the distributed context, or were successfully implemented in such systems. Out of these, versioning algorithms use a global locking structure, which is problematic for scalability in distributed systems, since all clients will have to contact a single network node. Nevertheless, the global lock can be replaced by a more comprehensive locking scheme using the knowledge of *a priori* access sets (we propose one in Chapter 6). Out of these systems TFA is specifically designed to operate in the DF model, while DTL2, versioning algorithms, and two-phase locking algorithms fit the CF model.

MS-PTM, PLE, and DATM are less well-suited for distributed systems, since they use global locking structures, which are a scalability stumbling block in such systems (all clients communicating with a single lock), and a potential single point of failure. Secondly, MS-PTM, PLE, and DATM require extensive communication between transactions (e.g. quiescence). However, communication between clients is impractical in many architectures, since it requires client applications to serve other client applications. Clients may be geo-distributed, which introduces additional delays, placed behind firewalls, which prevents them from communicating, or have limited processing capability, as in the case of mobile devices. Hence, in order to apply these algorithms in distributed systems, methods must be devised for transactions to push the required information to other transactions. Note that a global structure collecting the required information is not acceptable for the same reasons as a global lock. Hence, more comprehensive solutions are required.

An application of SemanticTM in distributed systems is difficult to envision, due to its requirement for transactional operations to be placed onto executor queues in order. Assuming that requirement is met, Semantic TM can be used to execute transactional operations in a distributed system, although its model does not necessarily fit CF nor DF. On the other hand, meeting the requirement for operation order in executor queues is unlikely to be met statically, and so, will require a distributed on-line scheduler to be employed during execution. Such a scheduler is a system whose complexity will match those of a TM, so we consider SemanticTM not to be applicable to distributed systems directly.

Note that out of the systems applicable to distributed systems only 2PL (specifically C2PL) and versioning algorithms provide support for irrevocable operations.

5
New Properties

In this chapter we introduce *last-use opacity*, a strong TM safety property that allows early release, but makes provisions for transaction safety. First we motivate the need for the property based on our analysis of the existing properties and how they apply to existing algorithms (Chapter 3). Then we provide an intuition, as well as the formal definition of the property, which we further explain using numerous examples. Finally, we provide an in-depth discussion of the scenarios allowed by the property, the implications for TM consistency in practice, and the relationship between last-use opacity and other TM safety properties and consistency conditions.

The introduced property allows a small class of inconsistent views to occur, which, we argue, are relatively harmless in practice, and only occur in a specific system model. Nevertheless, we follow by introducing a strong version of the last-use opacity property, which eliminates the inconsistent views altogether at the cost of parallelism. As such, this property is more generally applicable, but more difficult to enforce. The results presented in this chapter extend our work in [76, 77, 79].

5.1 Last-use Opacity

The survey of properties shows that, while there are many safety properties for TM with a wide range of guarantees they provide, with respect to early release they fall into three basic groups.

The first group consists of properties that allow early release but do not prevent overwriting: serializability and recoverability. These properties do not regulate what can be seen by aborting transactions. In effect, they allow any dangerous scenario to occur with respect to early release, as long as the situation is resolved by aborting offending transactions. As argued in [33], this is insufficient for TM in general, because operating on inconsistent state may lead to uncontrollable errors, including crashing the process.

The second group consists of properties that preclude the dangerous situations allowed by the first group. This group includes cascadelessness, strictness, rigorousness, opacity, markability TMS1, and TMS2. The properties in this group forbid early release altogether, thus solving all related consistency problems, but making them unusable in conjunction with the early release technique.

The third group allows early release but precludes overwriting and reading from aborting transactions. It includes live opacity, elastic opacity, and VWC. These properties seem to provide a reasonable middle ground between allowing early release and eliminating inconsistent views. However, these properties forbid transactions to release early and abort. As such they can be useful only for TM operating in the commit-only model, or in TM systems where transactions that release early become irrevocable.

On the other hand the commit-only model limits the applicability of such TMs in certain contexts, since arbitrary aborts can be a necessary prerequisite for some applications. For instance, aborts are a necessary part of recovery mechanisms that bring the TM system to a consistent state as a result of a partial failure. Another example is a deadlock recovery system, which aborts transactions to eliminate wait dependency cycles. Furthermore, TM systems that provide the programmer access to arbitrary aborts are more expressive. That is, there are situations where the programmer may want to withdraw any changes made by a transaction mid-execution. Reverting changes *ad hoc* detracts from the readability of the code, and it is usually less efficient. The problem becomes magnified in distributed TM, where performing an *ad hoc* abort and compensation remotely usually comes at a price of extra network communication overhead. Thus, for DTM and TM systems in the arbitrary abort model, live opacity and VWC are not useful.

On the other hand, if transactions are allowed to abort in general, but not in the case of ones with early release, then this results in additional complexity to a TM (see e.g., [92]). Moreover, in applications like distributed computing, transaction aborts may be induced by external stimuli, so it can be completely impossible to prevent transactions from aborting [74]. In addition, some of those properties also have specific problems that make them difficult to apply widely in practice. For instance, elastic opacity introduces unnecessary restrictions on the order of operations within a transaction, while simultaneously diverging from the minimal standard set by serializability. Meanwhile, live opacity arbitrarily precludes transactions that read variables released early from releasing early themselves.

In summary, properties from the first group are not adequate for *any* TM and those from the second group do not allow any form of early release. The third group imposes an overstrict requirement that transactions which release early be irrevocable. None of the properties provide a satisfactory, strong safety property that could be used for a TM with early release in general. Thus, guarantees given by a TM where early release is a necessary component, but where transactions cannot be prevented from aborting, cannot be adequately expressed with the existing properties.

5.1.1 Intuition

We present *last-use opacity*, a new TM safety property that provides strong consistency guarantees and allows early release without compromising on the ability of transactions to abort. The property is based on the preliminary work in [76, 77].

The idea of last-use opacity hinges on identifying the *closing write* operation execution on a given variable in individual transactions. Informally, a closing write on some variable is such, that the transaction which executed it will not subsequently execute another write operation on the same variable in any *possible* extension of the history. What is possible is determined by the program that is being evaluated to create that history. Knowing the program, it is possible to infer (to an extent) what operations a particular transaction will execute. Hence, knowing the program, we can determine whether a particular operation on some variable is the last possible such operation on that variable within a given transaction. Thus, we can determine whether a given operation is the

```
1 subprogram P1 {
2   transaction { // spawns as T1
3     x ← 1
4     if (y > 0)
5       x ← x + y
6     y ← x + 1
7   }
8 }
```

```
9 subprogram P2 {
10   transaction { // spawns as T2
11     y ← y + 1
12   }
13 }
```

Figure 5.1: Transactional program with closing write.

closing write operation in a transaction.

Take, for instance, the program in Fig. 5.1, where subprogram $\mathtt{P}_1$ spawns transaction T_1, and $\mathtt{P}_2$ spawns T_2. Let us assume that initially x and y are set to 0. Depending on the semantics of the TM, as these subprograms interweave during the execution, a number of histories can be produced. We can divide all of among them into two cases. In the first case T_2 writes 1 to y in line 11 (in $\mathtt{P}_2$) and this value is then read by T_1 in line 4 (in $\mathtt{P}_1$). As a consequence, T_1 will execute the write operation in line 5. The second case assumes that T_1 reads 0 in line 4 (e.g., because T_2 executed line 11 much later). In this case, T_1 will not execute the write operation in line 5. We can see, however, that in either of the above cases, once T_1 executes the write to x on line 3, then no further writes to x will follow in T_1 in any conceivable history. Thus, the write operation execution generated by line 5 is the closing write on x in T_1. On the other hand, the write operation execution generated by line 3 of $\mathtt{P}_1$ is never the closing write on x in T_1, because there exists a conceivable history where another write operation execution will appear (i.e., once line 5 is evaluated). This is true even in the second of the cases, because line 5 can be executed *in potentia*, even if it is not executed *de facto*.

Note that once any transaction T_i completes executing its closing write on some variable x, it is certain that no further modifications to that variable are intended by the programmer as part of T_i. This means, from the perspective of T_i (and assuming no other transaction modifies x) the state of x would be the same at the time of the closing write as if the transaction attempted to commit. Hence, with respect to x, we can treat T_i as if it had attempted to commit.

Last use opacity uses the concept of a closing write to dictate one transaction can read from another transaction. We give a formal definition in Section 5.1.2, but, in short, given any two transactions, T_i and T_j, last-use opacity allows T_i to read variable x from T_j if the latter is either committed or commit-pending, or, if T_j is live and it already executed its closing write on x. This has the benefit of allowing early release while excluding overwriting completely. However, last-use opacity does allow cascading aborts to occur. We discuss the guarantees given by the property in Section 5.1.4 and the implications of inconsistent views in Section 5.1.5, as well as ways of mitigating them. We compare the strength of last-use opacity with other properties in Section 5.1.6.

5.1.2 Definition

First, we define the concept of a *closing write* to some variable by a particular transaction. We do this by first defining a closing write operation invocation, and then extend the definition to complete operation executions.

Given program $\mathbb{P}$ and a set of processes Π executing $\mathbb{P}$, since different interleavings of Π cause an execution $\mathcal{E}(\mathbb{P}, \Pi)$ to produce different histories, then let $\mathbb{H}^{\mathbb{P},\Pi}$ be the set of all possible histories that can be produced by $\mathcal{E}(\mathbb{P}, \Pi)$, i.e., $\mathbb{H}^{\mathbb{P},\Pi}$ is the largest possible

set s.t. $\mathbb{H}^{\mathbb{P},\Pi} = \{H \mid H \models \mathcal{E}(\mathbb{P},\Pi)\}$.

Definition 20 (Closing Write Invocation)**.** *Given a program $\mathbb{P}$, a set of processes Π executing $\mathbb{P}$ and a history H s.t. $H \models \mathcal{E}(\mathbb{P},\Pi)$, i.e. $H \in \mathbb{H}^{\mathbb{P},\Pi}$, an invocation $inv_i\big[w(x)v\big]$ is the* closing write invocation *on some variable x by transaction T_i in H, if for any history $H' \in \mathbb{H}^{\mathbb{P},\Pi}$ for which H is a prefix (i.e., $H' = H \cdot R$) there is no operation invocation $inv_i\big[w(x)u\big]$ s.t. $inv_i\big[w(x)v\big]$ precedes $inv_i\big[w(x)u\big]$ in $H'|T_i$.*

Definition 21 (Closing Write)**.** *Given a program $\mathbb{P}$, a set of processes Π executing $\mathbb{P}$ and a history H s.t. $H \models \mathcal{E}(\mathbb{P},\Pi)$, an operation execution is the* closing write *on some variable x by transaction T_i in H if it comprises of an invocation and a response other than A_i, and the invocation is the* closing write invocation *on x by T_i in H.*

The *closing read invocation* and the *closing read* operation are defined analogously. We call a write invocation or operation that is not closing, a non-closing write invocation or operation, and so on for read invocations and operations. In transaction diagrams we mark a closing write operation execution in some history as $\dashv\!\circledcirc\!\vdash$. Note that an operation can be the ultimate operation execution in some transaction, but still not fit the definition of a closing operation execution.

If a transaction executes its closing write on some variable, we say that the transaction *decided on x.*

Definition 22 (Transaction Decided on x)**.** *Given a program $\mathbb{P}$, a set of processes Π and a history H s.t. $H \models \mathcal{E}(\mathbb{P},\Pi)$, we say transaction $T_i \in H$* decided on *variable x in H iff $H|T_i$ contains a complete write operation execution $w_i(x)v \to ok_i$ that is the closing write on x.*

Given some history H, let $\hat{\mathbb{T}}^H$ be a set of transactions s.t. $T_i \in \hat{\mathbb{T}}^H$ iff there is some variable x s.t. T_i decided on x in H. Given any $T_i \in H$, a *decided transaction subhistory*, denoted $H\hat{|}T_i$, is the longest subsequence of $H|T_i$ s.t.:

a) $H\hat{|}T_i$ contains $start_i \to ok_i$, and

b) for any variable x, if T_i decided on x in H, then $H\hat{|}T_i$ contains $(H|T_i)|x$.

In addition, a *decided transaction subhistory completion*, denoted $H\hat{\circ}|T_i$, is a sequence s.t. $H\hat{\circ}|T_i = H\hat{|}T_i \cdot \big[tryC_i \to C_i\big]$.

Given a sequential history S s.t. $S \equiv H$, $LVis(S,T_i)$ is the longest subhistory of S, s.t. for each $T_j \in S$:

a) if $i = j$ or T_j is committed in S and $T_j \prec_S T_i$, then $S|T_j \subseteq LVis(S,T_i)$,

b) if T_j is not committed in S but $T_j \in \hat{\mathbb{T}}^H$ and $T_j \prec_S T_i$, and it is not true that $T_j \prec_H T_i$, then either $S\hat{\circ}|T_j \subseteq LVis(S,T_i)$ or not.

Given a sequential history S and a transaction $T_i \in S$, we then say that transaction T_i is *last-use legal in S* if $LVis(S,T_i)$ is legal. Note that if S is legal, then it is also last-use legal (see appendix for proof).

Definition 23 (Final-state Last-use Opacity)**.** *A finite history H is* final-state last-use opaque *if, and only if, there exists a sequential history S equivalent to any completion of H s.t.,*

a*) S preserves the real-time order of H,*

b*) every transaction in S that is committed in S is legal in S,*

c*) every transaction in S that is not committed in S is last-use legal in S.*

Definition 24 (Last-use Opacity)**.** *A history H is* last-use opaque *if, and only if, every finite prefix of H is final-state last-use opaque.*

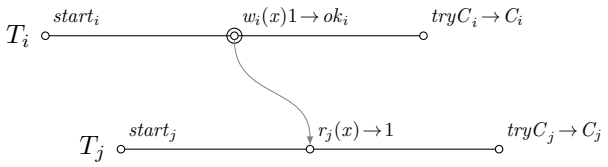


Figure 5.2: Early release—last-use opaque history.

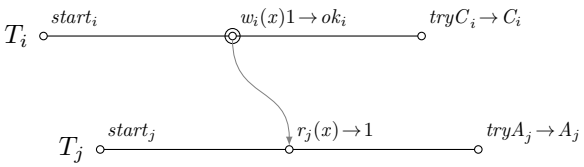


Figure 5.3: Early release to an aborting transaction—last-use opaque history.

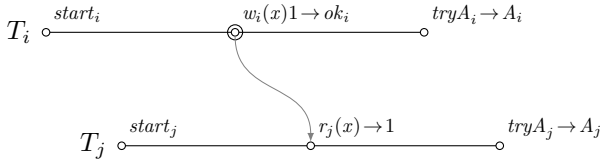


Figure 5.4: Early release with two aborting transactions—last-use opaque history.

Theorem 1. *Last-use opacity is a safety property.*

Proof. By Def. 24, last-use opacity is trivially prefix-closed.

Given H_L that is an infinite limit of any sequence of finite histories $H_0, H_1, ...$, s.t every H_h in the sequence is last-use opaque and every H_h is a prefix of H_{h+1}, since each prefix H_h of H_L is last-use opaque, then, by extension, every prefix H_h of H_L is also final-state last-use opaque, so, by Def. 24, H_L is last-use opaque. Hence, last-use opacity is limit-closed.

Since last-use opacity is both prefix-closed and limit-closed, then, by Def. 1, it is a safety property. □

5.1.3 Examples

In order to aid understanding of the property we present examples of last-use opaque histories. These are contrasted by examples of histories that are not last-use opaque. We discuss the examples below.

Early Release on Closing Write

The example in Fig. 5.2 shows T_i executing a write on x once and releasing x early to T_j. We assume that the program generating the history is such, that the write operation executed by T_i is the closing write operation execution on x. The history is intuitively correct, since both transactions commit, and T_j reads a value written by T_i. On the formal side, since both transactions are committed in this history, the equivalent sequential history would consist of all the events in T_i followed by the events in T_j and both transactions would be legal, since T_i writes a legal value to x and T_j reads the last value written by T_i to x. Thus, the history is final-state last-use opaque.

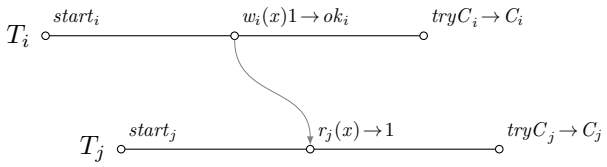


Figure 5.5: Early release before closing write—not last-use opaque.

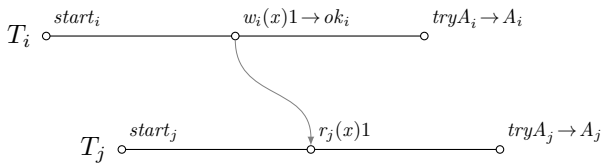


Figure 5.6: Early release with two aborting transactions before closing write—not last-use opaque.

Since last-use opacity requires prefix closeness, then all prefixes of the history in Fig. 5.2 also need to be final-state last-use opaque. We present only two of the interesting prefixes, since the remainder are either similar or trivial. The first interesting prefix is created by removing the commit operation execution from T_j, which means T_j is aborted in any completion of the history. We show such a completion in Fig. 5.3. Still, T_i writes a legal value to x and T_j reads the last value written by T_i to x, so that prefix is also final-state last-use opaque. Another interesting prefix is created by removing the commit operation executions from both transactions. Then, in the completion of the history both transactions are aborted, as in Fig. 5.4. Then, in an equivalent sequential history T_j would read a value written by an aborted transaction. In order to show legality of a committed transaction, we use the subhistory denoted *Vis*, which does not contain any transactions that were not committed in the history from which it was derived. Thus, if T_j were committed, it would not be legal, since its *Vis* would not contain a write operation execution writing the value the transaction actually read. However, since T_j is aborted, the definition of final-state last-use opacity only requires that *LVis* rather than *Vis* be legal, and *LVis* can contain operation executions on particular variables from an aborted transaction under the condition that the transaction already executed its closing write on the variables in question. Since, in the example T_i executed its closing write on x, then this write will be included in *LVis* for T_j, so T_j will be last-use legal. In consequence the prefix is also final-state last-use opaque. Indeed, all prefixes of example Fig. 5.2 are final-state last-use opaque, so the example is last-use opaque, and, by extension, so are the examples in Fig. 5.3 and Fig. 5.4.

Early Release on Non-closing Write

Contrast the example in Fig. 5.2 with the one in Fig. 5.5. The histories presented in both are identical, with the exception that the write operation in Fig. 5.2 is considered to be the closing operation execution, while in Fig. 5.5 it is not. The difference would stem from differences in the programs that produced these histories. For instance, the program producing the history in Fig. 5.5 could conditionally execute another operation on x, so, even though that condition was not met in this history, the potential of another write on x means that the existing write cannot be considered a closing write operation execution. The consequence of this is that while the example itself is final-state last-use opaque, one of its prefixes is not, so the history is not last-use opaque. The offending prefix is created

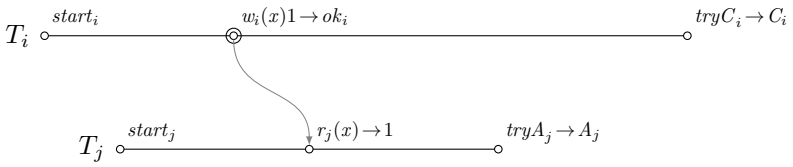


Figure 5.7: Early release to a prematurely aborting transaction—last-use opaque.

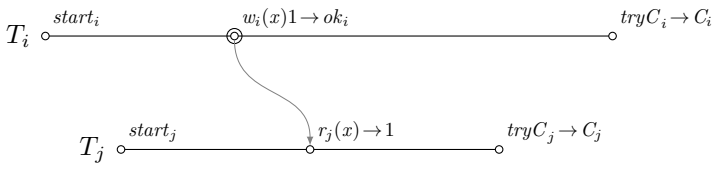


Figure 5.8: Commit order not respected—not last-use opaque.

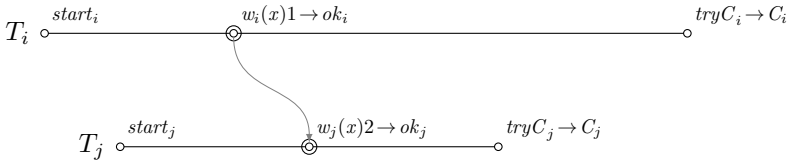


Figure 5.9: Reverse commit order in writer transactions—last-use opaque.

by removing commit operations in both transactions, so both transactions would abort in any completion, as in Fig. 5.6. Here, since T_i does not execute the closing write operation on x, then the write operation would not be included in *LVis* for T_j, so the value read by T_j could not be justified. Thus, T_j is not legal in that history, and, therefore, the history in Fig. 5.6 is not final-state last-use opaque (so also not last-use opaque). Fig. 5.6 represents the completion of a prefix of the history in Fig. 5.5, so Fig. 5.6 not being final-state last-use opaque, means that Fig. 5.5 is not last-use opaque.

Recoverability

The examples in Fig. 5.7 and Fig. 5.8, show that recoverability is required, i.e., transactions must commit in order. Last-use opacity of the example in Fig. 5.7 is analogous to the one in Fig. 5.3, since their equivalent sequential histories are identical, as are the sequential histories equivalent to their prefixes. Furthermore, intuitively, if T_j reads a value of a variable released early by T_i and aborts before T_i commits, this is correct behavior. On the other hand, the history in Fig. 5.8 is not last-use opaque, even though it is final-state last-use opaque (by analogy to Fig. 5.2). More specifically, a prefix of the history where the commit operation execution is removed from T_i is not final-state last-use opaque. This is because a completion will require that T_i be aborted, the operations executed by T_i are not going to be included in any *Vis*. Since T_j is committed, then its *Vis* must be legal, but it is not, because the read operation reading 1 will not be preceded by any writes in *Vis*. Since the prefix contains an illegal transaction, then it is not final-state last-use opaque, and thus, the history in Fig. 5.8 is not last-use opaque.

On the other hand, the example in Fig. 5.9 shows that the commitment order is not required for all conflicting transactions, just those with a reads-from relation. Here, the example is analogous to Fig. 5.8, but T_j does not read from T_i. This means that T_j's *LVis* and *Vis* will be legal regardless of whether T_i's operations are included or excluded.

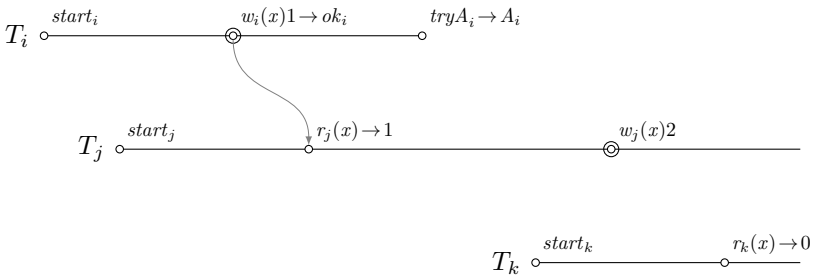


Figure 5.10: Freedom to read from or ignore an aborted transaction—last-use opaque.

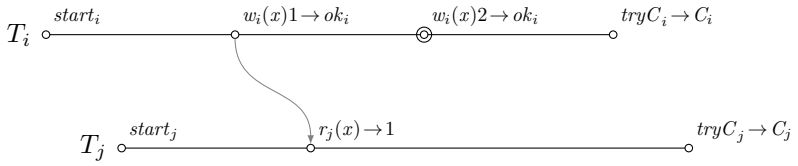


Figure 5.11: Early release with overwriting—not last-use opaque.

Then, given a sequential history equivalent to the example, where T_i precedes T_j, both T_i and T_j in such a history will be legal. Hence the history is final-state last-use opaque. Then, in all prefixes of the history T_i is aborted in the completion, whereas T_j may be either committed or aborted. If T_j is committed, then T_i will not be included in T_j's *Vis*, but this does not make *Vis* illegal, as we pointed out earlier. Similarly, if T_j is aborted, then T_i may or may not be included in T_j's *LVis*, but this is immaterial with respect to T_j's *LVis* being legal. Hence all the prefixes will be final-state last-use opaque as well, and, in effect, the example is last-use opaque.

Consistent Values

The example in Fig. 5.10 shows that a transaction is allowed to read from a transaction that eventually aborts, or ignore that transaction, because of the freedom left within the definition of *LVis*. I.e., transactions T_j is concurrent to T_i, but T_k follows T_i in real time. T_i executes a closing write on x, so T_j is allowed to include the write operation on in its *LVis*. Since T_j sees the value written to x by that write, T_j includes the write in *LVis*. On the other hand, T_k cannot include T_i's write in *LVis*, since T_i aborted before T_k even started, so the write should not be visible to T_k. On the other hand T_k is allowed to include T_j in its *LVis*. T_k should not do so, however, since it ignores T_j as well as T_i (which makes sense as T_j is doomed to abort). Hence T_k reads the value of x to be 0. If T_j is included in T_k's *LVis*, reading 0 would be incorrect. Hence, the definition of *LVis* allows T_j to be arbitrarily excluded. In effect all three transactions are correct (so long as T_j does not eventually commit).

Overwriting

Fig. 5.11 shows an example of overwriting, which is not last-use opaque, since there is no equivalent sequential history where the write operation in T_i writing 1 to x would precede the read operation in T_j reading 1 from x without the other write operation writing 2 to x also preceding the read. Thus, in all cases T_j is not legal, and the history is neither final-state last-use opaque, nor last-use opaque.

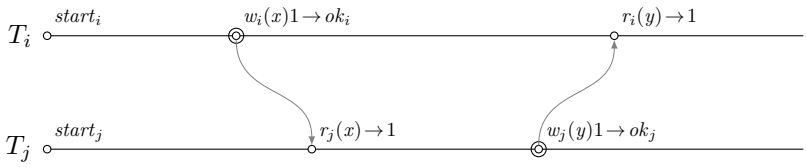


Figure 5.12: Dependency cycle—not last-use opaque.

Dependency Cycle

Finally, Fig. 5.12 shows an example of a cyclic dependency, where T_j reads x from T_i, and subsequently T_i reads y from T_j. Both writes in the history are closing writes. This example has unfinished transactions, which are thus aborted in any possible completion of this history. There are two possible sequential histories equivalent to that completion: one where T_i precedes T_j and one where T_j precedes T_i. In the former case, *LVis* of T_i does not contain any operations from T_j, because T_j follows T_i. Thus, there is no write operation on y preceding a read on y returning 1 in T_i's *LVis*, which does not conform to the sequential specification, so T_i's *LVis* is not legal. Hence, T_i is not legal in that scenario. The former case is analogous: T_j's *LVis* will not contain a write operation from T_i, because T_i follows T_j. Therefore T_j's *LVis* contains a read on x that returns 1, which is not preceded by any write on x, which causes the sequence not to conform to the sequential specification and renders the transaction not legal. Since either case contains a transaction that is not legal, then that history is not final-state last-use opaque, and therefore not last-use opaque.

5.1.4 Guarantees

Last-use opacity gives the programmer the following guarantees.

Serializability

If a transaction commits, then the value it reads can be explained by operations executed by preceding or concurrent transactions. This guarantees that a transaction that views inconsistent state will not commit.

Lemma 23 (Serializability)**.** *Every last-use opaque history is serializable.*

Proof. For the sake of contradiction let us assume that H is last-use opaque and not serializable. Since H is last-use opaque, then from Def. 24 H is also final-state last-use opaque. Then, from Def. 23 there exists a completion $H_C = \mathit{Compl}(H)$ such that there is a sequential history $\hat{S}_H$ s.t. $\hat{S}_H \equiv H_C$, $\hat{S}_H$ preserves the real-time order of H_C, and any committed transaction in $\hat{S}_H$ is legal in $\hat{S}_H$. However, since H is not serializable, then from Def. 7 there does not exist a completion $H_C = \mathit{Compl}(H)$ such that there is a sequential history $\hat{S}_H$ s.t. $\hat{S}_H \equiv H_C$, and any committed transaction in $\hat{S}_H$ is legal in $\hat{S}_H$. This contradicts the previous statement. □

Real-time Order

Successive transactions will not be rearranged to fit serializability, so a correct history will agree with an external clock, or an external order of events.

Lemma 24 (Real-time Order)**.** *Every last-use opaque history preserves real-time order.*

Proof. Trivially from Def. 24 and Def. 23a. □

Recoverability

If one transaction reads from another transaction, the former will commit only after the latter commits. This guarantees that transactions commit in order.

Lemma 25 (Recoverability)**.** *Every last-use opaque history is recoverable.*

Proof. Let us assume that H is not recoverable. Then there must be some transactions T_i and T_j s.t. T_j reads from T_i and then T_j commits before T_i. Such a history will contain a prefix P where any completion will contain an aborted T_i and a committed T_j, so for any equivalent sequential history $\hat{S}_H$ $\mathit{Vis}(\hat{S}_H, T_j)$ will not contain $\hat{S}_H|T_i$. Since T_j reads from T_i then such $\mathit{Vis}(\hat{S}_H, T_j)$ will not be legal, so by Def. 23 P is not last-use opaque and thus, by Def. 24, H is not last-use opaque, which is a contradiction. □

Last-use opacity does not preserve commitment order as defined in Def. 8, but we consider recoverability sufficient for TM. Note that strong properties like opacity also deal with commitment order only to the extent of recoverability.

Precluding Overwriting

If transaction T_i reads the value of some variable written by transaction T_j, then T_j will never subsequently modify that variable.

Lemma 26 (Precluding Overwriting)**.** *Last-use opacity does not support overwriting.*

Proof. For the sake of contradiction let us assume that there exists H that is a last-use opaque history with overwriting, i.e. (from Def. 5) there are transaction T_i and T_j s.t.:

a) T_i releases some variable x early,

b) $H|T_i$ contains $w_i(x)v \to ok_i$ and $w_i(x)v' \to ok_i$, s.t. the former precedes the latter in $H|T_i$,

c) $H|T_j$ contains $r_j(x) \to v$ that precedes $w_i(x)v' \to ok_i$ in H.

Since H is opaque, then there is a completion $C = \mathit{Compl}(H)$ and a sequential history S s.t. $S \equiv H$, S preserves the real-time order of H, and both T_i and T_j in S are legal in S. In S, either $T_i \prec_S T_j$ or $T_j \prec_S T_i$. In either case, any $\mathit{Vis}(S, T_j)$ or $\mathit{LVis}(S, T_j)$ by their definitions will contain either the sequence of both $w_i(x)v \to ok_i$ and $w_i(x)v' \to ok_i$ or neither of those write operation executions. In either case, $r_j(x) \to v$ will not be directly preceded by $w_i(x)v \to ok_i$ among operations on x in either $\mathit{Vis}(S, T_j)$ or $\mathit{LVis}(S, T_j)$. Therefore, T_j in S cannot be legal in S, which is a contradiction. □

Aborting Early Release

A transaction can release some variable early and subsequently abort.

Lemma 27 (Aborting Early Release)**.** *Last-use opacity supports aborting early release.*

Proof. Let H be the history depicted in Fig. 5.4. Here, T_i releases x early to T_j and subsequently aborts, which satisfies Def. 6. Since T_i and T_j are both aborted in H, H has a completion $C = \mathit{Compl}(H) = H$. Let S be a sequential history s.t. $S = H|T_i \cdot H|T_j$. S vacuously preserves the real-time order of H and trivially $S \equiv H$. Transaction T_i in S is last-use legal in S, because $\mathit{LVis}(S, T_i) = \overset{\diamond}{H|T_i}$—whose operations on x are limited to a single write operation execution—is within the sequential specification of x. Transaction T_j in S is also last-use legal in S, since $\mathit{LVis}(S, T_j) = \overset{\diamond}{H|T_i} \cdot \overset{\diamond}{H|T_j}$—whose operations on x consist of $w_i(x)v \to ok_i$ followed by $r_j(x) \to v$—is also within the sequential specification of x. Since both T_i and T_j in S are last-use legal in S, H is final-state last-use opaque. All

```
1 // invariant: x ⩾ 0
2 transaction {
3   x = y - 1
4   if (x < 0)
5     abort
6 }
```

(a) Abort example.

```
1 // invariant: x ⩾ 0
2 transaction {
3     *(_array + x);
4 }
```

(b) Memory error example.

Figure 5.13: Inconsistent view examples.

prefixes of H are trivially also final-state last-use opaque (since either their completion is the same as H's, they contain only a single write operation execution on x, or contain no operation executions on variables), so H is last-use opaque. □

Exclusive Access

Any transaction has effectively exclusive access to any variable it accesses, at minimum, from the first to the final modification it performs, regardless of whether it eventually commits or aborts.

Lemma 28 (Exclusive Access)**.** *Any transaction in any last-use opaque history has exclusive access to variable x between its first and last write to x.*

Proof. From Lemmas 23 and 26. □

5.1.5 Inconsistent Views

Last-use opacity does not preclude transactions from aborting after releasing a variable early. As a consequence there may be instances of cascading aborts, which have varying implications on consistency depending on whether the TM model allows transactions to abort programmatically. We distinguish three cases of models and discuss them below.

Commit-only Model

Let us assume that transactions cannot arbitrarily abort, but only do so as a result of receiving an abort response to invoking a read or write operation, or while attempting to commit. In other words, there is no *tryA* operation in the transactional API, as per the commit-only transactional model. In that case, since overwriting is not allowed, the transaction never reveals intermediate values of variables to other transactions. This means, that if a transaction released a variable early, then the programmer did not intend to change the value of that variable. So, if the transaction eventually committed, the value of the variable would have been the same. So, if the transaction is eventually forced to abort rather than committing, the value of any variable released early would be the same regardless of whether the transaction committed or aborted. Therefore, we can consider the inconsistent state to be safe. In other words, if the variable caused an error to occur, the error would be caused regardless of whether the transaction finally aborts or commits. Thus, we can say that with this set of assumptions, the programmer is guaranteed that none of the inconsistent views will cause unexpected behavior, even if cascading aborts are possible. Note that the use of this model is not uncommon (see eg. [28, 5, 6]).

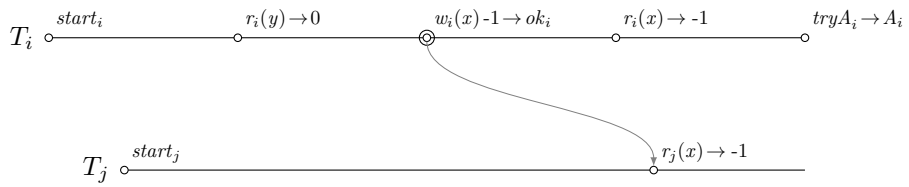


Figure 5.14: Last-use opaque history with inconsistent view.

Arbitrary Abort Model

Alternatively, let us assume that transactions can arbitrarily abort (in addition to forced aborts as described above) by executing the operation *tryA* as a result of some instruction in the program. In that case it is possible to imagine programs that use the abort instruction to cancel transaction due to the "business logic" of the program. Therefore a programmer explicitly specifies that the value of a variable is different depending on whether the transaction finally commits or not. An example of such a program is given in Fig. 5.13a. Here, the programmer enforced an invariant that the value of x should never be less than zero. If the invariant is not fulfilled, the transaction aborts. However, writing a value to x that breaks the invariant is the closing write operation execution for this program, so it is possible that another transaction reads the value of x before the transaction aborts. If the transaction that reads x is like the one in Fig. 5.13b, where x is used to index an array via pointer arithmetic, a memory error is possible. Nevertheless, the history from Fig. 5.14 that corresponds to a problematic execution of these two transactions is clearly allowed by last-use opacity (assuming that the domain of x is $\mathbb{Z}$). Thus, if the abort operation is available to the programmer the guarantee that inconsistent views will not lead to unexpected effects is lost. Therefore it is up to the programmer to use aborts wisely or to prevent inconsistent views from causing problems, by prechecking invariants at the outset of a transaction, or maintaining invariants also within a transaction (in a similar way as with monitor invariants). Alternatively, a mechanism can be built into the TM that prevents specific transactions at risk from reading variables that were released early, while other transactions are allowed to do so. However, if these workarounds are not satisfactory, we present a stronger variant of last-use opacity in Section 5.2 that deals specifically with this model and eliminates its inconsistent views.

Restricted Abort Model

We present a third alternative to aborts in transactions: a compromise between only forced aborts and programmer-initiated aborts. This option assumes that the *tryA* operation is not available to the programmer, so it cannot be used to implement business logic. However, we allow the TM system to somehow inject *tryA* operations in the code in response to external stimuli, such as crashes or exceptions and use aborts as a fault tolerance mechanism. However, since the programmer cannot use the operation, the programs must be coded as in the commit-only model, and therefore the same guarantees are given as in the commit-only model.

5.1.6 Strength

We compare the relative strength of last-use opacity with other properties from Chapter 3 and present the result of the comparison in Fig. 5.15. We provide the proofs for each comparison in Appendix A.

$\mathbb{H}_{\text{LU opacity}} \supset \mathbb{H}_{\text{opacity}}$
$\mathbb{H}_{\text{LU opacity}} \supset \mathbb{H}_{\text{markability}}$
$\mathbb{H}_{\text{LU opacity}} \supset \mathbb{H}_{\text{live opacity}}$
$\mathbb{H}_{\text{LU opacity}} \supset \mathbb{H}_{\text{rigorousness}}$
$\mathbb{H}_{\text{LU opacity}} \supset \mathbb{H}_{\text{TMS2}}$

$\mathbb{H}_{\text{LU opacity}} \parallel \mathbb{H}_{\text{elastic opacity}}$
$\mathbb{H}_{\text{LU opacity}} \parallel \mathbb{H}_{\text{VWC}}$
$\mathbb{H}_{\text{LU opacity}} \parallel \mathbb{H}_{\text{strictness}}$
$\mathbb{H}_{\text{LU opacity}} \parallel \mathbb{H}_{\text{cascadelessness}}$
$\mathbb{H}_{\text{LU opacity}} \parallel \mathbb{H}_{\text{TMS1}}$
$\mathbb{H}_{\text{LU opacity}} \parallel \mathbb{H}_{\text{CO}}$

$\mathbb{H}_{\text{LU opacity}} \subset \mathbb{H}_{\text{serializability}}$
$\mathbb{H}_{\text{LU opacity}} \subset \mathbb{H}_{\text{recoverability}}$

Figure 5.15: Strength of last-use opacity.

5.2 Strong Last-use Opacity

Even though last-use opacity prevents inconsistent views in the commit-only and restricted aborts models, it does not prevent inconsistent views in the arbitrary aborts model. Hence, we present a variant of last-use opacity called strong last-use opacity that extends the definition of a closing write operation to take *tryA* operations into account, as if it was an operation that modifies a given variable.

5.2.1 Intuition

Strong last-use opacity behaves in the same way as last-use opacity: it prevents transactions from reading from other live transactions, unless the transaction is guaranteed not to further modify the variable in question. The difference between last-use opacity and strong last-use opacity is that the latter considers aborts as operations that modify the variable as well as write operations, whereas last-use opacity considers only writes. Thus, strong last-use opacity defines its own variant of a closing write to be any write operation execution that is not followed by another write on the variable, nor any (voluntary) abort. In this way, transactions that can start a cascading abort are prevented from releasing early. This means that inconsistent views are excluded, while transactions with early release are prevented from aborting.

5.2.2 Definition

Below we define the concept of a *strongly closing write* to some variable by a particular transaction: we first define a strongly closing write operation invocation, and then extend the definition to complete operation executions.

Definition 25 (Strongly Closing Write Invocation)**.** *Given a program $\mathbb{P}$, a set of processes Π executing $\mathbb{P}$ and a history H s.t. $H \models \mathcal{E}(\mathbb{P}, \Pi)$, i.e. $H \in \mathbb{H}^{\mathbb{P},\Pi}$, an invocation $inv_i\big[w(x)v\big]$ is the* closing write invocation *on some variable x by transaction T_i in H, if for any history $H' \in \mathbb{H}^{\mathbb{P},\Pi}$ for which H is a prefix (i.e., $H' = H \cdot R$) there is no operation invocation inv' s.t. $inv_i\big[w(x)v\big]$ precedes inv' in $H'|T_i$ where (a) $inv' = inv_i\big[w(x)u\big]$ or (b) $inv' = inv_i\big[tryA\big]$.*

The remainder of the definitions of strong last-use opacity are formed by analogy to their counterparts in last-use opacity. Note that these definitions do not preclude some other operation than *tryA* returning A_i after a strongly closing write.

The definition of a strongly closing write operation execution is analogous to that of closing write operation execution Def. 21. The strongly closing write is used instead of the closing write to define a transaction *strongly decided on x* in analogy to Def. 22. Then, that definition is used to define $\hat{\mathbb{T}}^H_{\diamondsuit}$, $H\hat{|}_{\diamondsuit}T_j$, and $H\hat{|}^{\diamondsuit}T_j$ by analogy to $\hat{\mathbb{T}}^H$, $H\hat{|}T_j$ and

Fig.	Description	Last-use opaque	Strongly last-use opaque
5.2	Early release	✓	✓
5.3	Early release to aborting transaction	✓	✓
5.4	Early release with two aborting transactions	✓	×
5.5	Early release before closing write	×	×
5.6	Early release with two aborting transactions before closing write	×	×
5.7	Early release to a prematurely aborting transaction	✓	✓
5.8	Commit order not respected	×	×
5.9	Reversed commit order in writer transactions	✓	✓
5.10	Freedom to read or ignore an aborted transaction	✓	×
5.11	Early release with overwriting	×	×
5.12	Dependency cycle	×	×

Table 5.1: Histories satisfied by different variants of last-use opacity.

$H^{\circ}|T_j$. Next, those definitions are used to define *SLVis* by analogy to *LVis*. Finally, we say a transaction T_i is *strongly last-use legal* in some sequential history S if $SLVis(S, T_i)$ is legal. This allows us to define strong last-use opacity as follows.

Definition 26 (Final-state Strong Last-use Opacity)**.** *A finite history H is* final-state strongly last-use opaque *if, and only if, there exists a sequential history S equivalent to any completion of H s.t.,*

a) *S preserves the real-time order of H,*

b) *every transaction in S that is committed in S is legal in S,*

c) *every transaction in S that is not committed in S is strongly last-use legal in S.*

Definition 27 (Strong Last-use Opacity)**.** *A history H is* strongly last-use opaque *if, and only if, every finite prefix of H is final-state strongly last-use opaque.*

Theorem 2. *Strong last-use opacity is a safety property.*

Proof. By Def. 27, strong last-use opacity is trivially prefix-closed.

Given H_L that is an infinite limit of any sequence of finite histories $H_0, H_1, ...$, s.t every H_h in the sequence is strongly last-use opaque and every H_h is a prefix of H_{h+1}, since each prefix H_h of H_L is strongly last-use opaque, then, by extension, every prefix H_h of H_L is also final-state strongly last-use opaque, so, by Def. 27, H_L is strongly last-use opaque. Hence, strong last-use opacity is limit-closed.

Since strong last-use opacity is both prefix-closed and limit-closed, then, by Def. 1, it is a safety property. □

5.2.3 Examples

In Table 5.1 we show whether the examples in Fig. 5.2–5.12 satisfy strong last-use opacity alongside last-use opacity. Note that the properties allow and exclude histories in the same way except for two: Fig. 5.4 and 5.10. In those histories an aborting transaction releases a variable early, which means that each such transaction's last write was not, in fact, strongly closing, even though it was closing. This means that the write cannot be

$\mathbb{H}_{\text{SLU opacity}} \supset \mathbb{H}_{\text{opacity}}$
$\mathbb{H}_{\text{SLU opacity}} \supset \mathbb{H}_{\text{markability}}$
$\mathbb{H}_{\text{SLU opacity}} \supset \mathbb{H}_{\text{live opacity}}$
$\mathbb{H}_{\text{SLU opacity}} \supset \mathbb{H}_{\text{rigorousness}}$
$\mathbb{H}_{\text{SLU opacity}} \supset \mathbb{H}_{\text{TMS2}}$

$\mathbb{H}_{\text{SLU opacity}} \parallel \mathbb{H}_{\text{elastic opacity}}$
$\mathbb{H}_{\text{SLU opacity}} \parallel \mathbb{H}_{\text{VWC}}$
$\mathbb{H}_{\text{SLU opacity}} \parallel \mathbb{H}_{\text{strictness}}$
$\mathbb{H}_{\text{SLU opacity}} \parallel \mathbb{H}_{\text{cascadelessness}}$
$\mathbb{H}_{\text{SLU opacity}} \parallel \mathbb{H}_{\text{TMS1}}$
$\mathbb{H}_{\text{SLU opacity}} \parallel \mathbb{H}_{\text{CO}}$

$\mathbb{H}_{\text{SLU opacity}} \subset \mathbb{H}_{\text{serializability}}$
$\mathbb{H}_{\text{SLU opacity}} \subset \mathbb{H}_{\text{recoverability}}$
$\mathbb{H}_{\text{SLU opacity}} \subset \mathbb{H}_{\text{LU opacity}}$

Figure 5.16: Strength of strong last-use opacity.

included in *SLVis* of the other transaction that read from the aborting transaction. In effect, the reading transactions are not strongly legal, which causes the histories to fail to satisfy strong last-use opacity.

5.2.4 Guarantees

Strong last-use opacity gives most of the same guarantees as last-use opacity: serializability, real-time order, recoverability, precluding overwriting, aborting early release (in the case of forced aborts) and exclusive access. We forgo formal definitions and proofs of these, since they are analogous to those in Section 5.1.4.

5.2.5 Strength

We compare the relative strength of strong last-use opacity with last-use opacity and other properties from Chapter 3 and present the result of the comparison in Fig. 5.16. The proofs are analogous to those for last-use opacity. We also discuss how strong last-use opacity compares with last-use opacity in various abort models below.

In the commit-only model, the strong last-use opacity property is equivalent to last-use opacity. This is trivial, since if there there are no *tryA* operations in any history, then the definition of a strong closing write invocation is identical to the definition of a closing write invocation.

In the arbitrary abort model, strong last-use opacity property is strictly stronger than last-use opacity, because the definition of strong closing writes excludes histories that last-use opacity allows, including those with cascading aborts initiated by a voluntary abort.

In the restricted abort model, strong last-use opacity property is also strictly stronger than last-use opacity, but it is too strong to be applicable to systems with early release. In the first place, even though the histories that are excluded by strong last-use opacity contain inconsistent views, these are harmless, because as we argue in Section 5.1.5, transactions always release variables with "final" values. Since the *tryA* operation is not available to the programmer, these final values cannot be reverted by a programmer-initiated abort, so if the programmer sets up a closing write to a variable in a transaction, the value that was written was expected to both remain unchanged and be committed. Hence, it is acceptable for these values to be read by other transactions, even before the original transaction commits.

Finally, both in the restricted and the arbitrary abort models (but especially the former), if we assume that a TM system can inject a *tryA* operation into the transactional code to respond to some outside stimuli, such as crashes. Such events are unpredictable, so it may be possible for any transaction to abort at any time. Hence, it is necessary to assume that a *tryA* operation can be produced as the next operation invocation in any transaction at any time. In effect, as the definition of strong last-use opacity does not

Property	Application	Def. 4	Def. 5	Def. 6	⊆Serializable
Last-use opacity	TM	✓	×	✓	✓
Strong last-use opacity	TM	✓	×	✓	✓

Table 5.2: Summary of early release support in new properties: Def. 4 is early release support, Def. 5 is overwriting support, and Def. 6 is aborting early release support.

allow a transaction to release a variable early if a *tryA* is possible in the future, strong last-use opacity may prevent early release altogether in the restricted abort model.

In summary, strong last-use opacity is a useful variant of last-use opacity to exclude inconsistent views in the arbitrary abort model (if workarounds suggested in Section 5.1.5 are insufficient solutions). However strong last-use opacity may be too strict for TMs operating in the restricted and arbitrary abort models, where it may prevents early release altogether, depending on whether the injection of a *tryA* invocation into a transaction's code can be predicted or not. Certainly, in systems where aborts are used as a response to partial failures, strong last-use opacity prevents early release altogether. For that reason, we believe last-use opacity to be the more practical property.

5.3 Summary

In Table 5.2 we present a summary of the properties discussed in this chapter by analogy to the summary in Table 3.1. The table informs that a particular property is a TM safety property and whether it satisfies the definitions for early release support, overwriting support, and aborting early release support. Finally, the last column informs whether each property is at least as strong as serializability.

The table shows that both of the introduced properties allow early release without a requirement for transactions that release early not to abort. Nevertheless the properties are strong enough to prevent most inconsistent views and make others inconsequential. Specifically, neither property admits inconsistent views in the commit-only model and the compromise restricted abort model. Last-use opacity allow a relatively narrow class of inconsistent views in the arbitrary abort model, which can be mitigated by the programmer. On the other hand, the strong last-use opacity variant eliminates inconsistent views in all models, although does so for the price of preventing transactions that invoke the *tryA* operation to release any variable early. We consider strong last-use opacity and last-use opacity to be practical safety properties for TM systems that employ early release.

6
New Algorithms

In this chapter we present new pessimistic TM concurrency control algorithms designed with distributed systems and irrevocable operations in mind. Pessimistic TM is desirable in distributed systems, since aborts cause wasted effort on remote network nodes, introduce additional network traffic, and complicate the usage of non-transactional mechanisms within transactions, like network communication (outside the TM system). The goal of the algorithms is to achieve a high level of parallelism among conflicting transactions, as well as non-conflicting transactions, to match that of optimistic TM. The goal, however, should be achieved while maintaining the ability to execute transactions with irrevocable operations safely, i.e. without aborting or re-executing such operations.

We base the new algorithms on versioning algorithms presented in Section 4.1.2, with particular emphasis on SVA. We select these, because they are pessimistic, and so do not cause inconsistencies regarding irrevocable operations. In addition, SVA employs an early release mechanism that can be used to execute conflicting transactions partially in parallel, and potentially allow more concise histories than 2PL systems would for the same programs. They can also be implemented in a fully distributed fashion, with relatively minor adjustments.

Versioning algorithms have a specific set of limitations that we set out to overcome in the new algorithms. One drawback of versioning algorithms is that they use a global lock to acquire versions when transactions start. This introduces a single point of failure into the system, and limits the potential scalability of the system. The first section of the chapter introduces the problem in more detail and we provide variants of versioning algorithms that employ distributed locking schemes in place of the global lock.

Another drawback is that the versioning algorithms do not support an abort operation (in other words, they operate in the commit-only model). This limits their applicability in general, and specifically, makes them impractical for some classes of distributed systems, where faults need to be tolerated. We explain this problem in detail and provide versions of both BVA and SVA, which we call BVA+R and SVA+R, that operate in the more general arbitrary abort system model. The algorithms appeared in [74] and [75, 78], where they were implemented.

Finally, we tackle the biggest disadvantages versioning algorithms have. Since they were designed for web-service like architectures, where operations on remote objects have complex, and often unknown semantics, BVA+R and SVA+R (and their predecessors) operate under the assumption that the operations' semantics are unknowable, and that any operation may potentially conflict with any other operation on the same object.

Hence, they do not allow parallel executions of any type of operation on any type of object. However, this assumption is overstrict in certain classes of systems, like distributed data stores, where shared objects act like variables, and have known (and simple) operation semantics. In such a system model, SVA+R will perform relatively badly in comparison to other algorithms like 2PL, DTL2, or TFA, especially, if the ratio of read operations to write operations in executed transactions is high (see Chapter 8).

Hence, we introduce OptSVA+R, a new algorithm that builds on the versioning and early release mechanisms of SVA+R, but operates in the variable object model and recognizes different types of operations, and employs a number of optimizations to execute them in parallel to conflicting transactions. The approach taken, however, is different from the one used in TM algorithms like DTL2, MS-PTM, etc., since the optimizations do not require entire transactions to be read-only to trigger, but they can be employed on a variable-by-variable basis. In addition, OptSVA+R transactions use additional parallel threads to achieve local asynchrony, which means transactions effectively execute their own operations in parallel. We extend this result further by introducing OptSVA-CF+R, a variant of OptSVA+R that is intended for the homogeneous object system model, which better fits the distributed environment and the CF model than the variable model. The OptSVA+R and OptSVA-CF+R algorithms are novel and were introduced in [102, 82].

6.1 Distributed Version Acquisition

Before introducing new algorithms we present a modification we introduced to BVA and SVA (see Section 4.1.2) that applies to all versioning algorithms.

In versioning algorithms, when each transaction T_i starts, it is assigned a private version $\mathtt{pv}_i(\lceil x \rfloor)$ for each object $\lceil x \rfloor$ in its access set $\mathtt{ASet}_i$. A private version for $\lceil x \rfloor$ is generated from its global version $\mathtt{gv}(\lceil x \rfloor)$, which is initially 0, and is incremented with each starting transaction. In order to maintain the guarantees of private versions, this assignment must be done consistently. That is, the transaction must view a consistent snapshot of global versions for its entire access set, and must update global versions for its entire snapshot without interference. Hence, versioning algorithms specify that transactions use a global lock in order to generate and assign their private versions (see Fig. 4.10–4.11). Each transaction acquires the global lock at the beginning of the start procedure, and releases it at the end of the start procedure.

However, using a global lock to synchronize the start procedure is overstrict, since two transactions with disjoint access sets block each other from initializing. This limits parallelism, as the start procedure is executed in sequence for the entire TM system. If the algorithm is implemented in a non-distributed TM, the problem may not be evident, since computations performed within the start procedure are relatively lightweight. However, the problem is more pronounced when the algorithm is implemented in a distributed system. First, the assignment of a private version requires network communication, so executing the procedure is more costly. Thus, if a transaction waits for another to finish acquiring versions, the wait time is more likely to impact overall efficiency. Second, two transactions with disjoint access sets are likely executing operations on objects hosted on completely different network nodes. Making them block each other during the start procedure limits the overall scalability of the system. Finally, a global lock constitutes a possible bottleneck which will need to service all of the clients in the system. This has the dual drawbacks of introducing a single point of failure to the architecture, and detracting from the scalability of the distributed system.

```
1 proc start(Transaction T_i) {
2   for ⌈x⌋ ∈ ASet_i in order
3     lock lk(⌈x⌋) → W
4   for ⌈x⌋ ∈ ASet_i {
5     gv(⌈x⌋) ← gv(⌈x⌋) + 1
6     pv_i(⌈x⌋) ← gv(⌈x⌋)
7     unlock lk(⌈x⌋)
8   }
9 }
10 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
11   wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
12   execute m on ⌈x⌋ returning v
13   return v
14 }
15 proc commit(Transaction T_i) {
16   for ⌈x⌋ ∈ ASet_i {
17     wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
18     :dismiss(T_i, ⌈x⌋)
19   }
20   return C_i
21 }
22 proc :dismiss(Transaction T_i, Object ⌈x⌋) {
23   lv(⌈x⌋) ← pv_i(⌈x⌋)
24 }
```

(a) BVA.

```
1 proc start(Transaction T_i) {
2   for ⌈x⌋ ∈ ASet_i in order
3     lock lk(⌈x⌋) → W
4   for ⌈x⌋ ∈ ASet_i {
5     gv(⌈x⌋) ← gv(⌈x⌋) + 1
6     pv_i(⌈x⌋) ← gv(⌈x⌋)
7     unlock lk(⌈x⌋)
8   }
9 }
10 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
11   wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
12   execute m on ⌈x⌋ returning v
13   ac_i(⌈x⌋) ← ac_i(⌈x⌋) + 1
14   if ac_i(⌈x⌋) = supr_i(⌈x⌋)
15     :release(T_i, ⌈x⌋)
16   return v
17 }
18 proc commit(Transaction T_i) {
19   for ⌈x⌋ ∈ ASet_i {
20     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
21     :dismiss(T_i, ⌈x⌋)
22     ltv(⌈x⌋) ← pv_i(⌈x⌋)
23   }
24   return C_i
25 }
26 proc :release(Transaction T_i, Object ⌈x⌋) {
27   lv(⌈x⌋) ← pv_i(⌈x⌋)
28 }
29 proc :dismiss(Transaction T_i, Object ⌈x⌋) {
30   if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
31     lv(⌈x⌋) ← pv_i(⌈x⌋)
32 }
```

(b) SVA.

Figure 6.1: Versioning algorithms with FGL version acquisition.

Fine Grained Locking

Hence, we introduce a *fine-grained locking* (*FGL*) scheme for version acquisition. We substitute the global lock in versioning algorithms with a set of locks such that each lock $\texttt{lk}(\lceil x \rfloor)$ is associated with an individual shared object $\lceil x \rfloor$ (located so that $\texttt{location}(\lceil x \rfloor) = \texttt{location}(\texttt{lk}(\lceil x \rfloor))$). Then, given that transactions in versioning algorithms know their access sets *a priori*, during the start procedure each transaction only acquires locks for the objects in its access set. This allows transactions with disjoint access sets to start simultaneously. In order to prevent transactions from entering a deadlock, we eliminate circular waiting by imposing a global order of acquiring locks. The order can be whatever, as long as it is uniformly applied among transactions. In the remainder of the dissertation we assume one of the following locking orders apply: $\texttt{lk}(\lceil x \rfloor) < \texttt{lk}(\lceil y \rfloor) < \texttt{lk}(\lceil z \rfloor)$ or $\texttt{lk}(x) < \texttt{lk}(y) < \texttt{lk}(z)$. In contrast to the global lock solution where the lock is released once all the private versions are acquired, transactions can release $\texttt{lk}(\lceil x \rfloor)$ as soon as they acquire the private version for $\lceil x \rfloor$. In effect, the mechanism employs a simplified version of C2PL to acquire locks during transaction start. We show the pseudocode of BVA and SVA extended with this mechanism in Fig. 6.1. The optimization does not otherwise impact the execution of the algorithms.

In practice, the overhead of acquiring a lock for each object in the access set may be nevertheless prohibitive, and must be traded for the advantages it provides. In particular, a transaction acquiring a private version for $\lceil x \rfloor$ must send and receive additional messages to the remote host to acquire and release $\texttt{lk}(\lceil x \rfloor)$. For transactions with large access sets, this cost may become more apparent. Further, systems with low contention the cost may be greater than the cost of transactions competing for a global lock.

```
1 transaction {
2   execute withdraw(v) on ⌈x⌋
3   execute get_balance on ⌈x⌋ returning u
4   if (u < 0)
5     abort
6   else
7     execute deposit(v) on ⌈y⌋
8 }
```

(a) Abort.

```
1 transaction {
2   execute withdraw(v) on ⌈x⌋
3   execute get_balance on ⌈x⌋ returning u
4   if (u < 0)
5     execute deposit(v) on ⌈x⌋
6   else
7     execute deposit(v) on ⌈y⌋
8 }
```

(b) Compensation.

```
1 transaction {
2   execute withdraw(v) on ⌈x⌋
3   execute get_balance on ⌈x⌋ returning u
4   if (u < 0) {
5     execute deposit(v + u) on ⌈x⌋
6     execute cancel_loan on ⌈x⌋
7   } else
8     execute deposit(v) on ⌈y⌋
9 }
```

(c) Complex compensation.

```
1 transaction {
2   local_copy ← ⌈x⌋
3   execute withdraw(v) on ⌈x⌋
4   execute get_balance on ⌈x⌋ returning u
5   if (u < 0)
6     ⌈x⌋ ← local_copy
7   else
8     execute deposit(v) on ⌈y⌋
9 }
```

(d) Manual buffering.

Figure 6.2: Aborting transaction and manual counterparts.

Coarse Grained Locking

The number of messages sent during version acquisition may be reduced if the granularity of the locks is coarsened. Assuming objects are located on network nodes in groups, rather than locking individual objects, transactions can acquire a single lock for each node hosting objects. This reduces the potential parallelism, but requires fewer locks to be acquired on average. We refer to this variant as the *coarse-grained locking* (*CGL*) scheme, and we show an implementation of BVA and SVA employing this technique in Appendix B.

In the remainder of the dissertation we employ FGL version acquisition the presented algorithms, but note that a global lock or CGL version acquisition may be used in its place to tailor them to particular applications or workloads without impacting their correctness.

6.2 Versioning Algorithms in the Arbitrary Abort Model

The versioning concurrency control algorithms are pessimistic in nature, and do not need to abort any transaction to ensure correct execution.

Even so, a way to manually abort transactions is useful to the programmer, since this makes it easier to cancel a transaction mid-execution without having to manually scrub its effects. Consider the example transaction in Fig. 6.2a. The logic of the transaction is that of a bank transfer. The transaction attempts withdrawing some sum from the account represented by object $\lceil x \rfloor$ and deposit the same sum on the account represented by $\lceil y \rfloor$. However, the operation cannot proceed, if the balance of account $\lceil x \rfloor$ does not allow for it. Hence, after executing `withdraw` on $\lceil x \rfloor$, the transaction checks the balance of the account, and aborts it if it fell below 0. The abort operation erases the effects of the operation completely, and in this way it is very intuitive, as well as expressive.

If an abort operation were not available, the programmer would roll the transaction back manually. For instance, in Fig. 6.2b we show an example of how this can be achieved by compensation. If the balance of $\lceil x \rfloor$ falls below 0, the programmer simply puts the amount back into $\lceil x \rfloor$ and the transaction finishes. However, if this approach is to work in general, for every operation `m` in the the object's interface $\mathtt{M}_{\lceil x \rfloor}$ there must be an operation

```
1  proc start(Transaction T_i) {
2    for ⌈x⌋ ∈ ASet_i in order
3      lock lk(⌈x⌋) → W
4    for ⌈x⌋ ∈ ASet_i {
5      gv(⌈x⌋) ← gv(⌈x⌋) + 1
6      pv_i(⌈x⌋) ← gv(⌈x⌋)
7      unlock lk(⌈x⌋)
8    }
9  }
10 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
11   wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
12   if st_i(⌈x⌋) = ⊥
13     :checkpoint(T_i, ⌈x⌋)
14   execute m on ⌈x⌋ returning v
15   return v
16 }
17 proc commit(Transaction T_i) {
18   for ⌈x⌋ ∈ ASet_i {
19     wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
20     :dismiss(T_i, ⌈x⌋)
21   }
22   return C_i
23 }
24 proc abort(Transaction T_i) {
25   for ⌈x⌋ ∈ ASet_i {
26     wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
27     :dismiss(T_i, ⌈x⌋)
28     if st_i(⌈x⌋) ≠ ⊥
29       :recover(T_i, ⌈x⌋)
30   }
31   return A_i
32 }
33 proc :dismiss(Transaction T_i, Object ⌈x⌋) {
34   lv(⌈x⌋) ← pv_i(⌈x⌋)
35 }
36 proc :checkpoint(Transaction T_i, Object ⌈x⌋) {
37   st_i(⌈x⌋) ← ⌈x⌋
38 }
39 proc :recover(Transaction T_i, Object ⌈x⌋) {
40   ⌈x⌋ ← st_i(⌈x⌋)
41 }
```

Figure 6.3: BVA+R.

or a sequence of operations that compensate for $\lceil x \rfloor$. Furthermore, the programmer must know the semantics of `m` sufficiently well to compensate for it. Such semantics may not be obvious, for instance in Fig. 6.2c we show an example of a bank account object $\lceil x \rfloor$ that sets up a debt if the `withdraw` operation exceeds the balance. In such a case compensation might require more than simply depositing the amount back.

A more general solution shown in Fig. 6.2d would be to buffer the object before executing any of the operations within the transaction and restore it from a copy if the balance reaches 0. This is a manual implementation of the abort operation by the transaction. However, note that in a distributed system this requires that the object be copied across the network to a client. Assuming this is possible at all, the operation is likely to be expensive. If the operation were to be implemented on the server-side, sending the entire object *via* the network would not be necessary.

In addition to expressiveness, the implementation of such features as fault tolerance with respect to partial failures requires that transactions withdraw their effects and force the system into a consistent state. Since handling failures is, in practice, an unavoidable element of distributed systems, then transaction aborts become a mainstay of distributed TM, pessimistic or otherwise.

Hence, int this section we introduce variants of BVA and SVA that operate in the arbitrary abort model.

6.2.1 Basic Versioning Algorithm with Rollback

The *Basic Versioning Algorithm with Rollback* (*BVA+R*) is an extension of the BVA algorithm that allows it to operate in the arbitrary abort model. We give the complete pseudocode of the algorithm in Fig. 6.3 (with new mechanisms highlighted) and discuss it below.

The modification requires that transactions be provided an abort operation that implements *tryA*. In BVA, this operation is analogous to the commit procedure: transactions wait for each object in their access set to become available and release it, and finally the procedure returns the value indicating abort. However, since BVA is an encounter-time algorithm, the transaction might have modified each of the objects in its access set, so it needs to revert the state of the objects to some consistent state from before the transaction's modifications. Thus, whenever some transaction T_i executes an operation on object

```
1  proc start(Transaction T_i) {
2    for ⌈x⌋ ∈ ASet_i in order
3      lock lk(⌈x⌋) → W
4    for ⌈x⌋ ∈ ASet_i {
5      gv(⌈x⌋) ← gv(⌈x⌋) + 1
6      pv_i(⌈x⌋) ← gv(⌈x⌋)
7      unlock lk^g
8    }
9  }
10 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
11   wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
12   if st_i(⌈x⌋) = ⊥
13     :checkpoint(T_i, ⌈x⌋)
14   if ∃⌈y⌋ ∈ ASet_i: rv_i(⌈y⌋) > cv(⌈y⌋)
15     return abort(T_i)
16   execute m on ⌈x⌋ returning v
17   ac_i(⌈x⌋) ← ac_i(⌈x⌋) + 1
18   if ac_i(⌈x⌋) = supr_i(⌈x⌋)
19     :release(T_i, ⌈x⌋)
20   return v
21 }
22 proc commit(Transaction T_i) {
23   for ⌈x⌋ ∈ ASet_i {
24     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
25     :dismiss(T_i, ⌈x⌋)
26   }
27   if ∃⌈y⌋ ∈ ASet_i: rv_i(⌈y⌋) > cv(⌈y⌋)
28     return abort(T_i)
29   for ⌈x⌋ ∈ ASet_i
30     ltv(⌈x⌋) ← pv_i(⌈x⌋)
31   return C_i
32 }
33 proc abort(Transaction T_i) {
34   for ⌈x⌋ ∈ ASet_i {
35     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
36     :dismiss(T_i, ⌈x⌋)
37     if ac_i(⌈x⌋) ≠ 0 and rv_i(⌈x⌋) < cv(⌈x⌋)
38       :recover(T_i, ⌈x⌋)
39     ltv(⌈x⌋) ← pv_i(⌈x⌋)
40   }
41   return A_i
42 }
43 proc :dismiss(Transaction T_i, Object ⌈x⌋) {
44   if ac_i(⌈x⌋) = 0 and rv_i(⌈x⌋) = cv(⌈x⌋)
45     cv(⌈x⌋) ← pv_i(⌈x⌋)
46   if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
47     lv(⌈x⌋) ← pv_i(⌈x⌋)
48 }
49 proc :checkpoint(Transaction T_i, Object ⌈x⌋) {
50   st_i(⌈x⌋) ← ⌈x⌋
51   rv_i(⌈x⌋) ← cv(⌈x⌋)
52 }
53 proc :recover(Transaction T_i, Object ⌈x⌋) {
54   ⌈x⌋ ← st_i(⌈x⌋)
55   cv(⌈x⌋) ← rv_i(⌈x⌋)
56 }
57 proc :release(Transaction T_i, Object ⌈x⌋) {
58   cv(⌈x⌋) ← pv_i(⌈x⌋)
59   lv(⌈x⌋) ← pv_i(⌈x⌋)
60 }
```

Figure 6.4: SVA+R.

$\lceil x \rfloor$ for the first time, it executes the :checkpoint procedure which stores a copy $\lceil x \rfloor$ to buffer $\mathtt{st}_i(\lceil x \rfloor)$. Then, when the transaction is aborting, it uses the stored copy of the object to revert it to a consistent state. This is done by invoking the :recover procedure.

Nevertheless, since transactions retain exclusive access to every object in their access set between the first access to that object and either commit or abort, the introduction of the abort operation into the algorithm does not introduce inconsistent views. Hence, the extended BVA+R algorithm preserves the original's properties. Specifically, it is opaque, and free from deadlocks.

Note that we also moved the assignment of private versions at the end of commit (and also abort) to a separate procedure called :dismiss. We do this to indicate the semantics of that assignment—an unreleased object is released at the completion of a transaction. This is purely cosmetic at the moment and we do it for verisimilitude with algorithms that follow.

6.2.2 Supremum Versioning Algorithm with Rollback

Supremum Versioning Algorithm with Rollback (*SVA+R*) is an extension of SVA that allows it to operate in the arbitrary abort model. SVA+R was introduced in [74] (as SVA with rollback) and implemented in Atomic RMI [75, 78]. We give the full pseudocode of SVA+R in Fig. 6.4 and discuss the features of the new algorithm below.

Early Release

The main feature of SVA is that it employs an early release mechanism to execute concurrent transactions partially in parallel. The mechanism is illustrated in detail in Section 4.1.2 and does not differ in SVA+R, so we only remind it here briefly.

The early release mechanism triggers for a given object at a point in time after a transaction executes some operation on it and determines that it will execute no further

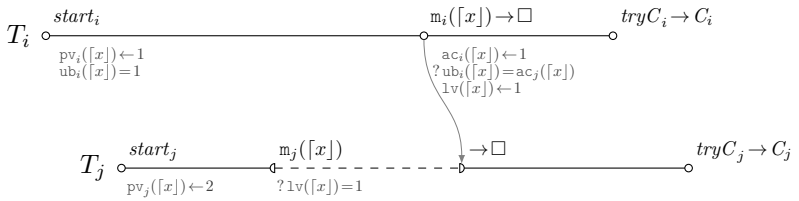


Figure 6.5: Early release via upper bounds (SVA+R).

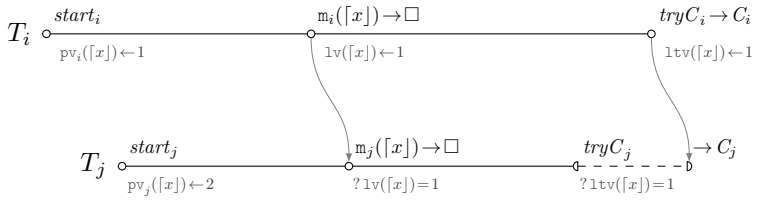


Figure 6.6: Commitment order preservation (SVA+R).

operations on that object in the future. This is determined based on *a priori* knowledge of each object's *supremum*, the maximum number of times it can be executed within a given transaction (e.g. as determined by analysis of a particular subprogram). The release is implemented by having the transaction write its private version to the object's local version counter at the point when the supremum was reached (via the release procedure), rather than waiting until the transaction commits (or aborts). When this is done, the transaction with the next consecutive private version number for the same object can satisfy the *access condition* and proceed to execute operations on the object (regardless of whether the first transaction committed yet or not). The early release mechanism is further illustrated in Fig. 6.5.

The early release feature is instrumental in increasing the degree of parallel execution among conflicting transactions, and, therefore, in making its implementation performant (see Section 8.1). However, the early release mechanism introduces additional complexity if transactions are allowed to abort in comparison to BVA+R. First of all, this design decision makes it necessary to enforce the order in which transactions commit to prevent a situation where transaction T_i releases $\lceil x \rfloor$ early and subsequently aborts, but before T_i does abort, T_j reads $\lceil x \rfloor$ and commits. That would mean that T_j committed having acted on an invalid, inconsistent value of $\lceil x \rfloor$, which is incorrect behavior (i.e., not serializable).

Commitment Order Preservation

In SVA+R we preclude this scenario by reflecting the order in which transactions access objects in the order in which they commit or abort. Each shared object has an associated *local terminal version* $\mathtt{ltv}(\lceil x \rfloor)$ which holds the private version of the transaction that either committed or aborted last. The local terminal version works by analogy to an object's local version, but transactions only write their private versions to the local terminal counter on commit on abort, but never when executing early release. Then, as the outset of executing the commit or abort procedure, each transaction T_j check whether $\mathtt{pv}_j(\lceil x \rfloor) - 1 = \mathtt{ltv}(\lceil x \rfloor)$, which we call the *commit condition*. If the condition is met, the transaction proceeds to commit or abort, and otherwise it must wait. In effect, by analogy to accessing objects, if transaction T_i accesses $\lceil x \rfloor$ before T_j, it is ensured that T_i commits or aborts before T_j. It means that the algorithm has the capability to forcibly abort any transaction that views inconsistent state.

We show an example of commitment order preservation in Fig. 6.6. Initially the local

version $\texttt{lv}(\lceil x \rfloor)$ is 0. Transaction T_i starts first and acquires the private version for $\lceil x \rfloor$ of 1. Then, transaction T_j starts and gets a private version for $\lceil x \rfloor$ equal to 2. Then, T_i attempts to execute an operation on $\lceil x \rfloor$. Since it can satisfy the access condition $\texttt{pv}_i(\lceil x \rfloor) - 1 = \texttt{lv}(\lceil x \rfloor)$, the access goes through. In addition, T_i determines that this was the last operation execution on $\lceil x \rfloor$, so T_i releases $\lceil x \rfloor$. This means that, when transaction T_j attempts to access $\lceil x \rfloor$ soon after it also satisfies the access condition and the operation executes. Afterward, T_j attempts to commit, but in order to do so, it must pass the commit condition $\texttt{pv}_j(\lceil x \rfloor) - 1 = \texttt{ltv}(\lceil x \rfloor)$, but it cannot do so until T_i commits (or aborts) and sets $\texttt{ltv}(\lceil x \rfloor)$ to its $\texttt{pv}_i(\lceil x \rfloor)$ of 1. Hence, T_j waits with its commit operation until other transactions that accessed $\lceil x \rfloor$ before it also commit (or abort).

Cascading Aborts

Furthermore, if some transaction potentially views the state of an object that was modified by another transaction, and the latter aborts, then the former cannot be allowed to commit having possibly acted upon inconsistent state. Hence, the transaction must be forced to abort.

To enforce aborts, SVA+R marks which version of an object is the most recent consistent version via a *current version* counter $\texttt{cv}(\lceil x \rfloor)$ shared by all transactions. This is used in conjunction with each transaction T_i's own *recovery version* $\texttt{rv}_i(\lceil x \rfloor)$, which indicates the last consistent version viewed by T_i. The current version can be compared to a transaction's recovery version to check whether the transaction is using a consistent (current) version of each object or not. Unlike other counters which are initially set to 0, rv is initially set to $-\infty$.

Specifically, whenever transaction T_i gains access to shared object $\lceil x \rfloor$ for the first time, it runs the :checkpoint procedure which buffers $\lceil x \rfloor$ and stores it in its buffer $\texttt{st}_i(\lceil x \rfloor)$ (just like in BVA+R). In addition, the :checkpoint procedure sets its recovery version $\texttt{rv}_i(\lceil x \rfloor)$ to $\lceil x \rfloor$'s current version $\texttt{cv}(\lceil x \rfloor)$.

The value of the current version for some object reflects the private version for this object of such transaction that most recently committed or released this object early. Thus, each transaction sets the current version for each object in its access set during commit, and for specific objects during early release of those objects. A transaction which aborts also updates the current version for each object in its access set while aborting, however the value written to the current version is not the transactions private version, but its recovery version—the last consistent version of the object that this transaction viewed. Initially, current version counters for all variables are set to 0.

When transaction T_i updates the current version of some object $\lceil x \rfloor$ during abort (via the dismiss procedure), the new value will be lower than that transaction's private version $\texttt{pv}_i(\lceil x \rfloor)$. However, if T_i released that variable early, it would have previously set the current version of the variable to be equal to its private version. Thus, if any other transaction T_j accessed $\lceil x \rfloor$ between T_i released it and aborted, it would have acquired a recovery version $\texttt{rv}_j(\lceil x \rfloor)$ equal to $\texttt{cv}(\lceil x \rfloor)$, which was then equal to $\texttt{pv}_i(\lceil x \rfloor)$. However once T_i aborted, $\texttt{cv}(\lceil x \rfloor)$ would become lower than $\texttt{pv}_i(\lceil x \rfloor)$, so lower than $\texttt{rv}_j(\lceil x \rfloor)$.

Thus, a difference between $\texttt{rv}_j(\lceil x \rfloor)$ and $\texttt{cv}(\lceil x \rfloor)$ where $\texttt{rv}_j(\lceil x \rfloor) > \texttt{cv}(\lceil x \rfloor)$ signals to T_j that the version of $\lceil x \rfloor$ it accesses has become invalid, because some previous transaction aborted and reverted it. Therefore, whenever a transaction tries to commit or access a shared object, it must test the consistency of the object it operates on by verifying that the current version of the object was not reverted to a lower value than the transaction's recovery version. If that is the case, the transaction is forced to abort instead of executing the originally intended access or the commit operation. During variable access, the condition for aborting is always checked for all objects in the access

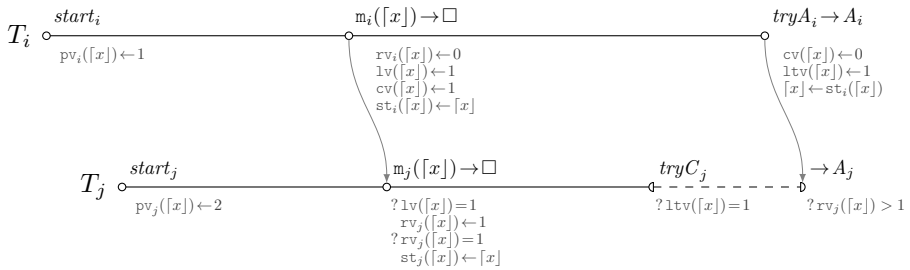


Figure 6.7: Forced abort (SVA+R).

set rather than just the one being accessed, in order to abort as quickly as possible, and to prevent the transaction from operating on both consistent and invalidated variables simultaneously.

Note that if the difference between $\mathtt{rv}_j(\lceil x \rfloor)$ and $\mathtt{cv}(\lceil x \rfloor)$ but $\mathtt{rv}_j(\lceil x \rfloor) < \mathtt{cv}(\lceil x \rfloor)$, transaction T_j does not need to abort, since that difference signifies that T_j released $\lceil x \rfloor$. The condition is also true if some other transaction whose private version for $\lceil x \rfloor$ is higher than T_j's released it. It could also be true that some of these transactions aborted and reverted $\lceil x \rfloor$, but this does not impact T_j nor force it to abort, since T_j will have released $\lceil x \rfloor$ by that time and (from its perspective) future invalid states are irrelevant.

We show an example of this in Fig. 6.7. Here, T_i and T_j access $\lceil x \rfloor$ and have private values for $\lceil x \rfloor$ equal to 1 and 2, respectively. Hence T_i accesses $\lceil x \rfloor$ first. As this is executed T_i sets its recovery version to 0, the value of the current version for $\lceil x \rfloor$. Then, after the operation on $\lceil x \rfloor$ finishes executing, T_i releases $\lceil x \rfloor$ by setting the local version to 1 and sets the current version $\mathtt{cv}(\lceil x \rfloor)$ to its own private version, i.e. 1. Subsequently T_j meets the access condition and accesses $\lceil x \rfloor$ for the first time, setting its own recovery version to 1 (as $\mathtt{cv}(\lceil x \rfloor) = 1$). Since $\mathtt{rv}_j(\lceil x \rfloor) = \mathtt{cv}(\lceil x \rfloor)$, the access is successful. However, as T_j tries to commit, it is delayed because it cannot satisfy the commit condition. Meanwhile transaction T_i aborts. As it does so, it sets the current version to its recovery version equal to 0. Then, T_i sets the local terminal version to its own private version, allowing T_j to resume committing. However, T_j satisfies the condition $\mathtt{rv}_j(\lceil x \rfloor) > \mathtt{cv}(\lceil x \rfloor)$ during commit, since $\mathtt{rv}_j(\lceil x \rfloor) = 1$ and $\mathtt{cv}(\lceil x \rfloor) = 0$, so T_j is forced to abort.

Inconsistent Views

While lifting BVA to its arbitrary abort variant BVA+R does not have consequences for safety, lifting SVA to SVA+R does. This is because, unlike BVA/BVA+R, SVA/SVA+R use the early release mechanism, which, by allowing transactions to view potential modifications done to objects by transactions that are still live, admits potential inconsistent views. Originally, in SVA those potential inconsistent states do not result in transactions seeing incorrect states of shared objects, because there are no aborting transactions. However, once transactions are allowed to abort, other transactions can view modifications introduced by transactions that will later abort, which can result in some dangerous situations (see Section 5.1.1).

Specifically, in SVA+R inconsistent views are limited to a particular situation. First, there must be some transaction that T_i executes some operation on $\lceil x \rfloor$, releases $\lceil x \rfloor$ early (after last write) and subsequently aborts. In that case, some transaction T_j can view an inconsistent state of $\lceil x \rfloor$ if where, given two transactions T_i and T_i that both access $\lceil x \rfloor$, transaction T_i executes some operation on $\lceil x \rfloor$, releases $\lceil x \rfloor$ early (after last write) and subsequently aborts, while T_j executes some operation on $\lceil x \rfloor$ after T_i releases $\lceil x \rfloor$ but before T_i aborts. In the next section we discuss a solution that eliminates this scenario for selected transactions, causing them never to be forced to abort.

```
1  proc start(Transaction T_i) {
2    for ⌈x⌋ ∈ ASet_i in order
3      lock lk(⌈x⌋) → W
4    for ⌈x⌋ ∈ ASet_i {
5      gv(⌈x⌋) ← gv(⌈x⌋) + 1
6      pv_i(⌈x⌋) ← gv(⌈x⌋)
7      unlock lk(x)
8    }
9  }
10 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
11   if T_i ∈ ℝ
12     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
13   else
14     wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
15   if st_i(⌈x⌋) = ⊥
16     :checkpoint(T_i, ⌈x⌋)
17   execute m on ⌈x⌋ returning v
18   ac_i(⌈x⌋) ← ac_i(⌈x⌋) + 1
19   if ac_i(⌈x⌋) = supr_i(⌈x⌋)
20     :release(T_i, ⌈x⌋)
21   return v
22 }
23 proc commit(Transaction T_i) {
24   for ⌈x⌋ ∈ ASet_i {
25     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
26     :dismiss(T_i, ⌈x⌋)
27   }
28   if ∃⌈y⌋ ∈ ASet_i: rv_i(⌈y⌋) > cv(⌈y⌋)
29     return abort(T_i)
30   for ⌈x⌋ ∈ ASet_i
31     ltv(⌈x⌋) ← pv_i(⌈x⌋)
32   return C_i
33 }
34 proc abort(Transaction T_i) {
35   for ⌈x⌋ ∈ ASet_i {
36     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
37     :dismiss(T_i, ⌈x⌋)
38     if ac_i(⌈x⌋) ≠ 0 and rv_i(⌈x⌋) < cv(⌈x⌋)
39       :recover(T_i, ⌈x⌋)
40     ltv(⌈x⌋) ← pv_i(⌈x⌋)
41   }
42   return A_i
43 }
44 proc :dismiss(Transaction T_i, Object ⌈x⌋) {
45   if ac_i(⌈x⌋) = 0 and rv_i(⌈x⌋) = cv(⌈x⌋)
46     cv(⌈x⌋) ← pv_i(⌈x⌋)
47   if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
48     lv(⌈x⌋) ← pv_i(⌈x⌋)
49 }
50 proc :checkpoint(Transaction T_i, Object ⌈x⌋) {
51   st_i(⌈x⌋) ← ⌈x⌋
52   rv_i(⌈x⌋) ← cv(⌈x⌋)
53 }
54 proc :recover(Transaction T_i, Object ⌈x⌋) {
55   ⌈x⌋ ← st_i(⌈x⌋)
56   cv(⌈x⌋) ← rv_i(⌈x⌋)
57 }
58 proc :release(Transaction T_i, Object ⌈x⌋) {
59   cv(⌈x⌋) ← pv_i(⌈x⌋)
60   lv(⌈x⌋) ← pv_i(⌈x⌋)
61 }
```

Figure 6.8: RSVA+R.

Because inconsistent views are limited to the aforementioned cases SVA+R is last-use opaque—we demonstrate and discuss this in detail in Section 7.2. Since it arranges both accesses and commits in the order dictated by private versions it trivially preserves commitment order. Like all versioning algorithms, SVA+R is deadlock-free by nature of concurrency control through versioning.

Reluctant Transactions

Given that a cascading abort may occur, SVA+R may be forced to abort transactions that contain irrevocable operations. If the transaction aborted voluntarily, then the programmer takes responsibility. However, it may happen that a transaction with irrevocable operations is forced to abort in a cascading abort scenario.

To solve this problem we introduce a variant of RSVA+R called *Reluctant SVA+R* (or just *RSVA+R*). In RSVA+R we assume there to be a class of *reluctant* transactions $\mathbb{R}$, a subclass of all transactions $\mathbb{T}$ that use a more conservative access condition, which prevents them from being drawn into a cascading abort. Specifically, reluctant transactions refuse to access objects that were potentially modified by uncommitted transactions by waiting until the transaction that modified it most recently commits. That is when executing a method on some object x, reluctant transaction T_i checks the commit condition $\mathrm{pv}_i(\lceil x \rfloor) - 1 = \mathtt{ltv}(\lceil x \rfloor)$ rather than the access condition $\mathrm{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$. Hence, by the time T_i executes its method on $\lceil x \rfloor$, $\lceil x \rfloor$ is definitely consistent, and, since inconsistent views are the only source of forced aborts, then T_i is never forced to abort. Thus, unless T_i aborts voluntarily, it is irrevocable.

We give RSVA+R's pseudocode in Fig. 6.8 with the changed access condition highlighted. In order for the condition to operate, the transaction must know (*a priori*) whether it is reluctant. Reluctant transactions can be designated manually, or detected via automatic means, like static analysis to find irrevocable operations within.

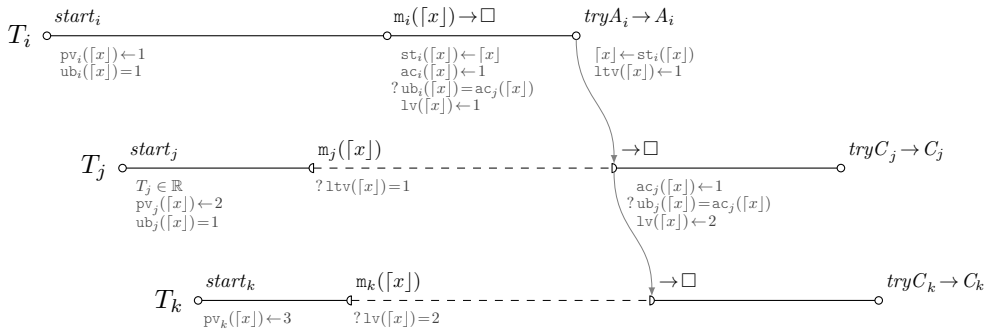


Figure 6.9: Early release with a reluctant transaction (RSVA+R).

The solution trades parallelism for safety of irrevocable operations, which may be deferred for a longer time than they would have been otherwise, but does so without limiting the early release capability of ordinary transactions. As such, it is a compromise between SVA+R and BVA+R and if $\mathbb{R} = \mathbb{T}$, then all transactions wait for preceding transactions to commit, so RSVA+R behaves exactly like BVA+R. However, if the set of reluctant transactions is limited, the solution retains a reasonable level of possible parallelism, since, nothing stops reluctant transactions from releasing early themselves. This contrasts our solution from algorithms like PLE and MS-PTM (Section 4.3.1–4.3.2) and irrevocable transactions in [92], where irrevocable transactions execute sequentially, for the most part.

We show an example of a reluctant transaction executing in Fig. 6.9. The history is analogous to Fig. 6.7, but here T_j is reluctant, meaning that when T_i releases early, T_j still waits, and only starts performing its operation in earnest when T_i aborts. This means that T_j does not execute operations on the inconsistent state of x, so it does not need to abort. Meanwhile, T_j can release $\lceil x \rfloor$ early after its operation is executed, meaning the next transaction (non-reluctant) T_k can benefit from improved concurrency.

RSVA+R retains all of the properties of SVA+R.

6.3 Optimized Supremum Versioning Algorithm

The versioning algorithms presented thus far all operate in the heterogeneous object model where they assume that the semantics of a particular operation executed on each object cannot be known. Hence, these algorithms treat all operations uniformly and conservatively: each operation is treated as if it both potentially views and modifies the object it is executed on. This approach applies to particular distributed application like web service orchestration, where transactional memory's shared objects are entire stateful services with complex and heterogeneous interfaces, sometimes dynamically changing interfaces among which the TM algorithm must maintain consistency. In systems as these treating all operations the same is a practical general approach.

On the other hand, in systems like distributed data stores, as well just non-distributed TM, where the homogeneous object model and the variable model apply, versioning algorithms are not capable of as much parallelism as other TM algorithms. The reason for this is that versioning algorithms do not take advantage of known semantics of operations, so they assume potential conflicts where they do not occur in practice. Hence, for instance, two read-only transactions will block each other, which is not true for most TM algorithms.

In this section we describe the *Optimized Supremum Versioning Algorithm with Rollback* (*OptSVA+R*) introduced in [102], a variant of SVA+R that operates in the variable model and takes advantage of the known semantics of operations to introduce a number of far-reaching optimizations that aim to eliminate its predecessors limitations and improve the degree to which conflicting transactions execute in parallel, and, in effect, efficiency of execution. Specifically, OptSVA+R uses buffering in order to make local operations invisible outside the transaction they are executed in. This allows to OptSVA+R transactions to expedite early release, which happens after the last non-local write operation on some variable (rather than once all operations are executed). In addition, OptSVA+R transactions defer the moment of checking the accesses condition to a given variable to the first non-local read operation or the last non-local write operation.

Furthermore, OptSVA+R delegate specific concurrency-control–related tasks to separate threads to achieve transaction-local asynchrony. That is, when a transaction has to wait for the access condition or the commit condition to be satisfied, it can delegate the waiting to a separate thread, and perform other computations in the meantime. This allows a transaction to perform local computations and non-conflicting operations while waiting to serialize conflicting operations with other transactions. This feature is especially valuable in distributed systems, where network communication introduces delays.

OptSVA+R is specified in full in Fig. 6.10 and we describe it in detail below. Given that the versioning concurrency control, commitment ordering, and forced abort portions of the algorithm are inherited from SVA+R, we do not discuss them again, instead focusing on the novel optimizations OptSVA+R introduces. These use the combination of the explicit distinction between read and write operations, buffering, and asynchronous execution of specific synchronization-related tasks to optimize accesses to read-only variables, to delay synchronization of the initial operation upon an initial write, and to expedite early release to the last (closing) write. We then summarize the entire algorithm.

Further, we show through formal analysis that OptSVA+R can produce tighter interleavings than SVA+R due to the increased level of parallelism, and, therefore, is more likely to produce tighter schedules than its predecessor.

Finally, we present two variants of OptSVA+R: one that allows a class of reluctant transactions which are never forced to abort, and one that precludes voluntary aborts. These variants trade parallelism or generality in return for precluding inconsistent views.

6.3.1 Read-only Variables

Since originally versioning algorithms did not distinguish between reads and writes, they did not allow read-only transactions to be executed in parallel to other read-only transactions. This is a run-of-the-mill optimization found in all but a small number of TMs, so it is also introduced in OptSVA+R. However, OptSVA+R goes a step further, and allows partial parallelization of transactions whenever a variable in a transaction is only read from and not written to, without requiring that all the variables in a transaction are not written to.

First, while OptSVA+R inherits the early release mechanism based on *a priori* knowledge, the information provided *a priori* is different. Each transaction still knows the suprema for each variable, but unlike in SVA+R, the suprema are divided into the maximum number of times the transaction will respectively read and write each individual variable. These suprema for reads and writes are denoted for transaction T_i and x as, respectively, $\mathtt{rub}_i(x)$ and $\mathtt{wub}_i(x)$.

Whenever transaction T_i accesses x in such a way that it reads from x but does not write to x (i.e., $\mathtt{rub}_i(x) > 0$ and $\mathtt{wub}_i(x) = 0$), we will refer to x as being a *read-only*

```
1  proc start(Transaction T_i) {
2    // Acquire private versions.
3    for x ∈ ASet_i in order
4      lock lk(x) → W
5    for x ∈ ASet_i {
6      gv(x) ← gv(x) + 1
7      pv_i(x) ← gv(x)
8      unlock lk(x)
9    }
10   // Asynchronously buffer read−only variables.
11   for x ∈ ASet_i: wub_i(x) = 0
12     async run :read_buffer(T_i, x)
13       when pv_i(x) - 1 = lv(x)
14   return ok_i
15 }
16 proc read(Transaction T_i, Variable x) {
17   // Wait for read−only variable to be bufferred.
18   if wub_i(x) = 0
19     join with :read_buffer(T_i, x)
20   // Copy value of variable to buffer on first read.
21   else if wc_i(x) = 0 and rc_i(x) = 0 {
22     wait until pv_i(x) - 1 = lv(x)
23     :checkpoint(T_i,x)
24     buf_i(x) ← st_i(x)
25   }
26   // Abort on inconsistent view.
27   if ∃y: rv_i(y) ≠ cv(y)
28     return abort(T_i)
29   rc_i(x) ← rc_i(x) + 1
30   // Return bufferred value.
31   return buf_i(x)
32 }
33 proc write(Transaction T_i, Variable x, Value v) {
34   // Abort on inconsistent view.
35   if ∃y: rv_i(y) ≠ cv(y)
36     return abort(T_i)
37   // Write to buffer.
38   buf_i(x) ← v
39   wc_i(x) ← wc_i(x) + 1
40   // Asynchronous release on last write.
41   if wc_i(x) = wub_i(x)
42     async run :write_buffer(T_i,x)
43       when pv_i(x) - 1 = lv(x)
44   return ok_i
45 }
46 proc :read_buffer(Transaction T_i, Variable x) {
47   // Buffer and release a read−only variable.
48   rv_i(x) ← cv(x)
49   buf_i(x) ← x
50   :release(T_i,x)
51   async run :read_commit(T_i, x)
52     when pv_i(x) - 1 = ltv(x)
53 }
54 proc :read_commit(Transaction T_i, Variable x) {
55   // Commit a read−only variable early.
56   if ∃y: rv_i(y) > cv(y)
57     return abort(T_i)
58   ltv(x) ← pv_i(x)
59 }
60 proc :write_buffer(Transaction T_i, Variable x) {
61   if st_i(x) = ⊥
62     :checkpoint(T_i,x)
63   if ∃y: rv_i(y) ≠ cv(y)
64     return abort(T_i)
65   x ← buf_i(x)
66   :release(T_i,x)
67 }
68 proc commit(Transaction T_i) {
69   for x ∈ ASet_i {
70     // Synchronize with extant read thread.
71     if wub_i(x) = 0
72       join with :read_comit(T_i,x)
73     else {
74       // If released, synchronize with write thread.
75       if wc_i(x) = wub_i(x)
76         join with :write_buffer(T_i,x)
77       else {
78         // Catch up: get access and update variable.
79         wait until pv_i(x) - 1 = lv(x)
80         if st_i(x) = ⊥
81           :checkpoint(T_i, x)
82         if ∃y: rv_i(y) ≠ cv(y)
83           return abort(T_i)
84         if wc_i(x) > 0
85           x ← buf_i(x)
86       }
87       // Maintain commitment order.
88       wait until pv_i(x) - 1 = ltv(x)
89       :dismiss(T, x)
90     }
91   }
92   // Abort on inconsistent view.
93   if ∃y: rv_i(y) > cv(y)
94     return abort(T_i)
95   for x ∈ ASet_i
96     ltv(x) ← pv_i(x)
97   return C_i
98 }
99 proc abort(Transaction T_i) {
100  for(x ∈ ASet_i) {
101    // Maintain commitment order.
102    wait until pv_i(x) - 1 = ltv(x)
103    // Restore if consistent backup and modified.
104    if (wc_i(x) > 0 and pv_i(x) - 1 > lv(x)
105        and rv_i(x) = cv(x)) {
106      if wc_i(x) = wub_i(x)
107        join with :write_buffer(T_i,x)
108      :recover(T_i, x)
109    }
110    :dismiss(T_i, x)
111    ltv(x) ← pv_i(x)
112  }
113  return A_i
114 }
115 proc :dismiss(Transaction T_i, Variable x) {
116   if pv_i(x) - 1 = lv(x)
117     lv(x) ← pv_i(x)
118   if (wc_i(x) + rc_i(x)> 0 and rv_i(x) = cv(x)
119       and pv_i(x) - 1 > lv(x))
120     cv(x) ← pv_i(x)
121 }
122 proc :checkpoint(Transaction T_i, Variable x) {
123   st_i(x) ← x
124   rv_i(x) ← cv(x)
125 }
126 proc :recover(Transaction T_i, Variable x) {
127   x ← st_i(x)
128   cv(x) ← rv_i(x)
129 }
130 proc :release(Transaction T_i, Variable x) {
131   cv(x) ← pv_i(x)
132   lv(x) ← pv_i(x)
133 }
```

Figure 6.10: OptSVA+R.

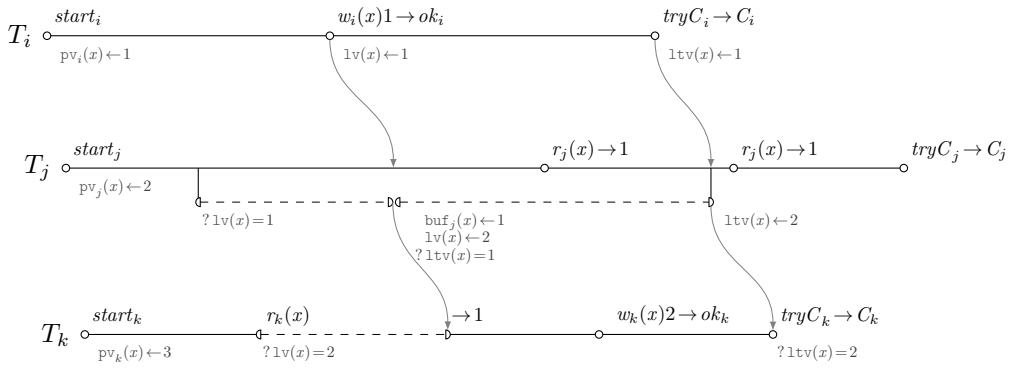


Figure 6.11: Read-only variable optimization (OptSVA+R).

variable in T_i. In the case of such variables, OptSVA+R can optimize the accesses by buffering the variable and reading the buffer instead of the actual variable. In addition, since all the reads will be done using the buffer, and the upper bounds indicate that no writes will follow, the variable can be released after it is buffered, irrespective of what operations the transaction will execute later.

Obviously, it is best for parallelism to release any variable as soon as it is no longer needed by a transaction, because it allows other transactions to start acting sooner. Since read-only variables are not needed after they are buffered, they can be released immediately after this happens. The variable must be buffered before or during the first read operation on it is executed, but it could be buffered before that point, even during transaction start. However, in order to buffer a variable, its state must be viewed, so, for the sake of consistency, buffering within versioning concurrency control must be done only after the transaction passes the access condition. Since waiting at the access condition would prevent the transaction from executing operations on other variables or performing local computations, it is best for parallelism for the transaction not to start waiting until it is absolutely necessary.

The algorithm finds balance between buffering as soon as possible and delaying synchronization much as necessary by executing it asynchronously. This is achieved by using the **`async run`** `P` **`when`** `C` construct which relegates the execution of procedure `P` to some separate thread. However, before the thread starts executing `P` it waits until condition `C` is satisfied. This allows the transaction to wait at condition `C` without preventing the procedure from delaying other operations that could be executing in the mean time. On the other hand, `P` will be executed as soon as `C` is satisfied, so as soon as it is safe.

OptSVA executes buffering via procedure `:read_buffer`. This procedure is relegated to asynchronous execution at lines 12–13, and will execute once the access condition is satisfied. Within the `:read_buffer` procedure, the transaction T_i saves the value of some variable x to its buffer $\mathtt{buf}_i(x)$ (line 49), and releases it immediately afterward by executing `:release` (line 50). Since it is possible that the transaction that wrote the value of x that is being buffered will subsequently abort, T_i also updates its recovery value (line 48), but it does not need to make a checkpoint for x, since the transaction will not modify x. Once read-only variable x is buffered, read operations can use the buffer to retrieve that value, without accessing the variable (line 31), so without waiting. However, a read on a read-only variable cannot be executed until buffering is finished (line 19), which we indicate using the **`join with`** `P` construct.

Since a transaction does not modify a read-only variable, if it aborts, it does not need to force other transactions to abort to maintain consistency. Hence, the transaction tries to immediately perform all commit-related operations for a read-only variable immedi-

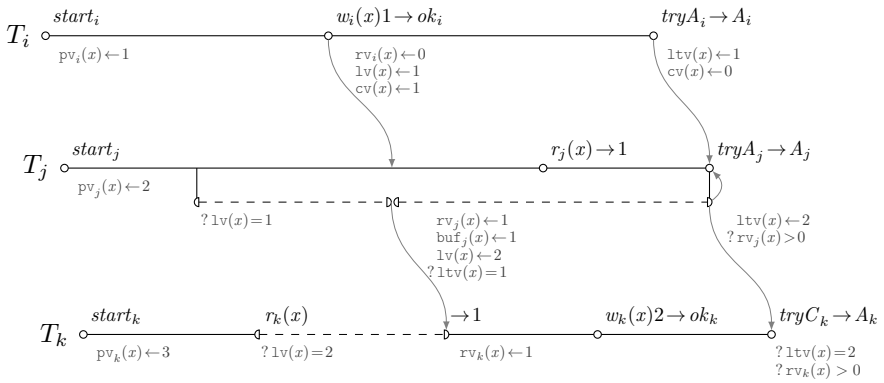


Figure 6.12: Aborts and read-only variable optimization (OptSVA+R).

ately after buffering it. This involves waiting for the local terminal version of the object, so by analogy to buffering, the procedure is executed asynchronously, so as not to block other operations. The procedure that executes the commit for variable x is :`read_commit` and it is started asynchronously at line 52. The procedure executes a simplified version of `commit` for just x. Hence, once `commit` is executed by the transaction for other variables, it can be skipped for x, and the transaction simply waits for :`read_commit` to finish executing.

We show an example of an execution of a transaction T_j with a read-only variable x in Fig. 6.11. Transaction T_j asynchronously waits for the access condition on x to be met right after T_j starts, but before any reads actually occur. A parallel line below transaction (such as the one below T_j) indicates procedures executed asynchronously with respect to the thread executing the transaction. Meanwhile T_j can perform local operations or operations on other variables without obstacle. Once T_i releases x, T_j immediately buffers x, and releases it. Then T_j asynchronously tries to commit x, which requires that it waits for the appropriate local terminal version of x. Meanwhile T_k can now access x in parallel to T_j and even write to it, without interfering with T_j's consistency. Once T_i commits, T_j can then asynchronously commit x, which then allows T_k to commit earlier than it would have otherwise. Since T_j treats x as read-only and hence releases it earlier, transaction T_k is able to execute its operations much sooner, and thus shorten the total execution time of the three transactions.

Note, that in Fig. 6.11, since T_i eventually commits, the value of x read by T_k is always consistent, regardless of whether T_j aborts, because T_j never modifies x (or any other variable read by T_k). Hence, if does T_j abort, it does not force T_k to abort and the history is nevertheless consistent. On the other hand, if the history is analogous, but T_i eventually aborts, as we show in Fig. 6.12, both T_j and T_k read inconsistent values, so both must be aborted. This will happen, because when T_i first accesses x it sets its $\mathtt{rv}_i(x)$ to 0, and as it releases x, T_i sets $\mathtt{cv}(x)$ to 1, i.e. T_i's private version. Then both T_j and T_k will both set their own recovery versions, $\mathtt{rv}_j(x)$ and $\mathtt{rv}_k(x)$ respectively, to 1. This is because transactions do not set a new value of `cv` for read-only variables. Then, when T_i eventually aborts, it reverts x to some earlier state and sets $\mathtt{cv}(x)$ to its recovery version $\mathtt{rv}_i(x)$ which is 0. Whenever transactions execute operations they must check one of the following conditions, $\mathtt{rv}_j(x) > \mathtt{cv}(k)x$ or $\mathtt{rv}_j(x) \neq \mathtt{cv}(k)x$, and are forced to abort if they fulfill any of them. Since both T_j and T_k's recovery versions for x equal 1, and the current version of x equals 0, then both transactions fulfill those conditions. So, from this point on, whenever the transactions check those conditions, they will be forced to abort. In the case of T_j the condition is first fulfilled when the thread handling read-only variables

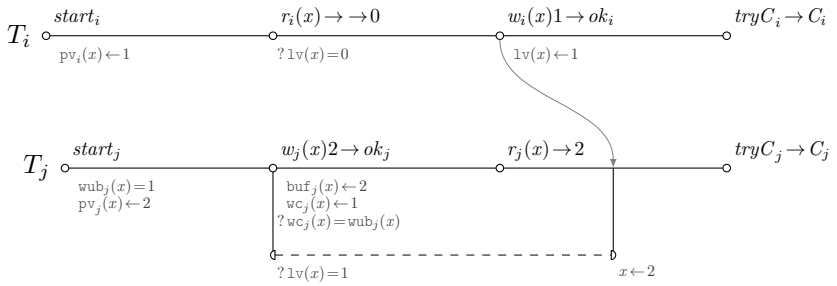


Figure 6.13: Delayed synchronization on first write (OptSVA+R).

tries to execute procedure :`read_commit`, which causes that thread to end and force the main thread of the transaction to execute the `abort` procedure (which we mark in the diagram as $tryA_j \to A_j$, but this is not a voluntary abort). If an operation in the main thread fulfilled the condition in T_j, then it would have returned A_j and the main thread would terminate all other threads instead. Transaction T_k checks the condition when it finally attempts to commit, but since the condition is fulfilled, it is forced to abort instead. Thus, T_j's abort forces both T_j and T_k to abort and consistency is maintained.

From the examples above it is apparent, that the read-only variable optimization moves the point at which such a variable is acquired, released, and committed forward in time. The earlier a shared variable is released by a transaction, the earlier another transaction can start using it, increasing the possibility of acting in parallel, and, therefore, shortening the schedule of execution.

6.3.2 Delayed Synchronization on First Write

If the first (or only) operation that a transaction executes on a particular shared variable is a write operation, then all read operations on that variable are *local*, i.e., they only need to view what the current transaction wrote, and can ignore writes by other transactions. Hence, there is no need for the transaction to synchronize on this variable with other transactions for the sake of those operations. The synchronization is only needed to prevent the current transaction from writing a value to the variable in the middle of another transaction's operations on it. But if the write is saved to a buffer, rather than immediately updating the state of the variable, the synchronization can be delayed until after the write itself, or even after any of the successive read operations.

Since it is beneficial to synchronize as late as possible while performing other tasks beforehand, OptSVA then never checks access conditions on writes (see procedure `write`): either the transaction started with a write, and no synchronization is necessary, or there was a preceding read that already did all the necessary synchronization. Instead, the operation is performed on a buffer (line 38). Then, since all the written values are only visible to the current transaction, the transaction must at some point update the state of the actual variable. This is done either upon executing the last write or during commit. In the former case, when the upper bound on writes is reached (line 41), the transaction asynchronously starts procedure :`write_buffer` (line 43), which executes when the access condition is met, and updates the state of the variable (line 65). If the upper bound is not reached during execution, the transaction will instead catch up by update the variable (line 85), also after waiting at the access condition (line 79).

We illustrate this optimization further in Fig. 6.13. Here transaction T_i can pass access condition for x first, but nevertheless T_j performs a write simultaneously, since it writes to the buffer rather than wait at the access condition. Transaction T_j only waits

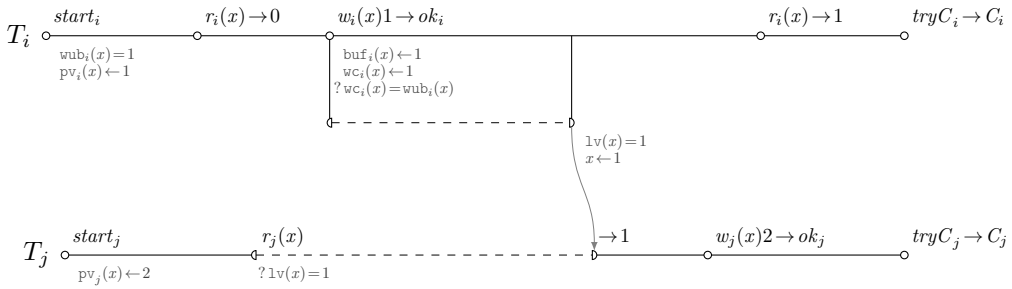


Figure 6.14: Early release on last write (OptSVA+R).

at the access condition when it had performed all of its write operations (of which there is one) and starts a separate thread (indicated by the line below) to write the changes to the variable once the access condition is passed. The thread passes the access condition once T_i releases x. Then, T_j applies the value from the buffer to x.

6.3.3 Early Release on Last Write

Various TMs with early release determine the point at which variables are released variously. For instance, DATM (see Section 4.4.1) releases variables after each operation, erring on the side of efficiency and guaranteeing only conflict-serializability. SVA, on the other hand, errs on the side of caution and only allows early release after last access to some variable, which it must do because it treats read and write operations uniformly. OptSVA improves on this, since it distinguishes between reads and writes, so early release is done after last write not last access. In effect all reads following last write are executed as if privatized. We argue in [79] that this approach is a solid compromise for TMs with early release.

The early release happens if at some point in the execution of transaction T_i, the upper bound on the number of writes for some variable x is reached when performing a write (line 41). The transaction asynchronously executes :`write_buffer` in that instance for the purpose of applying the changes from the buffer to the actual shared variable. After this is done, x will no longer be accessed directly by the transaction, so T_i also executes :`release` (at line 66), which sets $\mathtt{lv}(x)$ to $\mathtt{pv}_i(x)$, which allows other transactions to pass the access condition. Nevertheless, since x was buffered during writes, subsequent reads still have access to a local, consistent value of x (retrieved from the buffer at line 31).

This is illustrated in Fig. 6.14. Here, T_i knows *a priori* that it will write to x at most once, since $\mathtt{wub}_i(x) = 1$. Hence, after the one write to x, a separate thread is started which releases x by setting $\mathtt{lv}(x)$ to 1. Since T_i passes the access condition, this happens almost instantaneously (the figure shows a wait time merely for the reason of aesthetics). Once x is released in this fashion, T_j, whose private version for x is 2, can execute its own read and write operations on x freely. Nevertheless, T_i can continue to execute reads on x after releasing x, and since the value of x is read from $T_i's$ buffer, T_j's operations do not interfere.

6.3.4 Summary

OptSVA+R operates on the basis of the versioning mechanism, using private, local, and local terminal version counters to ensure that accesses to objects and commits are

performed in the order defined by private versions. The individual operations are handled as follows:

Start

When an OptSVA+R transaction T_i starts it acquires a private version $\mathtt{pv}_i(x)$ for each shared variable in its access set. If any of these variables are read-only (its supremum for writes equals 0), the transaction also starts separate threads that clone such variables into buffers $\mathtt{buf}_i(x)$ and release them afterward.

Read

Whenever transaction T_i attempts to execute a read operation on some variable x, its behavior primarily depends on whether the variable is read-only. If it is, the read operation waits until the separate read-only thread finishes buffering the variable, and executes the read operation on the buffer $\mathtt{buf}_i(x)$.

Otherwise, the transaction checks whether the variable was previously accessed. If not, then the transaction must wait until the access condition to x is satisfied and makes a checkpoint by copying the state of the shared variable to buffer $\mathtt{st}_i(x)$. Buffer $\mathtt{st}_i(x)$ is a copy buffer like $\mathtt{buf}_i(x)$, but it is never modified and only used to restore the variable in the event of an abort. Next, the transaction checks if any variable (for which T_i acquired a recovery version) was invalidated so far, and if so forcibly aborts. If any variable from the access set was invalidated at any point, the transaction is doomed to abort eventually, so by checking for all the variables we force it to do so as early as we can detect. This is also vital in enforcing safety: transactions should not execute any operations or local computations using stale values of variables. If there are not invalidated variables, the transaction returns the value of the buffer $\mathtt{buf}_i(x)$.

Write

When write operations are executed, the transaction first checks whether any variable for which it has a recovery version was invalidated, and aborts if that is the case. Otherwise, the write may proceed and transaction T_i simply writes a new value to its buffer $\mathtt{buf}_i(x)$. If the transaction determines this was the closing write on x, i.e. there will be no further writes on x within this transaction, T_i releases x by starting a separate thread. The thread will wait at the access condition and subsequently: make a checkpoint to $\mathtt{st}_i(x)$ (if none was made before), copy the value from the buffer $\mathtt{buf}_i(x)$ to the original variable x, re-check consistency of all variables in the access set, and release x. Meanwhile, the transaction's main thread proceeds.

Commit

When the transaction commits it waits for extant threads to finish in the case such threads are still running for read-only variables and variables that are being released after last write. Afterward, the transaction waits until the commit condition is satisfied for all variables in its access set. Then, if the transaction did not access a particular variable at any time, it makes a checkpoint. If it only ever executed writes on an variable, the transaction applies the log buffer to the variable. If the variable was not released, the transaction releases it. Afterward, the transaction checks whether any variable was invalidated, and aborts if that is the case. Otherwise, the transaction updates the local terminal versions of all variables and finishes execution. No further actions may be performed by the transaction after the commit finishes executing.

Abort

When the transaction aborts, just like with commit, it waits for the appropriate threads to finish, and for the commit condition to be satisfied. Then, each variable in the transaction's access set is restored from $\mathtt{st}_i(x)$, unless some other transaction that previously aborted already restored it to an older version beforehand. Then, the transaction updates the local terminal versions of all variables and finishes execution. No further actions may be performed by the transaction after the abort finishes executing.

6.3.5 Interleaving Comparison

In this section we compare the histories admitted by OptSVA+R to those admitted by SVA+R.

Definitions

In order to compare the interleavings of the two algorithms, let us first provide definitions of the relevant concepts.

As program $\mathbb{P}$ is being evaluated by some TM implementation, by a set of processes Π, it takes time to evaluate each statement. Hence, each event e in a trace $\mathscr{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$ appears at a specific point in time, which we denote $\tau_{\mathscr{T}}(e)$. Since each process p_k executes statements in $\mathtt{P}_k$ in sequence, then, given two events e_1, e_2 s.t. $e_1 \prec_{\mathscr{T}} e_2$, it is true that $\tau_{\mathscr{T}}(e_1) < \tau_{\mathscr{T}}(e_2)$. Given a complete operation execution op consisting of an invocation event e_1 and a response event e_2, the point of time at which op starts executing is $\tau^{\vdash}_{\mathscr{T}}(op) = \tau_{\mathscr{T}}(e_1)$ and the time at which op finishes executing is $\tau^{\dashv}_{\mathscr{T}}(op) = \tau_{\mathscr{T}}(e_2)$. The *execution time* of trace $\mathscr{T} \vdash \mathcal{E}(\mathbb{P}, \Pi)$, denoted $\tau_{\mathscr{T}}$, is equal to the largest execution time for all events in $\mathscr{T}$.

The *release time* of variable x in transaction T_i in $\mathscr{T}$, denoted $\tau^{r}_{\mathscr{T}}(T_i, x)$, is the point in time at which T_i updates $\mathtt{lv}(x)$. The *completion time* of variable x in transaction T_i in $\mathscr{T}$, denoted $\tau^{c}_{\mathscr{T}}(T_i, x)$, is the point in time at which T_i updates $\mathtt{ltv}(x)$.

Execution Time

In this section, we show that the execution time of OptSVA+R histories is lower than that of SVA+R histories resulting from the execution of the same program by the same processes.

Let $\mathcal{E}_{\mathrm{S}}(\mathbb{P}, \Pi)$ denote a complete execution of program $\mathbb{P}$ by processes Π according to the SVA+R concurrency control algorithm, and $\mathcal{E}_{\mathrm{O}}(\mathbb{P}, \Pi)$, an otherwise identical execution, but according to OptSVA+R. Then, there are traces $\mathscr{T}_{\mathrm{S}} \vdash \mathcal{E}_{\mathrm{S}}(\mathbb{P}, \Pi)$ and $\mathscr{T}_{\mathrm{O}} \vdash \mathcal{E}_{\mathrm{O}}(\mathbb{P}, \Pi)$, and histories $H_{\mathrm{S}} = \mathit{Hist}(\mathscr{T}_{\mathrm{S}})$ and $H_{\mathrm{O}} = \mathit{Hist}(\mathscr{T}_{\mathrm{O}})$. We denote the set of all transactions in H_{O} and H_{S} as $\mathbb{T}$. The histories contain corresponding transactions: $T_i \in H_{\mathrm{S}}$ if, and only if, $T_i \in H_{\mathrm{O}}$. Note that corresponding transactions execute the same sequence of operations in both histories, i.e., for any $T_i \in \mathbb{T}$, $H_{\mathrm{S}}|T_i = H_{\mathrm{O}}|T_i$. We also assume the variable model for both SVA+R and OptSVA+R, so $\mathit{Obj} = \mathit{Var}$.

For the purpose of the comparison we assume that the events in histories are instantaneous. We also do not account for the time it takes to execute concurrency control code. This means that if some operation execution op does not wait for either the access condition nor the commit condition, we consider op to be of constant length between SVA+R or OptSVA+R, i.e, $\tau^{\dashv}_{H_{\mathrm{S}}}(op) = \tau^{\vdash}_{H_{\mathrm{S}}}(op) + \Delta^{S}_{op}$, $\tau^{\dashv}_{H_{\mathrm{O}}}(op) = \tau^{\vdash}_{H_{\mathrm{O}}}(op) + \Delta^{O}_{op}$, and $\Delta^{S}_{op} = \Delta^{O}_{op}$. Finally, we assume that apart from the details of the concurrency control, the execution proceeds the same, regardless of whether it is SVA+R or OptSVA+R. This means that transactions start executing at the same time in both SVA+R and

OptSVA+R histories: for each $T_i \in \mathbb{T}$, $\tau^{\vdash}_{H_\mathrm{S}}(start_i \rightarrow ok_i) = \tau^{\vdash}_{H_\mathrm{O}}(start_i \rightarrow ok_i)$. It also means that the time between operation executions is constant within a transaction (although the time between an invocation event of an operation execution and the response event of that operation execution may differ between algorithms). More formally, for any $T_i \in \mathbb{T}$, if $H_\mathrm{S}|T_i = H_\mathrm{O}|T_i = \left[op_1, op_2, ...op_n\right]$, then for any $k = 2, 3, ..., n$, $\tau^{\vdash}_{H_\mathrm{S}}(op_k) = \tau^{\dashv}_{H_\mathrm{S}}(op_{k-1}) + \delta^S_{k-1}$, $\tau^{\vdash}_{H_\mathrm{O}}(op_k) = \tau^{\dashv}_{H_\mathrm{O}}(op_{k-1}) + \delta^O_{k-1}$, and $\delta^O_{k-1} = \delta^S_{k-1}$.

We say transaction T_i is *initial* if for every $x \in \mathtt{ASet}_i$, $\mathtt{pv}_i(x) = 1$. Note that initial transactions instantaneously satisfy the access and commit conditions for all the variables in their access sets. We say transaction T_i *waits for* transaction T_j (on variable x) if for some variable $x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$, $\mathtt{pv}_i(x) = \mathtt{pv}_j(x) + 1$. Intuitively, if T_i waits for T_j then T_j is the older transaction, since it started earlier. Note that in both SVA+R and OptSVA+R, if T_i waits for T_j, then for every variable $y \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$, it is true that $\mathtt{pv}_i(y) > \mathtt{pv}_j(x)$.

The main lemma of this demonstration will be proved by induction, after we introduce some helper lemmas. In the first step we show that in the case of initial transactions an OptSVA+R transaction executes operations at the same time as the corresponding SVA+R transaction (which implies that the OptSVA+R transaction executes operations no later than the corresponding SVA+R transaction).

Lemma 29 (Initial Early Operation Execution)**.** *For any initial $T_i \in \mathbb{T}$, and any operation execution op in $H_\mathrm{O}|T_i$ and $H_\mathrm{S}|T_i$, $\tau^{\dashv}_{H_\mathrm{O}}(op) = \tau^{\dashv}_{H_\mathrm{S}}(op)$.*

Proof. Since T_i is initial it does not wait at any access conditions or commit conditions. Then, any operation execution *op* in $H_\mathrm{O}|T_i$ and $H_\mathrm{S}|T_i$ is constant in both SVA+R and OptSVA+R, so:

$$\tau^{\dashv}_{H_\mathrm{S}}(op) = \tau^{\vdash}_{H_\mathrm{S}}(op) + \Delta_{op},$$

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \tau^{\vdash}_{H_\mathrm{O}}(op) + \Delta_{op}.$$

Then, since local computations performed by transactions are constant between SVA+R and OptSVA+R, and since $\tau^{\vdash}_{H_\mathrm{S}}(start_i \rightarrow ok_i) = \tau^{\vdash}_{H_\mathrm{O}}(start_i \rightarrow ok_i)$, then trivially:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \tau^{\dashv}_{H_\mathrm{S}}(op). \qquad \square$$

Next, we show that an OptSVA+R transaction releases variables, commits, and aborts (i.e., update `lv` and `ltv` counters) no later than the corresponding initial SVA+R transaction.

Lemma 30 (Initial Early Release)**.** *For any initial $T_i \in \mathbb{T}$ and $x \in \mathtt{ASet}_i$, $\tau^{r}_{H_\mathrm{O}}(T_i, x) \leqslant \tau^{r}_{H_\mathrm{S}}(T_i, x)$.*

Proof. An SVA+R transaction releases x by updating $\mathtt{lv}(x)$ on commit, on abort, and during the last operation execution on x. An OptSVA+R transactions does so on commit, on abort, during the last write operation execution on x, and after buffering a read-only variable.

a) If x is a read-only variable an SVA+R transaction releases x no sooner than the last operation execution on x, so given any read operation execution $r_i(x) \rightarrow \square \in H_\mathrm{S}|T_i$:

$$\tau^{r}_{H_\mathrm{S}}(T_i, x) \geqslant \tau^{\dashv}_{H_\mathrm{S}}(r_i(x) \rightarrow \square).$$

On the other hand, OptSVA+R releases x as soon as possible. That is during $start_i \rightarrow ok_i$ at the earliest and no later than any $r_i(x) \rightarrow v \in H_\mathrm{O}|T_i$ at the latest. Thus:

$$\tau^{\dashv}_{H_\mathrm{O}}(start_i \rightarrow ok_i) \leqslant \tau^{r}_{H_\mathrm{O}}(T_i, x) \leqslant \tau^{\dashv}_{H_\mathrm{O}}(r_i(x) \rightarrow \square).$$

From Lemma 29:

$$\tau^{\dashv}_{H_\mathrm{O}}(r_i(x)\to\square) = \tau^{\dashv}_{H_\mathrm{S}}(r_i(x)\to\square).$$

In that case:

$$\tau^{r}_{H_\mathrm{O}}(T_i, x) \leqslant \tau^{r}_{H_\mathrm{S}}(T_i, x).$$

b) Alternatively, if the last operation execution in $H_\mathrm{S}|T_i|x$ is $r_i(x)\to\square$ and assuming tight suprema, an SVA+R transaction releases x during $r_i(x)\to\square$, so:

$$\tau^{r}_{H_\mathrm{S}}(T_i, x) = \tau^{\dashv}_{H_\mathrm{S}}(r_i(x)\to\square).$$

On the other hand, if the last operation execution in $H_\mathrm{O}|T_i|x$ is $r_i(x)\to\square$ and if the suprema are tight, then an OptSVA+R transaction releases x no sooner than any $w_i(x)\square\to ok_i$ in $H_\mathrm{O}|T_i$ and no later than $r_i(x)\to\square$. Hence:

$$\tau^{\dashv}_{H_\mathrm{O}}(r_i(x)\to\square) \geqslant \tau^{r}_{H_\mathrm{O}}(T_i, x) \geqslant \tau^{\dashv}_{H_\mathrm{O}}(w_i(x)\square\to ok_i).$$

From Lemma 29:

$$\tau^{\dashv}_{H_\mathrm{O}}(r_i(x)\to\square) = \tau^{\dashv}_{H_\mathrm{S}}(r_i(x)\to\square).$$

Therefore:

$$\tau^{r}_{H_\mathrm{O}}(T_i, x) \leqslant \tau^{r}_{H_\mathrm{S}}(T_i, x).$$

c) Otherwise, if the last operation execution on x in T_i is $r_i(x)\to\square$ but suprema are not tight, then both OptSVA+R and SVA+R release x either during commit, or abort. Thus:

$$\tau^{r}_{H_\mathrm{S}}(T_i, x) = \tau_{H_\mathrm{S}}(res_i\big[C_i\big]) \text{ or } \tau^{r}_{H_\mathrm{S}}(T_i, x) = \tau_{H_\mathrm{S}}(res_i\big[A_i\big]), \text{ and}$$

$$\tau^{r}_{H_\mathrm{O}}(T_i, x) = \tau_{H_\mathrm{O}}(res_i\big[C_i\big]) \text{ or } \tau^{r}_{H_\mathrm{O}}(T_i, x) = \tau_{H_\mathrm{O}}(res_i\big[A_i\big]).$$

From Lemma 29, for any op in T_i:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \tau^{\dashv}_{H_\mathrm{S}}(op).$$

This implies that:

$$\tau_{H_\mathrm{O}}(res_i\big[C_i\big]) = \tau_{H_\mathrm{S}}(res_i\big[C_i\big]) \text{ or } \tau_{H_\mathrm{O}}(res_i\big[A_i\big]) = \tau_{H_\mathrm{S}}(res_i\big[A_i\big]).$$

Therefore:

$$\tau^{r}_{H_\mathrm{O}}(T_i, x) = \tau^{r}_{H_\mathrm{S}}(T_i, x).$$

d) Finally, if the last operation execution in T_i is $w_i(x)\square\to ok_i$, both SVA+R and OptSVA+R transactions will release x during $w_i(x)\square\to ok_i$ if the suprema are tight or otherwise during commit or abort. In the former case, from Lemma 29:

$$\tau^{\dashv}_{H_\mathrm{O}}(w_i(x)\square\to ok_i) = \tau^{\dashv}_{H_\mathrm{S}}(w_i(x)\square\to ok_i).$$

Thus:

$$\tau^{r}_{H_\mathrm{O}}(T_i, x) = \tau^{r}_{H_\mathrm{S}}(T_i, x).$$

In the latter case, by analogy to c:

$$\tau^{r}_{H_\mathrm{O}}(T_i, x) = \tau^{r}_{H_\mathrm{S}}(T_i, x).$$

Thus, for any initial transaction T_i and any $x \in \mathtt{ASet}_i$, $\tau^{r}_{H_\mathrm{O}}(T_i, x) \leqslant \tau^{r}_{H_\mathrm{S}}(T_i, x)$. $\square$

Lemma 31 (Initial Early Completion)**.** *For any initial $T_i \in \mathbb{T}$ and $x \in \mathtt{ASet}_i$, $\tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau^c_{H_\mathrm{S}}(T_i, x)$.*

Proof. An SVA+R transaction updates $\mathtt{ltv}(x)$ on commit or on abort, so:

$$\tau^c_{H_\mathrm{S}}(T_i, x) = \tau_{H_\mathrm{S}}(res_i[C_i]) \text{ or } \tau^c_{H_\mathrm{S}}(T_i, x) = \tau_{H_\mathrm{S}}(res_i[A_i]).$$

An OptSVA+R transaction updates $\mathtt{ltv}(x)$ on commit, on abort, or after releasing a read-only variable. The latter-most potentially precedes a commit, so:

$$\tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau_{H_\mathrm{O}}(res_i[C_i]) \text{ or } \tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau_{H_\mathrm{O}}(res_i[A_i]).$$

From Lemma 29, for any *op* in T_i:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \tau^{\dashv}_{H_\mathrm{S}}(op).$$

This implies that:

$$\tau_{H_\mathrm{O}}(res_i[C_i]) = \tau_{H_\mathrm{S}}(res_i[C_i]) \text{ and } \tau_{H_\mathrm{O}}(res_i[A_i]) = \tau_{H_\mathrm{S}}(res_i[A_i]).$$

Therefore for any initial $T_i \in \mathbb{T}$:

$$\tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau^c_{H_\mathrm{S}}(T_i, x). \qquad \square$$

We then demonstrate that any OptSVA+R transaction T_i executes operations no later than the corresponding SVA+R transaction, provided that each transaction T_j for which T_i waits releases variables, commits, and aborts no later than the corresponding SVA+R transaction.

Lemma 32 (Consecutive Early Operation Execution)**.** *For any $T_i \in \mathbb{T}$ and any operation execution op in $H_\mathrm{O}|T_i$ and in $H_\mathrm{S}|T_i$ it is true that $\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op)$, given that if T_i waits for any older transaction T_j on some variable x, then $\tau^r_{H_\mathrm{O}}(T_j, x) \leqslant \tau^r_{H_\mathrm{S}}(T_j, x)$ and $\tau^c_{H_\mathrm{O}}(T_j, x) \leqslant \tau^c_{H_\mathrm{S}}(T_j, x)$.*

Proof. The case for $\mathtt{pv}_i(x) = 1$ is trivial. If $\mathtt{pv}_i(x) > 1$ then there exists $T_j \in \mathbb{T}$ s.t. $\mathtt{pv}_j(x) + 1 = \mathtt{pv}_i(x)$. We first assume for convenience that *op* is not preceded in T_i by operations on variables other than x.

I If *op* is a read operation execution, *op* can return a value and be a non-local read operation execution, or a local one, or the operation can return A_i.

a) If $op = r_i(x) \to v$ is a non-local read operation execution in both SVA+R and OptSVA+R, then there is some non-local $op_r = r_i(x) \to v$ (possibly $op_r = op$) such that op_r is the first operation in $H_\mathrm{S}|T_i|x$ and $H_\mathrm{O}|T_i|x$. Operation execution op_r will not finish before the access condition is satisfied. If the access condition is met before op_r starts, then trivially from V (below):

$$\tau^{\dashv}_{H_\mathrm{O}}(op_r) = \tau^{\dashv}_{H_\mathrm{S}}(op_r).$$

Otherwise, the operation will finish executing as soon as the access condition can be satisfied, so:

$$\tau^{\dashv}_{H_\mathrm{S}}(op_r) = \tau^r_{H_\mathrm{S}}(T_j, x),$$
$$\tau^{\dashv}_{H_\mathrm{O}}(op_r) = \tau^r_{H_\mathrm{O}}(T_j, x).$$

Then, since T_i waits for T_j, then from the Lemma's assumption:

$$\tau^r_{H_\mathrm{O}}(T_j, x) \leqslant \tau^r_{H_\mathrm{S}}(T_j, x),$$

Therefore:

$$\tau^{\dashv}_{H_\mathrm{O}}(op_r) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op_r).$$

Then, since either $op_r = op$ or op_r precedes op, and since any non-local read operation executions on x following op_r already satisfy the access condition:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

b) If $op = r_i(x) \rightarrow v$ is a local read operation execution, then, by definition, local reads follow a write operation execution, so there is some $op_w = w_i(x)\square \rightarrow ok_i$ in T_i s.t. $op_w \prec_{H_\mathrm{O}} op$ and $op_w = w_i(x)v \rightarrow ok_i \in H_\mathrm{S}|T_i$ s.t. $op_w \prec_{H_\mathrm{S}} op$. We assume without loss of generality that op_w immediately precedes op in T_i. Since local computation time is equal in both H_S and H_O, if we denote the length of time between the write operation returns and the read operation is invoked in either history as δ, then:

$$\tau^{\vdash}_{H_\mathrm{S}}(op) = \tau^{\dashv}_{H_\mathrm{S}}(op_w) + \delta,$$

$$\tau^{\vdash}_{H_\mathrm{O}}(op) = \tau^{\dashv}_{H_\mathrm{O}}(op_w) + \delta.$$

In addition, given that in SVA+R the access condition will already have been passed by the time op executes, and given that the access condition is not checked in OptSVA+R for non-local reads, then op executes in constant time in both algorithms. Thus:

$$\tau^{\dashv}_{H_\mathrm{S}}(op) = \tau^{\vdash}_{H_\mathrm{S}}(op) + \Delta_{op} = \tau^{\dashv}_{H_\mathrm{S}}(op_w) + \delta + \Delta_{op},$$

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \tau^{\vdash}_{H_\mathrm{O}}(op) + \Delta_{op} = \tau^{\dashv}_{H_\mathrm{O}}(op_w) + \delta + \Delta_{op}.$$

From II (below):

$$\tau^{\dashv}_{H_\mathrm{O}}(op_w) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op_w).$$

Hence, it follows that:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

c) If $op = r_i(x) \rightarrow A_i$, then in both SVA+R and OptSVA+R the operation execution waits until $\mathtt{ltv}(y) = \mathrm{pv}_i(y) - 1$ is true for all $y \in \mathtt{ASet}_i$. So, for each y, there is some transaction T_k s.t. $\mathrm{pv}_i(y) - 1 = \mathrm{pv}_k(y)$. Every such T_k must update $\mathtt{ltv}(y)$ to $\mathrm{pv}_k(y)$ before T_i can abort. If this happens before the invocation event of op, then the execution of the abort depends only on the execution of preceding operations in T_i. Thus from Ia, Ib, IIa, IIb, III, and V:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

Otherwise, the operation will finish executing as soon as the last transaction T_k s.t. $\mathrm{pv}_i(y) - 1 = \mathrm{pv}_k(y)$ updates $\mathtt{ltv}(y)$ to $\mathrm{pv}_k(y)$. Thus:

$$\tau^{\dashv}_{H_\mathrm{S}}(op) = \max_{\forall T_k, y} \tau^{c}_{H_\mathrm{S}}(T_k, y), \text{ where } \mathrm{pv}_i(y) - 1 = \mathrm{pv}_k(y),$$

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \max_{\forall T_k, y} \tau^{c}_{H_\mathrm{O}}(T_k, y) \text{ where } \mathrm{pv}_i(y) - 1 = \mathrm{pv}_k(y).$$

Since T_i necessarily waits for each such T_k, then, from the Lemma's assumptions:

$$\tau^{c}_{H_\mathrm{O}}(T_k, y) \leqslant \tau^{c}_{H_\mathrm{S}}(T_k, y),$$

Therefore:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

Thus, if op is a read operation execution, $\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op)$.

II If op is a write operation execution, op can return ok_i and be preceded by a non-local read operation execution. Alternatively, it can return ok_i and be preceded by no operations or by a write (and possibly by non-local read operation executions). Otherwise the write operation execution can return A_i.

a) If $op = w_i(x)\Box \to ok_i$ is preceded by some non-local $op_r = r_i(x) \to \Box$ in T_i, in both SVA+R and OptSVA+R then we assume without loss of generality that op immediately follows op_r in T_i. In that case, since local computation time is equal in both H_S and H_O, if we denote the length of time between the read operation returns and the write operation is invoked in either history as δ, then:

$$\tau^{\vdash}_{H_\mathrm{S}}(op) = \tau^{\dashv}_{H_\mathrm{S}}(op_r) + \delta,$$

$$\tau^{\vdash}_{H_\mathrm{O}}(op) = \tau^{\dashv}_{H_\mathrm{O}}(op_r) + \delta.$$

In addition, given that in SVA+R the access condition will already have been passed by the time op executes, and given that the access condition is not checked in OptSVA+R for non-local reads, then op executes in constant time in both algorithms. Thus:

$$\tau^{\dashv}_{H_\mathrm{S}}(op) = \tau^{\vdash}_{H_\mathrm{S}}(op) + \Delta_{op} = \tau^{\dashv}_{H_\mathrm{S}}(op_r) + \delta + \Delta_{op},$$

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \tau^{\vdash}_{H_\mathrm{O}}(op) + \Delta_{op} = \tau^{\dashv}_{H_\mathrm{O}}(op_r) + \delta + \Delta_{op}.$$

From Ia:

$$\tau^{\dashv}_{H_\mathrm{O}}(op_r) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op_r).$$

Therefore:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

b) If $op = w_i(x)\Box \to ok_i$ is not preceded by non-local read operation executions, then there is such $op_w = w_i(x)\Box \to ok_i$ (possibly $op_w = op$) such that op_w is the first operation in $H_\mathrm{S}|T_i|x$ and $H_\mathrm{O}|T_i|x$. In addition, op_w is necessarily preceded by $op_s = start_i \to ok_i$, so:

$$\tau^{\dashv}_{H_\mathrm{S}}(op_w) > \tau^{\dashv}_{H_\mathrm{S}}(op_s),$$

$$\tau^{\dashv}_{H_\mathrm{O}}(op_w) > \tau^{\dashv}_{H_\mathrm{O}}(op_s).$$

In SVA+R the first write waits for the access condition, so:

$$\tau^{\dashv}_{H_\mathrm{S}}(op) \geqslant \max(\tau^{r}_{H_\mathrm{S}}(T_j, x), \tau^{\dashv}_{H_\mathrm{S}}(op_s)).$$

In OptSVA+R such writes do not wait for the access condition at all, so:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \geqslant \tau^{\dashv}_{H_\mathrm{O}}(op_s), \text{ regardless of } \tau^{r}_{H_\mathrm{O}}(T_j, x).$$

From V:

$$\tau^{\dashv}_{H_\mathrm{O}}(op_s) = \tau^{\dashv}_{H_\mathrm{S}}(op_s).$$

Then, since either $op_w = op$ or op_w precedes op:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

c) If $op = w_i(x)\square \rightarrow A_i$, then, by analogy to Ic:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

Thus, if op is a write operation execution, $\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op)$.

III If $op = tryC_i \rightarrow \square$, then in both SVA+R and OptSVA+R transactions wait until $\mathtt{ltv}(y) = \mathtt{pv}_i(y) - 1$ is true for all $y \in \mathtt{ASet}_i$ before returning from op. This means that each transaction T_k s.t. $\mathtt{pv}_k(y) + 1 = \mathtt{pv}_i(y)$ must update $\mathtt{ltv}(y)$ to $\mathtt{pv}_k(y)$ before T_i can commit or abort. If this happens before the invocation event of op, then the execution of the commit depends only on the execution of preceding operations in T_i. Thus from Ia, Ib, IIa, IIb, and V:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

Otherwise, the operation will finish executing as soon as the last transaction T_k s.t. $\mathtt{pv}_i(y) - 1 = \mathtt{pv}_k(y)$ updates $\mathtt{ltv}(y)$ to $\mathtt{pv}_k(y)$. Thus:

$$\tau^{\dashv}_{H_\mathrm{S}}(op) = \max_{\forall T_k, y} \tau^{c}_{H_\mathrm{S}}(T_k, y), \text{ where } \mathtt{pv}_i(y) - 1 = \mathtt{pv}_k(y),$$

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \max_{\forall T_k, y} \tau^{c}_{H_\mathrm{O}}(T_k, y), \text{ where } \mathtt{pv}_i(y) - 1 = \mathtt{pv}_k(y).$$

Since T_i necessarily waits for each such T_k, then, from the Lemma's assumptions:

$$\tau^{c}_{H_\mathrm{O}}(T_k, y) \leqslant \tau^{c}_{H_\mathrm{S}}(T_k, y),$$

Therefore:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

IV If $op = tryA_i \rightarrow A_i$, then, by analogy to III:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

V If $op = start_i \rightarrow ok_i$, then, trivially:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) = \tau^{\dashv}_{H_\mathrm{S}}(op).$$

Let us now assume that op is preceded by operations on variables other than x in $H_\mathrm{S}|T_i$ and in $H_\mathrm{O}|T_i$. In that case, there is some operation execution op' on variable y that precedes op in $H_\mathrm{S}|T_i$ and in $H_\mathrm{O}|T_i$ and that is not preceded by operation executions on variables other than y. It then follows from I-V that:

$$\tau^{\dashv}_{H_\mathrm{O}}(op') \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op').$$

Since the time it takes to execute local computations is constant between OptSVA+R and SVA+R, then trivially, for any operation execution in op'' that follows op' in $H_\mathrm{S}|T_i$ and in $H_\mathrm{O}|T_i$ it is also true that:

$$\tau^{\dashv}_{H_\mathrm{O}}(op'') \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op'').$$

Therefore, for any operation execution op in T_i given that for any T_j, s.t. T_i waits (on x), it is true that $\tau^{r}_{H_\mathrm{O}}(T_i, x) \leqslant \tau^{r}_{H_\mathrm{S}}(T_i, x)$ and $\tau^{c}_{H_\mathrm{O}}(T_i, x) \leqslant \tau^{c}_{H_\mathrm{S}}(T_i, x)$:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op). \qquad \square$$

We also demonstrate that any OptSVA+R transaction T_i releases variables, commits, and aborts no later than the corresponding SVA+R transaction if each transaction T_j for which T_i waits executes operations no later than its corresponding SVA+R transaction.

Lemma 33 (Consecutive Early Release)**.** *For any $T_i \in \mathbb{T}$ and $x \in \mathtt{ASet}_i$, $\tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x)$ given that if T_i waits for any older transaction T_j on some variable y, then $\tau^r_{H_\mathrm{O}}(T_j, y) \leqslant \tau^r_{H_\mathrm{S}}(T_j, y)$ and $\tau^c_{H_\mathrm{O}}(T_j, y) \leqslant \tau^c_{H_\mathrm{S}}(T_j, y)$.*

Proof. An SVA+R transaction releases x by updating $\mathtt{lv}(x)$ on commit, on abort, and during the last operation execution on x. An OptSVA+R transactions does so on commit, on abort, during the last write operation execution on x, and after buffering a read-only variable.

a) If x is a read-only variable, an SVA+R transaction releases x no sooner than the last operation execution on x, so given any read operation execution $r_i(x)\to\Box \in H_\mathrm{S}|T_i$:

$$\tau^r_{H_\mathrm{S}}(T_i, x) \geqslant \tau^\dashv_{H_\mathrm{S}}(r_i(x)\to\Box).$$

On the other hand, OptSVA+R releases x as soon as possible. That is during $start_i \to ok_i$ at the earliest and no later than any $r_i(x)\to v \in H_\mathrm{O}|T_i$ at the latest. Thus:

$$\tau^\dashv_{H_\mathrm{O}}(start_i \to ok_i) \leqslant \tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^\dashv_{H_\mathrm{O}}(r_i(x)\to\Box).$$

From Lemma 32:

$$\tau^\dashv_{H_\mathrm{O}}(r_i(x)\to\Box) \leqslant \tau^\dashv_{H_\mathrm{S}}(r_i(x)\to\Box).$$

In that case:

$$\tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x).$$

b) Alternatively, if the last operation execution in $H_\mathrm{S}|T_i|x$ is $r_i(x)\to\Box$ and assuming tight suprema, an SVA+R transaction releases x during $r_i(x)\to\Box$, so:

$$\tau^r_{H_\mathrm{S}}(T_i, x) = \tau^\dashv_{H_\mathrm{S}}(r_i(x)\to\Box).$$

On the other hand, if last operation execution in $H_\mathrm{O}|T_i|x$ is $r_i(x)\to\Box$ and if the suprema are tight, then an OptSVA+R transaction releases x no sooner than any $w_i(x)\Box\to ok_i$ in $H_\mathrm{O}|T_i$ and no later than $r_i(x)\to\Box$. Hence:

$$\tau^\dashv_{H_\mathrm{O}}(r_i(x)\to\Box) \geqslant \tau^r_{H_\mathrm{O}}(T_i, x) \geqslant \tau^\dashv_{H_\mathrm{O}}(w_i(x)\Box\to ok_i).$$

From Lemma 32:

$$\tau^\dashv_{H_\mathrm{O}}(r_i(x)\to\Box) \leqslant \tau^\dashv_{H_\mathrm{S}}(r_i(x)\to\Box).$$

Therefore:

$$\tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x).$$

c) Otherwise, if the last operation execution on x in T_i is $r_i(x)\to\Box$ but suprema are not tight, then both OptSVA+R and SVA+R release x either during commit, or abort. Thus:

$$\tau^r_{H_\mathrm{S}}(T_i, x) = \tau_{H_\mathrm{S}}(res_i\big[C_i\big]) \text{ or } \tau^r_{H_\mathrm{S}}(T_i, x) = \tau_{H_\mathrm{S}}(res_i\big[A_i\big]), \text{ and}$$

$$\tau^r_{H_\mathrm{O}}(T_i, x) = \tau_{H_\mathrm{O}}(res_i\big[C_i\big]) \text{ or } \tau^r_{H_\mathrm{O}}(T_i, x) = \tau_{H_\mathrm{O}}(res_i\big[A_i\big]).$$

From Lemma 32, for any op in T_i:

$$\tau^\dashv_{H_\mathrm{O}}(op) \leqslant \tau^\dashv_{H_\mathrm{S}}(op).$$

This implies that:

$$\tau_{H_\mathrm{O}}(res_i[C_i]) \leqslant \tau_{H_\mathrm{S}}(res_i[C_i]) \text{ or } \tau_{H_\mathrm{O}}(res_i[A_i]) \leqslant \tau_{H_\mathrm{S}}(res_i[A_i]).$$

Therefore:

$$\tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x).$$

d) Finally, if the last operation execution in T_i is $w_i(x)\Box \to ok_i$, both SVA+R and OptSVA+R transactions will release x during $w_i(x)\Box \to ok_i$ if the suprema are tight or otherwise during commit or abort. In the former case, from Lemma 32:

$$\tau^{\dashv}_{H_\mathrm{O}}(w_i(x)\Box \to ok_i) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(w_i(x)\Box \to ok_i).$$

Thus:

$$\tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x).$$

In the latter case, by analogy to c, also:

$$\tau^r_{H_\mathrm{O}}(T_i, x) = \tau^r_{H_\mathrm{S}}(T_i, x).$$

Therefore, given T_i as defined above, $\tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x)$. □

Lemma 34 (Consecutive Early Completion)**.** *For any $T_i \in \mathbb{T}$ and $x \in \mathtt{ASet}_i$, $\tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau^c_{H_\mathrm{S}}(T_i, x)$ given that if T_i waits for any older transaction T_j on some variable y, then $\tau^r_{H_\mathrm{O}}(T_j, y) \leqslant \tau^r_{H_\mathrm{S}}(T_j, y)$ and $\tau^c_{H_\mathrm{O}}(T_j, y) \leqslant \tau^c_{H_\mathrm{S}}(T_j, y)$.*

Proof. An SVA+R transaction updates $\mathtt{ltv}(x)$ on commit, or on abort, so:

$$\tau^c_{H_\mathrm{S}}(T_i, x) = \tau_{H_\mathrm{S}}(res_i[C_i]) \text{ or } \tau^c_{H_\mathrm{S}}(T_i, x) = \tau_{H_\mathrm{S}}(res_i[A_i]).$$

An OptSVA+R transaction updates $\mathtt{ltv}(x)$ on commit, on abort, or after releasing a read-only variable. The latter-most potentially precedes a commit, so:

$$\tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau_{H_\mathrm{S}}(res_i[C_i]) \text{ or } \tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau_{H_\mathrm{S}}(res_i[A_i]).$$

From Lemma 32, for any *op* in T_i:

$$\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op).$$

Therefore, for any such $T_i \in \mathbb{T}$:

$$\tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau^c_{H_\mathrm{S}}(T_i, x).$$ □

Finally, we show by induction that any OptSVA+R transaction executes any operation no later than the corresponding SVA+R transaction, which means that the execution time of an OptSVA+R history will be equal to or shorter than that of an SVA+R history generated from the same execution.

Lemma 35 (Early Operation Execution)**.** *For any $T_i \in \mathbb{T}$, and any operation execution op in $H_\mathrm{O}|T_i$ and $H_\mathrm{S}|T_i$, $\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op)$.*

Proof. Induction:

I For any initial $T_i \in \mathbb{T}$:

a) From Lemma 29, for any operation execution *op* in T_i, $\tau^{\dashv}_{H_\mathrm{O}}(op) \leqslant \tau^{\dashv}_{H_\mathrm{S}}(op)$,

b) From Lemma 30, for any variable $x \in \mathtt{ASet}_i$, $\tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x)$, and

c) From Lemma 31, for any variable $x \in \mathtt{ASet}_i$, $\tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x)$.

II For any transaction $T_i \in \mathbb{T}$, if for each $T_j \in \mathbb{T}$ s.t. T_i waits for T_j:

a) For any operation execution *op* in T_j, $\tau^\dashv_{H_\mathrm{O}}(op) \leqslant \tau^\dashv_{H_\mathrm{S}}(op)$,
b) For any variable $x \in \mathtt{ASet}_j$, $\tau^r_{H_\mathrm{O}}(T_j, x) \leqslant \tau^r_{H_\mathrm{S}}(T_j, x)$, and
c) For any variable $x \in \mathtt{ASet}_j$, $\tau^c_{H_\mathrm{O}}(T_j, x) \leqslant \tau^c_{H_\mathrm{S}}(T_j, x)$.

Then:

a) From Lemma 32, for any operation execution *op* in T_i, $\tau^\dashv_{H_\mathrm{O}}(op) \leqslant \tau^\dashv_{H_\mathrm{S}}(op)$,
b) From Lemma 33, for any variable $x \in \mathtt{ASet}_i$, $\tau^r_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x)$, and
c) From Lemma 34, for any variable $x \in \mathtt{ASet}_i$, $\tau^c_{H_\mathrm{O}}(T_i, x) \leqslant \tau^r_{H_\mathrm{S}}(T_i, x)$.

It follows from induction that for any $T_i \in \mathbb{T}$, and any operation execution *op* in $H_\mathrm{O}|T_i$ and $H_\mathrm{S}|T_i$, $\tau^\dashv_{H_\mathrm{O}}(op) \leqslant \tau^\dashv_{H_\mathrm{S}}(op)$. □

Corollary 18 (Lower Execution Time)**.** $\tau_{H_\mathrm{O}} \leqslant \tau_{H_\mathrm{S}}$.

Hence, the execution time of OptSVA+R is no worse than SVA+R. Intuitively, OptSVA+R is likely perform better in almost all cases though, and especially, if high contention causes many transactions to wait to access the same object—then, the expedited release times and delayed synchronization come into play.

6.3.6 Properties

Despite achieving a higher level of parallelism, OptSVA+R retains the property of last-use opacity, just like SVA+R. We provide a proof and a discussion of the proof technique employed in Section 7.3. OptSVA+R is also trivially deadlock free. However, OptSVA+R does not preserve commitment order, like SVA+R does (see Fig. 6.11). We consider commitment order a guarantee of secondary important for TM though. On the other hand, commitment order preservation can be trivially ensured in OptSVA+R, if the `read_commit` procedure is not executed asynchronously at the end of the `read_buffer` procedure, but instead, executed synchronously during commit.

6.3.7 Reluctant Transactions

Given that a cascading abort may occur, OptSVA+R may be forced to abort transactions that contain irrevocable operations. Hence, we present a reluctant variant of OptSVA+R. ROptSVA+R operates by analogy to RSVA+R: a class of *reluctant* transactions $\mathbb{R}$ may be designated (manually by the programmer or through static analysis), that use a more conservative access condition, and all transactions within that class refuse to access variables that were potentially modified by uncommitted transactions by waiting until the transaction that modified it most recently commits or aborts. As with RSVA+R, this is accomplished by checking the commit condition $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{ltv}(\lceil x \rfloor)$ rather than the access condition $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$ when accessing variables. Reluctant transactions never encounter inconsistent views, and therefore may be treated as irrevocable. The drawback of this solution is that such transactions may wait longer to access shared objects, but, in return, they never forcibly abort. Nevertheless, the programmer retains the power to electively abort such transactions, e.g. due to some business logic requirement. We give RSVA+R's pseudocode in Fig. 6.15 with the changed access condition highlighted.

```
1  proc start(Transaction T_i) {
2    // Acquire private versions.
3    for x ∈ ASet_i in order
4      lock lk(x) → W
5    for x ∈ ASet_i {
6      gv(x) ← gv(x) + 1
7      pv_i(x) ← gv(x)
8      unlock lk(x)
9    }
10   // Asynchronously buffer read−only variables.
11   for x ∈ ASet_i: wub_i(x) = 0
12     if T_i ∈ ℝ
13       async run :read_buffer(T_i, x)
14         when pv_i(x) - 1 = ltv(x)
15     else
16       async run :read_buffer(T_i, x)
17         when pv_i(x) - 1 = lv(x)
18   return ok_i
19 }
20 proc read(Transaction T_i, Variable x) {
21   // Wait for read−only variable to be bufferred.
22   if wub_i(x) = 0
23     join with :read_buffer(T_i, x)
24   // Copy value of variable to buffer on first read.
25   else if wc_i(x) = 0 and rc_i(x) = 0 {
26     if T_i ∈ ℝ
27       wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
28     else
29       wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
30     :checkpoint(T_i,x)
31     buf_i(x) ← st_i(x)
32   }
33   // Abort on inconsistent view.
34   if ∃y: rv_i(y) ≠ cv(y)
35     return abort(T_i,x)
36   rc_i(x) ← rc_i(x) + 1
37   // Return bufferred value.
38   return buf_i(x)
39 }
40 proc write(Transaction T_i, Variable x, Value v) {
41   // Abort on inconsistent view.
42   if ∃y: rv_i(y) ≠ cv(y)
43     return abort(T_i)
44   // Write to buffer.
45   buf_i(x) ← v
46   wc_i(x) ← wc_i(x) + 1
47   // Asynchronous release on last write.
48   if wc_i(x) = wub_i(x)
49     if T_i ∈ ℝ
50       async run :write_buffer(T_i,x)
51         when pv_i(x) - 1 = ltv(x)
52     else
53       async run :write_buffer(T_i,x)
54         when pv_i(x) - 1 = lv(x)
55   return ok_i
56 }
57 proc :read_buffer(Transaction T_i, Variable x) {
58   // Buffer and release a read−only variable.
59   rv_i(x) ← cv(x)
60   buf_i(x) ← x
61   :release(T_i,x)
62   async run :read_commit(T_i, x)
63     when pv_i(x) - 1 = ltv(x)
64 }
65 proc :read_commit(Transaction T_i, Variable x) {
66   // Commit a read−only variable early.
67   if ∃y: rv_i(y) > cv(y)
68     return abort(T_i)
69   ltv(x) ← pv_i(x)
70 }
71 proc :write_buffer(Transaction T_i, Variable x) {
72   if st_i(x) = ⊥
73     :checkpoint(T_i,x)
74   if ∃y: rv_i(y) ≠ cv(y)
75     return abort(T_i)
76   x ← buf_i(x)
77   :release(T_i,x)
78 }
```

```
79 proc commit(Transaction T_i) {
80   for x ∈ ASet_i {
81     // Synchronize with extant read thread.
82     if wub_i(x) = 0
83       join with :read_comit(T_i,x)
84     else {
85       // If released, synchronize with write thread.
86       if wc_i(x) = wub_i(x)
87         join with :write_buffer(T_i,x)
88       else {
89         // Catch up: get access and update variable.
90         wait until pv_i(x) - 1 = lv(x)
91         if st_i(x) = ⊥
92           :checkpoint(T_i, x)
93         if ∃y: rv_i(y) ≠ cv(y)
94           return abort(T_i)
95         if wc_i(x) > 0
96           x ← buf_i(x)
97       }
98       // Maintain commitment order.
99       wait until pv_i(x) - 1 = ltv(x)
100      :dismiss(T, x)
101    }
102  }
103  // Abort on inconsistent view.
104  if ∃y: rv_i(y) > cv(y)
105    return abort(T_i)
106  for x ∈ ASet_i
107    ltv(x) ← pv_i(x)
108  return C_i
109 }
110 proc abort(Transaction T_i) {
111   for(x ∈ ASet_i) {
112     // Maintain commitment order.
113     wait until pv_i(x) - 1 = ltv(x)
114     // Restore if consistent backup and modified.
115     if (wc_i(x) > 0 and pv_i(x) - 1 > lv(x)
116         and rv_i(x) = cv(x)) {
117       if wc_i(x) = wub_i(x)
118         join with :write_buffer(T_i,x)
119       :recover(T_i, x)
120     }
121     :dismiss(T_i, x)
122     ltv(x) ← pv_i(x)
123   }
124   return A_i
125 }
126 proc :dismiss(Transaction T_i, Variable x) {
127   if pv_i(x) - 1 = lv(x)
128     lv(x) ← pv_i(x)
129   if (wc_i(x) + rc_i(x)> 0 and rv_i(x) = cv(x)
130       and pv_i(x) - 1 > lv(x))
131     cv(x) ← pv_i(x)
132 }
133 proc :checkpoint(Transaction T_i, Variable x) {
134   st_i(x) ← x
135   rv_i(x) ← cv(x)
136 }
137 proc :recover(Transaction T_i, Variable x) {
138   x ← st_i(x)
139   cv(x) ← rv_i(x)
140 }
141 proc :release(Transaction T_i, Variable x) {
142   cv(x) ← pv_i(x)
143   lv(x) ← pv_i(x)
144 }
```

Figure 6.15: ROptSVA+R.

```
1  proc start(Transaction T_i) {
2    // Acquire private versions.
3    for x ∈ ASet_i in order
4      lock lk(x) → W
5    for x ∈ ASet_i {
6      gv(x) ← gv(x) + 1
7      pv_i(x) ← gv(x)
8      unlock lk(x)
9    }
10   // Asynchronously buffer read−only variables.
11   for x ∈ ASet_i: wub_i(x) = 0
12     async run :read_buffer(T_i, x)
13       when pv_i(x) - 1 = lv(x)
14   return ok_i
15 }
16 proc read(Transaction T_i, Variable x) {
17   // Wait for read−only variable to be bufferred.
18   if wub_i(x) = 0
19     join with :read_buffer(T_i, x)
20   // Copy value of variable to buffer on first read.
21   else if wc_i(x) = 0 and rc_i(x) = 0 {
22     wait until pv_i(x) - 1 = lv(x)
23     buf_i(x) ← x
24   }
25   rc_i(x) ← rc_i(x) + 1
26   // Return bufferred value.
27   return buf_i(x)
28 }
29 proc :read_buffer(Transaction T_i, Variable x) {
30   // Buffer and release a read−only variable.
31   buf_i(x) ← x
32   :release(T_i,x)
33   async run :read_commit(T_i, x)
34     when pv_i(x) - 1 = ltv(x)
35 }
36 proc :read_commit(Transaction T_i, Variable x) {
37   ltv(x) ← pv_i(x)
38 }
39 proc :write_buffer(Transaction T_i, Variable x) {
40   x ← buf_i(x)
41   :release(T_i,x)
42 }
43 proc write(Transaction T_i, Variable x, Value v) {
44   // Write to buffer.
45   buf_i(x) ← v
46   wc_i(x) ← wc_i(x) + 1
47   // Asynchronous release on last write.
48   if wc_i(x) = wub_i(x)
49     async run :write_buffer(T_i,x)
50       when pv_i(x) - 1 = lv(x)
51   return ok_i
52 }
53 proc commit(Transaction T_i) {
54   for x ∈ ASet_i {
55     // Synchronize with extant read thread.
56     if wub_i(x) = 0
57       join with :read_comit(T_i,x)
58     else {
59       // If released, synchronize with write thread.
60       if wc_i(x) = wub_i(x)
61         join with :write_buffer(T_i,x)
62       else {
63         // Catch up: get access and update variable.
64         wait until pv_i(x) - 1 = lv(x)
65         if wc_i(x) > 0
66           x ← buf_i(x)
67       }
68       // Maintain commitment order.
69       wait until pv_i(x) - 1 = ltv(x)
70       if pv_i(x) - 1 = lv(x)
71       :release(T, x)
72     }
73     ltv(x) ← pv_i(x)
74   }
75 }
76 proc :release(Transaction T_i, Variable x) {
77   lv(x) ← pv_i(x)
78 }
```

Figure 6.16: An abort-free variant of OptSVA.

6.3.8 Commit-only Model

OptSVA+R is designed to operate in the arbitrary abort model. We submit that this model is the most practical for distributed applications, where partial failures may necessitate that some transaction perform an abort in order to maintain the consistency of the system as a whole. However, the abort mechanism introduces a lot of additional complexity into OptSVA+R. Since the algorithm is pessimistic, the aborts are not necessary for the purpose of concurrency control, and as such the mechanisms for restoring variables after an abort can be excluded from it completely if the abort operation is missing from the transactional API.

Hence, we introduce an abort-free variant of OptSVA for the commit-only system model, where the transaction cannot issue a *tryA* operation. This allows us to remove the checkpoint and forced abort mechanisms completely (see Sections 6.2.1 and 6.2.2 respectively). We present the algorithm in Fig. 6.16.

The algorithm is last-use opaque by analogy to OptSVA+R. We conjecture that it also produces histories equivalent to opaque by analogy to SVA. Finally, transactions never abort or deadlock.

6.4 OptSVA in Control Flow Distributed TM

Initially, versioning algorithms were conceived as operation-type agnostic, which made them suitable for use with heterogeneous objects used in the CF model. These objects have arbitrary interfaces, whose operations (methods) execute arbitrary computations on state that can be composed of multiple discrete variables (fields). Such operations may not be limited to reading or writing the state of the object, but may do both, or neither, or cause side-effects in the process. Furthermore, each object may have a different interface. It is therefore practical to treat such objects as black boxes with respect to their state and operations they execute.

Contrast this to variables, used commonly in TM, where each object has a single read operation that reads the state of the object, and a single write operation which supersedes the previous state of the object with a new state. Both operations are simple, completely transparent, and contain no side effects, which allows to better orchestrate their execution.

An apt example is that of operation *locality*. According to [33], a *local read* is a read that is preceded during transaction execution by a write on the same shared variable—because it only depends on state written by that write operation, so it does not depend on what other transactions write. A *local write* is a write that is followed by a write on the same variable—because whatever that first write writes is superseded by the value written by the second. Local operation executions do not impact the system outside of their transaction. Thus, buffering can be used to make them invisible to the outside world. A local write modifies a transaction-local buffer, rather than the actual object. This means that local writes *de facto* do not operate on shared objects, so they do not need to pass the access condition to be executed. As we showed with OptSVA+R, using such optimizations with this model means that transactions execute more in parallel, and produce tighter schedules as a result, which improves system throughput.

On the other hand, the simplification of heterogeneous objects to variables restricts the flexibility of such a system model and limits its applicability in distributed systems. This is especially true in the CF model, where a complex object can be used not only to store and retrieve data, but also to delegate more involved, possibly long-running computations to a remote host. Once the arbitrary nature of interfaces and operation semantics is removed, the latter is lost and the system model loses its expressiveness, as well as the room for optimization.

Hence, in this section we introduce the distinction between read and write operations in the heterogeneous object model with arbitrary interfaces. We then list explain how OptSVA-CF+R extends the buffering system used in OptSVA+R to accommodate heterogeneous object operation executions. Next, we show how the optimizations introduced in OptSVA+R are lifted into the new model, and discuss the changes between OptSVA+R and OptSVA-CF+R executions. Finally, we summarize OptSVA-CF+R in full and discuss its properties as well as variants with irrevocable operation support. The algorithm was introduced and implemented in [82].

6.4.1 Heterogeneous Objects

OptSVA-CF+R requires that given some object $\lceil x \rfloor$ each operation $\mathtt{m} \in \mathtt{M}_{\lceil x \rfloor}$ be classified as one of the following:

a) a *read* operation is any operation that executes any code (including code with side effects) and may read the shared object's state and return a value, but during

execution the state is never modified,

b) a *write* operation is any operation that executes any code and may modify the state of the shared object, but the state is not read and a value is not returned,

c) an *update* operation is any operation that executes any code and may both modify and read the object's state and return a value.

This classification allows us to mimic the optimizations used within the variable model when synchronizing operations in the heterogeneous object model, but without knowing the details of each operation's semantics. We introduce the update operation, because we expect a typical operation on a complex object to modify its state based on the existing state of the object, hence to behave both like a read and write. However, such an operation is difficult to make invisible and parallelize. On the other hand, "pure" writes, can be expected to be rare, but they do not need to view the state to execute, so more optimizations apply to them. Specifically, they can also be made to execute on an "empty" buffer without prior synchronization, just like writes in OptSVA. Thus, we keep them apart from updates. Note that the complex shared object may still contain composite state, consisting of some number of independent variables, and read, write, and update operations are not required to read and/or modify the state holistically. Whether or not a particular operation will only read state written locally or whether it requires synchronization depends largely on how objects buffering is implemented.

Not also that the operation types presented here can be used with reference to homogeneous objects. A read operation in the heterogeneous object model is equivalent with a read operation in the homogeneous model, and either the update or a write operations may be selected to represent a write operation in the homogeneous object, depending on the semantics of the operation.

6.4.2 Buffering

When creating buffers for variable-like objects, given the semantics of the two available operations, it is simply a matter of copying a value from a shared variable to some local variable. Such a buffer can also be locally written to without knowledge of its state, since the new value of the variable supersedes the old. Thus, local writes can simply write to uninitialized local variables.

Given the composite state of complex objects and arbitrary semantics of operations, two types of buffers are needed. The first, a *copy buffer*, is one that copies the entire state of a shared object, and can be used to both locally read and modify the object. Such a buffer can be used to read a released object or restore an object during abort. For the first purpose we use a buffer denoted $\texttt{buf}_i(\lceil x \rfloor)$ (for some transaction T_i and some object $\lceil x \rfloor$), and for the latter we use a buffer denoted $\texttt{st}_i(\lceil x \rfloor)$. However, since the state of the original object is copied, in order to create a copy buffer the transaction must check fulfill the access condition before doing so. Such a copy buffer is not universal, since it cannot be used to execute local writes without prior synchronization.

Thus, we introduce a second buffer type. A *log buffer* is an object that maintains the interface of the original shared object but none of its state. When a method is executed on the object, the buffer logs the method and its parameters. The method may be executed completely, assuming that it does not need any state other than local data to do so. In that case, any changes the method does to the state are tracked and stored. If this is impossible, the method will not execute, apart from being logged. The log buffer can be applied to the original object to update the state of the latter. If some method was pre-executed before applying the log, its effects are applied to the state of the original object. If a method was not previously executed, it is executed on the original object at the time

the log is being applied. Given the log buffer does not use the object's state, it can be used to execute write operations without prior synchronization. Since write operations modify the object's state without viewing it, write operations are always capable of executing methods on the log buffer in place, and do not need to commute the execution to the point when the buffer is applied.

Since the CF semantics require that computations are performed wherever the shared object is located in the distributed system, either type of buffer resides on the same host in a distributed system as the original object. Otherwise, not only would the assumptions of the CF model be violated, but if the execution of operations caused any side effects, the side effects would be removed from the location of the original node.

6.4.3 Asynchronous Buffering

If a transaction only ever executes read operations on a shared object (although it may execute writes and updates on other objects), we will refer to such an object as a *read-only object* with respect to this transaction. OptSVA-CF+R handles read-only objects similarly to how OptSVA+R handles variables. In the case of such an object, synchronization needs to be done when the first read is executed, but all subsequent reads only need to use the buffer to execute. Hence, once the buffer is created, the reads execute as if they were local and the read-only object can be released. Other transactions benefit from the object being released as soon as possible, and it is possible for a read-only object to be released even before the first read operation occurs. I.e., first write does not need to access the actual object either, as long as the state of the object is buffered. The only condition that must be satisfied to store the state of some object in the buffer is that it must pass the access condition, but otherwise it can be done at any point in the transaction. Hence, OptSVA-CF+R transactions attempt to buffer a read-only object as soon as they retrieve private versions at start. But, since waiting at the access condition may block the executing of operations that precede the first read on the read-only object in the code of the transaction, the buffering procedure is executed asynchronously: the transaction delegates it to a separate thread and proceeds to execute other operations as normal. The separate thread waits until the access condition for the object is met, following which the thread buffers, and immediately releases the object. Then, all reads, including the first read, execute the operation on the buffer. In effect, early release of read-only objects is potentially expedited.

Similar asynchrony is used in the case of a final modification of an object. The procedure is more complex and more conservative in OptSVA-CF+R than in OptSVA+R, since there are more operation and buffer types. When a transaction executes its last write or update operation on some shared object, the object is immediately buffered afterward and released. This allows all following read operations to only use the buffer, and therefore be invisible to outside transactions. The final update can only be executed if the access condition is passed, since it may need to view the state of the object to execute. However, a write may execute using the log buffer instead and do it without synchronization, since it does not view the state. Then, in the specific case of a write operation that is the only write operation on an object, or in case of a write operation preceded only by other write operations on that object, the transaction may not have attempted to satisfy the access condition yet. In such a case, the final write can be split into a write that executes using the log buffer without synchronization, and a procedure that subsequently updates the state of the actual object. This procedure can only be executed if the access condition is passed, but it can release the object immediately after it finishes updating the object's state. The procedure is executed asynchronously with respect to the main body of the transaction, since it has no impact on following

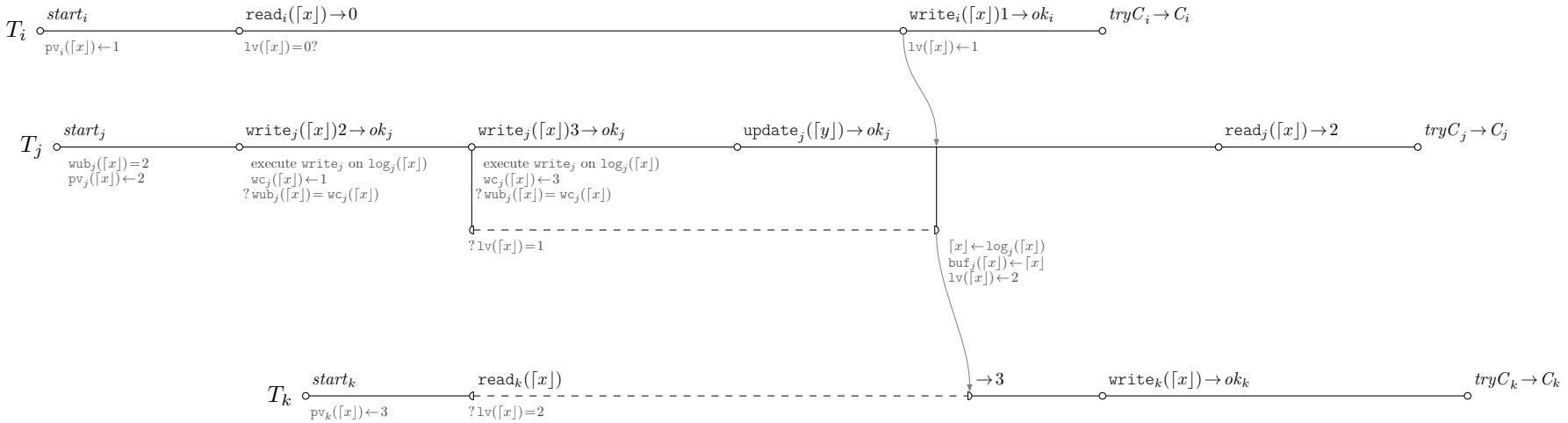


Figure 6.17: Asynchronous release (OptSVA-CF+R).

operations—all following operations on this object, if any, will be reads, and will read from the local buffer. In this way, the last write avoids blocking the entire transaction to wait for the access condition. In addition, the point at which the transaction must wait for the access condition for this object can be delayed to any point between the last write and the end of the transaction.

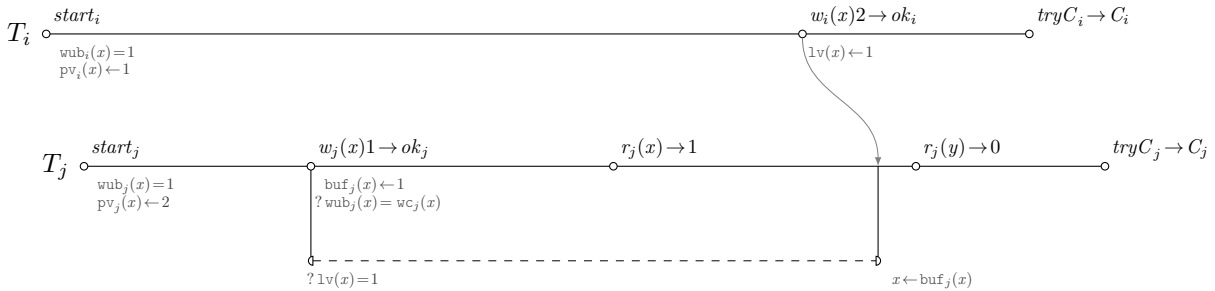
We illustrate this optimization further in Fig. 6.17. Here, transaction T_i can pass access condition for $\lceil x \rfloor$ first and execute a read and a write on $\lceil x \rfloor$. Nevertheless T_j performs operations on $\lceil x \rfloor$ simultaneously. First, T_j executes a write, and can do so without waiting at the access condition, since it works on the log buffer rather than directly on $\lceil x \rfloor$. Meanwhile T_i can execute operations on the actual object. Then T_j executes another write operation, using the log buffer. Since this is the last operation execution on $\lceil x \rfloor$ in T_j (which the transaction knows because $\mathtt{wub}_j(\lceil x \rfloor) = \mathtt{wc}_j(\lceil x \rfloor)$), T_j may write the changes from the log buffer into the object. Hence, a separate thread starts at the end of the write operation and it starts waiting at the access condition. When the access condition is satisfied, the thread updates the state of $\lceil x \rfloor$ using the log buffer and releases it, which allows T_k to start accessing $\lceil x \rfloor$. The thread also creates a copy buffer from the updated object, which is sufficient for future local reads to use. Note, however, that T_j can immediately start doing other operations, while the separate thread is still waiting at the access condition to $\lceil x \rfloor$. Hence T_j can execute an update operation on $\lceil y \rfloor$, which can be executed regardless of the access condition to $\lceil x \rfloor$. Then, T_j can continues to execute read operations on $\lceil x \rfloor$ using the copy buffer, and does so in parallel to T_k. If buffers were not used, none of these operations could be executed in parallel by several transactions. In addition, if a separate thread were not used to synchronize and release $\lceil x \rfloor$, but if instead this were done as part of the last write, the operation on $\lceil y \rfloor$ in T_j would be significantly, but needlessly delayed.

6.4.4 Consequences of Model Generalization

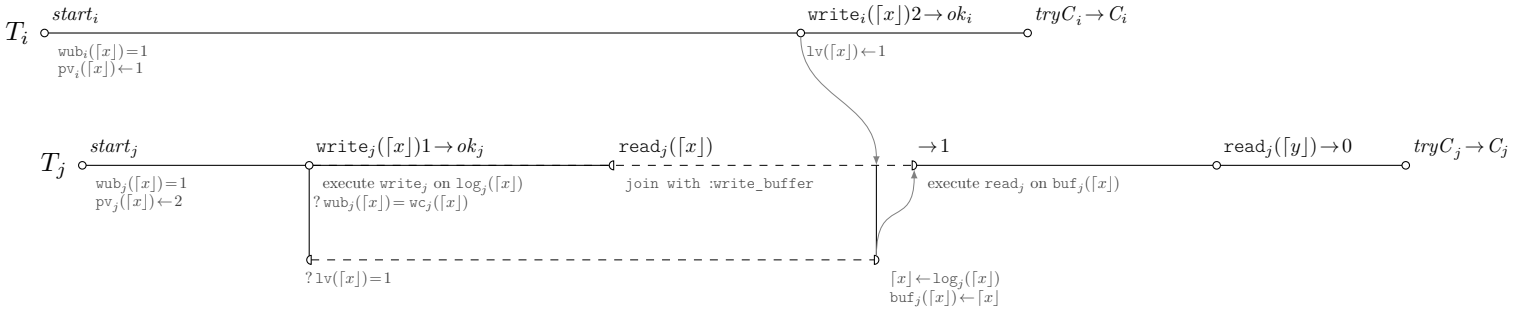
OptSVA-CF+R is based on the optimizations introduced in OptSVA+R, but applies them to a different, more universal system model. The complex object model is more general, since a variable-like object can be implemented as a reference cell, a complex object with one field, a read operation returning its value, and a write operation setting the old value to a new one.

Given such a specification, OptSVA-CF+R will execute the same way as OptSVA+R with one significant exception. Given a transaction that executes a write operation on some object $\lceil x \rfloor$ followed by a read operation on $\lceil x \rfloor$, OptSVA-CF+R will execute the write without synchronization, but must synchronize before the read executes. This is because the changes in the log buffer must be applied to the actual object and copy buffer before the read proceeds. Hence, the read might be forced to wait until the access condition for $\lceil x \rfloor$ is satisfied. In contrast, OptSVA+R will allow the read to proceed without synchronization, since it is a local operation, and therefore completely dependent on the preceding write.

We show an example of this in Fig. 6.18. The figure shows the execution of the same transactional code under the same circumstances using OptSVA+R in Fig. 6.18a and OptSVA-CF+R in Fig. 6.18b. The object $\lceil x \rfloor$ is meant to represent a reference cell, a simple object with effectively the same interface and semantics as x. In both histories T_i starts first, but executes a write operation writing 2 to x (or $\lceil x \rfloor$) much later. In the mean time T_j executes its own write to x ($\lceil x \rfloor$), writing 1 to it. Since this is the initial write OptSVA+R executes it on buffer $\mathtt{buf}_j(x)$ and OptSVA-CF+R executes it on log buffer $\mathtt{log}_j(\lceil x \rfloor)$, so in neither algorithm is T_j forced to wait for T_i. Then, T_j executes a read operation on x (or $\lceil x \rfloor$). In OptSVA+R the read operation can proceed without waiting



(a) OptSVA+R.



(b) OptSVA-CF+R.

Figure 6.18: Local read operation handling in OptSVA-CF+R and OptSVA+R.

since it is executed using $\texttt{buf}_j(x)$, but in OptSVA-CF+R the read cannot proceed, since the read cannot be executed on $\texttt{log}_j(\lceil x \rfloor)$. Instead, in OptSVA-CF+R, T_j must first synchronize with T_i to retrieve a consistent version of $\lceil x \rfloor$ on which it can apply the log buffer. Only then can T_j finish reading from $\lceil x \rfloor$ in OptSVA-CF+R. Therefore, T_j in OptSVA-CF+R waits longer than in OptSVA+R.

In effect, we can see that given reference cells, there are certain executions that will be allowed by OptSVA+R that are not allowed by OptSVA-CF+R. Since these OptSVA+R executions are tighter than their equivalent executions in OptSVA-CF+R, OptSVA+R admits a higher level of parallelism. Therefore, OptSVA-CF+R trades generality for performance.

6.4.5 Summary

OptSVA-CF+R operates on the basis of the versioning mechanism, using private, local, and local terminal version counters to ensure that accesses to objects and commits are performed in the order defined by private versions. The individual operations are handled as follows:

Start

When an OptSVA-CF+R transaction T_i starts it acquires a private version for each shared object in its access set. If any of these objects are read-only with respect to T_i, the transaction also starts separate threads that clone the objects into copy buffers $\texttt{buf}_i(\lceil x \rfloor)$ and release them afterward.

Read

Whenever transaction T_i attempts to execute a read operation on some object $\lceil x \rfloor$, its behavior primarily depends on whether the object is read-only, and whether it was released or not. If the object is read-only with respect to T_i, the read operation waits until the separate read-only thread finishes buffering the object, and executes the read operation on the buffer.

Otherwise, if the object was not previously accessed, then the transaction checks if there were preceding reads or updates. If not, the transaction must wait until the access condition to $\lceil x \rfloor$ is satisfied and makes a checkpoint by copying the state of the shared object to buffer $\texttt{st}_i(\lceil x \rfloor)$. Buffer $\texttt{st}_i(\lceil x \rfloor)$ is a copy buffer like $\texttt{buf}_i(\lceil x \rfloor)$, but it is never modified and only used to restore the object in the event of abort.

In addition, if only preceding operations were writes, then they were performed using the log buffer $\texttt{log}_i(\lceil x \rfloor)$, so the transaction applies the log buffer to $\lceil x \rfloor$ before proceeding. Next, the transaction checks if any object was invalidated so far, and if so forcibly aborts. If any object was invalidated at any point, the transaction is doomed to abort eventually, so by checking for all the objects we force it to do so as early as we can detect. If the transaction is not aborted, it then executes the code of the read operation on $\lceil x \rfloor$. If this is the last operation (of any kind) on $\lceil x \rfloor$, the transaction subsequently releases $\lceil x \rfloor$.

If the object was previously released, the read waits until the thread responsible for releasing the object is finished. Once this is the case, the transaction executes the code of the read operation on $\lceil x \rfloor$ using the copy buffer $\texttt{buf}_i(\lceil x \rfloor)$ (created at release).

Update

In the case of an update operation, the transaction checks whether any reads or updates were executed on the same object before. If that is the case, the transaction waits until the

```
1  proc start(Transaction T_i) {
2    // Acquire private versions.
3    for ⌈x⌋ ∈ ASet_i in order
4      lock lk(⌈x⌋) → W
5    for ⌈x⌋ ∈ ASet_i {
6      gv(⌈x⌋) ← gv(⌈x⌋) + 1
7      pv_i(⌈x⌋) ← gv(⌈x⌋)
8      unlock lk(⌈x⌋)
9    }
10   // Asynchronously buffer read−only variables.
11   for ⌈x⌋ ∈ ASet_i: wub_i(⌈x⌋) = 0
12     async run :read_buffer(T_i, ⌈x⌋)
13       when pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
14   return ok_i
15 }
16 proc read(Transaction T_i, Object ⌈x⌋, Method m) {
17   // Read−only object.
18   if ⌈x⌋ is read-only {
19     join with :read_buffer(T_i,⌈x⌋)
20     if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
21       return abort(T_i)
22     execute m on buf_i(⌈x⌋) returning v
23     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
24     return v
25   }
26   // Object not previously released.
27   if (wc_i(⌈x⌋) < wub_i(⌈x⌋)
28       or uc_i(⌈x⌋) < uub_i(⌈x⌋)) {
29     if wc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
30       wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
31       :checkpoint(T_i,⌈x⌋)
32       if (wc_i(⌈x⌋) > 0)
33         apply log_i(⌈x⌋) to ⌈x⌋
34     }
35     if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
36       return abort(T_i)
37     execute m on ⌈x⌋ returning v
38     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
39     if (rc_i(⌈x⌋) = rub_i(⌈x⌋)
40         and wc_i(⌈x⌋) = wub_i(⌈x⌋)
41         and uc_i(⌈x⌋) = uub_i(⌈x⌋))
42       :release(T_i,⌈x⌋)
43     return v
44   }
45   // Object previously released.
46   if wc_i(⌈x⌋) = wub_i(⌈x⌋)
47       and uc_i(⌈x⌋) = uub_i(⌈x⌋)) {
48     if write_buffer(T_i,⌈x⌋) is running
49       join with :write_buffer(T_i,⌈x⌋)
50     if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
51       return abort(T_i)
52     execute m on buf_i(⌈x⌋) returning v
53     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
54     return v
55   }
56 }
57 proc update(Transaction T_i, Object ⌈x⌋, Method m) {
58   if rc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
59     wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
60     :checkpoint(T_i,⌈x⌋)
61     if (wc_i(⌈x⌋) > 0
62       apply log_i(⌈x⌋) to ⌈x⌋
63   }
64   if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
65     return abort(T_i)
66   execute m on ⌈x⌋ returning v
67   uc_i(⌈x⌋) ← uc_i(⌈x⌋) + 1
68   if (wub_i(⌈x⌋) = wc_i(⌈x⌋)
69       and uub_i(⌈x⌋) = uc_i(⌈x⌋)) {
70     buf_i(⌈x⌋) ← ⌈x⌋
71     :release(T_i,⌈x⌋)
72   }
73   return v
74 }
75 proc write(Transaction T_i, Object ⌈x⌋, Method m) {
76   // No preceding reads or updates.
77   if rc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
78       execute m on log_i(⌈x⌋)
79     wc_i(⌈x⌋) ← wc_i(⌈x⌋) + 1
80     if wc_i(⌈x⌋) = wub_i(⌈x⌋)
81       async run :write_buffer(T_i,⌈x⌋)
82         when pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
83   }
84   // Some preceeding reads or updates.
85   if rc_i(⌈x⌋) > 0 or uc_i(⌈x⌋) > 0 {
86     if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
87       return abort(T_i)
88     execute m on ⌈x⌋
89     wc_i(⌈x⌋) ← wc_i(⌈x⌋) + 1
90     if wc_i(⌈x⌋) = wub_i(⌈x⌋) {
91       buf_i(⌈x⌋) ← ⌈x⌋
92       :release(T_i, ⌈x⌋)
93     }
94   }
95 }
96 proc commit(Transaction T_i) {
97   for(⌈x⌋ ∈ ASet_i) {
98     if wub_i(⌈x⌋) = 0
99       join with read_comit(T_i,⌈x⌋)
100    else {
101      if (wc_i(⌈x⌋) = wub_i(⌈x⌋)
102          and rc_i(⌈x⌋) = uc_i(⌈x⌋) = 0)
103        join with write_buffer(T_i,⌈x⌋)
104      else {
105        if wc_i(⌈x⌋) + rc_i(⌈x⌋) = uc_i(⌈x⌋) = 0
106          wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
107        if (wc_i(⌈x⌋) > 0
108            and rc_i(⌈x⌋) = uc_i(⌈x⌋) = 0) {
109          :checkpoint(T_i,⌈x⌋)
110          if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
111            return abort(T_i)
112          apply log_i(⌈x⌋) to ⌈x⌋
113        }
114      }
115      wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
116      if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
117        lv(⌈x⌋) ← pv_i(⌈x⌋)
118      if (rc_i(⌈x⌋) + wc_i(⌈x⌋) + uc_i(⌈x⌋) > 0
119          and rv_i(⌈x⌋) = cv(⌈x⌋)
120          and pv_i(⌈x⌋) - 1 > lv(⌈x⌋))
121        cv(⌈x⌋) ← pv_i(⌈x⌋)
122    }
123  }
124  if ∃⌈y⌋: rv_i(⌈y⌋) > cv(⌈y⌋)
125    return abort(T_i)
126  for ⌈x⌋ ∈ ASet_i
127    ltv(⌈x⌋) ← pv_i(⌈x⌋)
128  return ok_i
129 }
130 proc abort(Transaction T_i) {
131   for ⌈x⌋ ∈ ASet_i {
132     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
133     if (rc_i(⌈x⌋) + wc_i(⌈x⌋) + uc_i(⌈x⌋) > 0
134         and pv_i(⌈x⌋) - 1 > lv(⌈x⌋)
135         and rv_i(⌈x⌋) = cv(⌈x⌋)
136         and wub_i(⌈x⌋) + uub_i(⌈x⌋) > 0) {
137       if wc_i(⌈x⌋) = wub_i(⌈x⌋)
138         join with :write_buffer(T_i,⌈x⌋)
139       :recover(T_i,⌈x⌋)
140     }
141     if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
142       lv(⌈x⌋) ← pv_i(⌈x⌋)
143     ltv(⌈x⌋) ← pv_i(⌈x⌋)
144   }
145   return A_i
146 }
```

Figure 6.19: OptSVA-CF+R.

```
147 proc :read_buffer(Transaction T_i, Object ⌈x⌋) {
148   rv_i(⌈x⌋) ← cv(⌈x⌋)
149   buf_i(⌈x⌋) ← ⌈x⌋
150   :release(T_i,⌈x⌋)
151   async run :read_commit(T_i,⌈x⌋)
152     when pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
153 }
154 proc :read_commit(Transaction T_i, Object ⌈x⌋) {
155   if ∃⌈y⌋: rv_i(⌈y⌋) > cv(⌈y⌋)
156     return abort(T_i)
157   ltv(⌈x⌋) ← pv_i(⌈x⌋)
158 }
159 proc :write_buffer(Transaction T_i, Object ⌈x⌋) {
160   :checkpoint(T_i,⌈x⌋)
161   apply log_i(⌈x⌋) to ⌈x⌋
162   buf_i(⌈x⌋) ← ⌈x⌋
163   :release(T_i,⌈x⌋)
164 }
```

```
165 proc :checkpoint(Transaction T_i, Object ⌈x⌋) {
166   st_i(⌈x⌋) ← ⌈x⌋
167   rv_i(⌈x⌋) ← cv(⌈x⌋)
168 }
169 proc :recover(Transaction T_i, Object ⌈x⌋) {
170   ⌈x⌋ ← st_i(⌈x⌋)
171   cv(⌈x⌋) ← rv_i(⌈x⌋)
172 }
173 proc :release(Transaction T_i, Object ⌈x⌋) {
174   cv(⌈x⌋) ← pv_i(⌈x⌋)
175   lv(⌈x⌋) ← pv_i(⌈x⌋)
176 }
```

Figure 6.19: OptSVA-CF+R.

access condition is satisfied and then makes a checkpoint. In addition, the transaction will also apply the log buffer $\texttt{log}_i(\lceil x \rfloor)$ to $\lceil x \rfloor$ if there were preceding writes (but no preceding reads or updates). In any case, the transaction subsequently checks whether any objects were invalidated, and aborts if that is the case. Afterward, the code of the operation is executed on $\lceil x \rfloor$. If there are no further updates or writes to be performed on $\lceil x \rfloor$, the transaction makes a copy of $\lceil x \rfloor$ in $\texttt{buf}_i(\lceil x \rfloor)$ and releases it.

Write

Pure write operations are executed in one of two ways, depending on whether there were any read or update operations executed prior (by the same transaction). This is because updates and reads both wait on the access condition, meaning that then the object can be operated on directly. Otherwise, the write can be performed using a log buffer. Specifically, if there were no preceding reads or updates, the transaction simply executes the operation on the log buffer. If this is the final write and there will also not be update operations on this object in the transaction, the transaction then starts a thread, which will wait at the access condition and subsequently: make a checkpoint to $\texttt{st}_i(\lceil x \rfloor)$, apply the log buffer $\texttt{log}_i(\lceil x \rfloor)$ to the original object $\lceil x \rfloor$, copy the modified object to the copy buffer $\texttt{buf}_i(\lceil x \rfloor)$, and release $\lceil x \rfloor$. Meanwhile, the transaction's main thread proceeds.

If there were preceding reads or updates, the transaction operates using the up-to-date object that is already under its control. Making a checkpoint would be redundant, but the transaction checks whether any objects were invalidated, and if so, aborts. Otherwise, it executes the code of the operation on the object, and if this was the last write or update operation on $\lceil x \rfloor$, then $\lceil x \rfloor$ is cloned to $\texttt{st}_i(\lceil x \rfloor)$ and released. The last step is not done in a separate thread, since the transaction already has access to $\lceil x \rfloor$.

Commit

When the transaction commits it waits for extant threads to finish in the case such threads are still running for read-only objects and objects that are being released after last write. Afterward, the transaction waits until the commit condition is satisfied for all objects in its access set. Then, if the transaction did not access a particular object at any time, it makes a checkpoint. If it only ever executed writes on an object, the transaction applies the log buffer to the object. If the object was not released, the transaction releases it. Afterward, the transaction checks whether any object was invalidated, and aborts if that is the case. Otherwise, the transaction updates the local terminal versions of all

objects and finishes execution. No further actions may be performed by the transaction after the commit finishes executing.

Abort

When the transaction aborts, just like with commit, it waits for the appropriate threads to finish, and for the commit condition to be satisfied. Then, each object in the transaction's access set is restored from $\mathtt{st}_i(\lceil x \rfloor)$, unless some other transaction that previously aborted already restored it to an older version beforehand. Then, the transaction updates the local terminal versions of all objects and finishes execution. No further actions may be performed by the transaction after the abort finishes executing.

6.4.6 Properties

OptSVA-CF+R is last-use opaque, which we discuss in Chapter 7. We briefly demonstrate the liveness, and progress properties of OptSVA-CF+R below.

Theorem 3. *OptSVA-CF+R is deadlock-free.*

Proof. There are two types of occurrence where an operation can wait. The first is waiting on an access condition, or the similar condition when a transaction attempts to commit or abort. In this case, the condition is satisfied in the order enforced by transactions' private versions for specific objects. Since private versions are consecutive integers and since they are acquired atomically by the transaction, it is impossible for a circular wait to occur. The other case of waiting is during transaction start, when private versions are acquired. In order for this to be done atomically, transactions lock a series of locks before getting private versions, and release the locks afterward. These locks are always acquired in accordance to an arbitrary global order, regardless of transaction. That eliminates the possibility that a circular wait occurs during start. Since circular wait cannot occur among transactions, OptSVA-CF+R is deadlock free.

□

Theorem 4. *OptSVA-CF+R is strongly progressive.*

Proof. Any transaction in OptSVA-CF+R can either abort manually or forcibly. In order for a transaction T_i to abort forcibly, there must be some transaction T_j that forces T_i to abort, i.e., such T_j that accessed some object $\lceil x \rfloor$ and released it before T_i accessed $\lceil x \rfloor$, and T_j must have aborted after T_i accessed $\lceil x \rfloor$. Thus for every forcibly aborted transaction, there must be another aborted transaction. Hence, given any set of conflicting transactions, there will be at least one transaction that will not be forcibly aborted (but it will be manually aborted). Therefore, OptSVA-CF+R is strongly progressive [33]. □

6.4.7 Reluctant Transactions

The versioning mechanism is pessimistic: it delays operations rather then aborting transactions on conflict. Transactions only abort if the abort operation is invoked programmatically, or as a result of a cascade. Further, cascading aborts start only due to a transaction being aborted manually. Hence, if no transaction in the system manually aborts, no transaction ever aborts. Then, it is to execute any irrevocable operations within any transaction. However, if any transaction manually aborts, it is possible that it will force some other transaction into a cascade. In order for transaction T_i to participate in a cascading abort, a preceding transaction T_j must release an object early and then abort after the T_i executed an operation directly on that object. These conditions rarely occur

in practice, so cascading aborts are also infrequent. However, they do introduce a chance of unsafe executions of irrevocable operations.

Hence, by analogy to RSVA+R and ROptSVA+R we introduce ROptSVA-CF+R. In this variant of the algorithm in order to exclude the possibility of abort completely for transactions with irrevocable operations, such transactions can be labeled reluctant. ROptSVA-CF+R prevents reluctant transactions from ever becoming part of a cascade, by replacing all access condition checks with termination condition checks. This means that irrevocable transactions never "accept" objects released early. The drawback is that such transactions may wait longer to access shared objects, but in return they never forcibly abort.

We give the pseudocode of ROptSVA+R in Appendix B. It is almost identical to OptSVA-CF+R.

6.4.8 Commit-only Model

It is also possible to derive a variant of OptSVA-CF that operates in the commit-only model. Given that OptSVA-CF is pessimistic, it does not need to abort unless arbitrary aborts are introduced, so OptSVA-CF in the commit only model is completely abort-free. This means that it becomes impractical for certain classes of distributed systems, but also completely safe for irrevocable operations. We give this algorithm in Appendix B in full. It is similar to OptSVA-CF+R but lacks mechanisms for aborting transactions and reverting objects to previous states, which makes it simpler.

6.5 Summary

We present a summary of the characteristics of versioning algorithms in Table 6.1. All of the algorithms are pessimistic and blocking, as well as deadlock-free, so we omit this from the table.

Using the original versioning algorithms as a base (primarily SVA), we introduced three classes of novel TM algorithms that aim to take advantage of the versioning concurrency control and early release mechanisms. First, we extended pessimistic algorithms into the arbitrary abort model. This allows versioning algorithms to be used in a broader range of systems, including in distributed systems with a possibility of partial failure. Since SVA uses early release, the introduction of the abort operation into SVA+R required the introduction of additional mechanism to contain inconsistent views and retain strong safety properties. These mechanisms are novel and are not needed for other pessimistic systems with the ability to abort (BVA+R, 2PL). However, the drawback of these algorithms is that they are agnostic of the semantics of the objects on which they execute operations, hence their performance falls short in comparison to traditional optimistic TM in systems where the semantics of operations are known. (We show this experimentally in Section 8.1.)

The second class of versioning algorithms are aimed to alleviate the problem by applying the versioning control mechanism in the variable model. OptSVA+R is the representative of that class that introduces a number of optimizations to the basic *modus operandi* of its predecessor: heavy use of buffering and commit-time updates rather than encounter-time modifications, read operation parallelization, early release of shared objects on last write instead of last operation of any kind, and transaction-local operation asynchrony which allows transactions to delegate some tasks that require waiting to separate thread and proceed with other computation in the mean time. These optimizations

Algorithm	**Updates**	**Aborts**	**A priori**	**Objects**	**Safety**	**Early release**	**Irrevocable**
BVA	encounter-time	commit-only, abort-free	*ASet*	heterogeneous	opaque	no	$T_i \in \mathbb{T}$
BVA+R	encounter-time	arbitrary abort	*ASet*	heterogeneous	opaque	no	$T_i \in \mathbb{T}$
SVA	encounter-time	commit-only, abort-free	*ASet*, `supr`	heterogeneous	opaque-equivalent	yes	$T_i \in \mathbb{T}$
SVA+R	encounter-time	arbitrary abort, cascade	*ASet*, `supr`	heterogeneous	last-use opaque	yes	$\varnothing$
RSVA+R	encounter-time	arbitrary abort, cascade	*ASet*, `supr`, $\mathbb{R}$	heterogeneous	last-use opaque	yes	$T_i \in \mathbb{R}$
OptSVA	commit-time	commit-only, abort-free	*ASet*, `wub`, `rub`,	variable	last-use opaque*	yes	$T_i \in \mathbb{T}$
OptSVA+R	commit-time	arbitrary abort, cascade	*ASet*, `wub`, `rub`,	variable	last-use opaque	yes	$\varnothing$
ROptSVA+R	commit-time	arbitrary abort, cascade	*ASet*, `wub`, `rub`, $\mathbb{R}$	variable	last-use opaque	yes	$T_i \in \mathbb{R}$
OptSVA-CF	commit-time	commit-only, abort-free	*ASet*, `wub`, `rub`, operation types	any	last-use opaque*	yes	$T_i \in \mathbb{T}$
OptSVA-CF+R	commit-time	arbitrary abort, cascade	*ASet*, `wub`, `rub`, operation types	any	last-use opaque	yes	$\varnothing$
ROptSVA-CF+R	commit-time	arbitrary abort, cascade	*ASet*, `wub`, `rub`, $\mathbb{R}$, operation types	any	last-use opaque	yes	$T_i \in \mathbb{R}$

Table 6.1: Summary comparison of versioning algorithms.

reduce the amount of scenarios where one transaction has to wait for another and in this way improve the transactional throughput, while maintaining the string safety properties of SVA+R. We show this improvement theoretically and conclude that OptSVA+R is an algorithm with high potential for parallelism. To the best of our knowledge OptSVA+R is the first algorithm to use transaction-local asynchrony to execute transparently execute transactional operations in parallel with local computations and operations on other variables.

However, the system model used in OptSVA+R is not as well-suited to practical applications in distributed TM. Hence, we introduce a third class of versioning algorithms which allow the generalization of the introduced algorithms to any system model, while maintaining a high degree of parallel execution of conflicting transactions. We show experimentally in Section 8.2 that an implementation of these algorithms can outperform quality optimistic distributed TM systems.

In each class of algorithms we introduce variants that prevent specific reluctant transactions from aborting, making them completely safe for irrevocable operations, and leaving the decision whether a particular transaction can be allowed to abort to the programmer, who can determine this based on business logic and the code of the transaction and decide whether it pays off to trade consistency for efficiency. We also include commit-only variants of the algorithms which can be applied to systems where programmers are not allowed to execute programmatic aborts, which makes the system completely abort free. We conjecture that such systems are equivalent to opaque systems in terms of safety in the commit-only model (we denote this in Table 6.1 by an asterisk). This also means that all transactions safely execute irrevocable operations.

7 Safety

In this chapter we discuss the safety properties of selected algorithms from Chapter 6. Specifically, in the first section we discuss the relationship between SVA and opacity. SVA is not opaque, since it admits histories with early release, which are forbidden by opacity. However, opacity prevents early release specifically to prevent inconsistent views, and SVA histories do not allow inconsistent views. This is because SVA is pessimistic and operates in the commit-only model, which means neither forced nor voluntary aborts occur. Hence, we attempt to draw an equivalency between histories produced by SVA and opaque histories, by showing the former are observationally indistinguishable from the latter. The technique presented in this section were introduced in [80].

Next, we discuss the safety of SVA+R and demonstrate that SVA+R is last-use opaque. We first provide a proof sketch showing the intuition and describing the method, and give the complete proof in Appendix A. The proof was originally presented in [79].

Then, we demonstrate the last-use opacity of OptSVA+R. Since OptSVA+R makes heavy use of buffering and largely detaches operations from their actual effects on memory, the proof is not straightforward. For this reason, we introduce *trace harmony*, a proof technique that allows to prove last-use opacity (and can be extended to opacity) of algorithms that operate on buffers. Trace harmony decomposes last-use opacity into several simpler criteria. If a history can be shown to satisfy those criteria, then it follows that it is last-use opaque (we demonstrate this formally in Appendix A). Hence, we show that OptSVA+R histories are harmonious, and therefore last-use opaque. The proof and trace harmony were presented in [102].

Finally, we show that the proof for last-use opacity of OptSVA-CF+R follows from the proof of OptSVA+R, after [82].

7.1 Opacity of SVA

Opacity introduces the requirement that transactions never read from live transactions, since this could lead to situations where inconsistent views cause unexpected and dangerous situations to occur, like infinite loops and division by zero errors. However opacity precludes early release, an important programming technique, where two transactions technically conflict but nevertheless both commit correctly, and still produce a history that is intuitively correct. This is particularly true with pessimistic concurrency control,

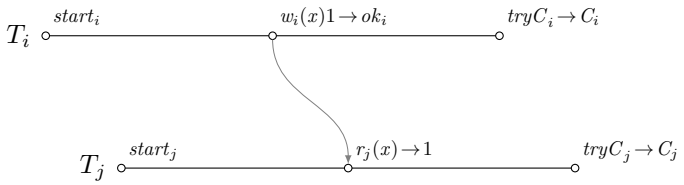


Figure 7.1: A history with early release and no inconsistent views.

where transactions, as a rule, do not abort. If they do not abort, then viewing the final state of a variable does not cause inconsistencies, even if the value is read from a live transaction. On the other hand, systems that employ early release gain a significant improvement in performance.

SVA is one such pessimistic concurrency control algorithm. SVA allows transactions to read from other live transactions under the condition that the live transactions being read from will not attempt to modify the objects in question. Since SVA transactions also never abort, then they never experience inconsistent views, and are otherwise free of all the dangerous situations opacity is meant to exclude. Nevertheless SVA does not meet the requirements of opacity.

Here, we present a technique that can show that opacity can be fulfilled by transforming the original history with early release to a different form called a *decomposed history*. The transformation can be performed only under stringent assumptions with respect to the original history, but the decomposed form can then be proven to be opaque. Since we also show that the decomposed history is a *refinement* of the original history, this suffices to acknowledge that the original history provides the same safety guarantees as opacity. In this way, we can show TM systems with early release need not necessarily relax consistency in trade for efficiency.

7.1.1 History Decomposition

In Section 3.2.6 we showed that a history with an instance of early release cannot be opaque. However, the history shown in Fig. 7.1 is an intuitively correct execution, since all operations are legal, the real-time order is preserved, and no inconsistent views are introduced. Indeed, the history is even final-state opaque and only its prefix created by removing both commit operation executions is not (proof in Appendix A).

This intuition that the history is correct would be especially true if history H_1 were generated by a pessimistic TM system or any system in the commit only model, where aborts do not occur. In the particular case of these systems, forming a completion by defaulting an execution of an uncommitted transaction to an abort is too conservative and leads to perfectly legal histories being unable to satisfy opacity. On the other hand, the assumption that all transactions will eventually commit is not one that can be incorporated into the definition of opacity without compromising its meaningfulness for optimistic TMs.

Therefore, rather than modifying the definition of opacity to allow for non-aborting pessimistic TMs with early release, in this section we propose a simple technique called *decomposition*. The technique allows to create a decomposed history by splitting transactions with early release into sequences of atomic single-operation transactions (given certain stringent assumptions about their execution). This history will prevent transactions with early release from violating the consistency requirements of opacity, but will nevertheless be commensurate with the original history on the basis of observational refinement.

However, please note that this is a transformation done "on paper" rather than a technique that is used during the actual execution of operations by a particular TM system, i.e., the state changes are done by the system as if all transactions were atomic.

Intuition

Intuitively, the idea behind decomposition is to redefine any non-aborting transaction that releases early as a sequence of smaller transactions, each of which performs a single complete operation (i.e., an invocation and a response) and immediately commits. Such a decomposed transaction preserves the semantics and operations of the original transaction, but, since it is executed piecemeal, it no longer meets the definition of early release. This allows a transaction that originally had early release to satisfy opacity.

An example of this is given in Fig. 7.2 where Fig. 7.2b shows history H' which decomposes H from Fig. 7.2a. Here, T_i from H is emptied, and the original operations comprising T_i are executed as separate sequential transactions $T_{i,1}$, $T_{i,2}$, $T_{i,3}$, and $T_{i,4}$. This can be thought of essentially as nesting transactions $T_{i,1}$, $T_{i,2}$, $T_{i,3}$, and $T_{i,4}$ within T_i and using them to execute code within T_i.

The decomposed history may be considered interchangeable with the original history because decomposed histories behave exactly like the original histories from which they were produced. That is, the decomposed history observationally refines the original history.

Definition

Let H be a TM history with unique writes. Let $\mathbb{T}_{er}$ be a set of transactions s.t., $\mathbb{T}_{er} \subseteq \mathbb{T}$ and $T_i \in \mathbb{T}_{er}$ if, and only if, T_i is guaranteed to eventually commit and T_i releases some object early. We say a transaction eventually commits if the semantics of the TM ensure that it never aborts.

Given two transactions T_i and T_j ($T_i, T_j \in \mathbb{T}$) and some invocation or response event e_i executed by transaction T_i, let $\mathit{reassign}_j(e_i)$ be an event executed by T_j and defined as,

$$\mathit{reassign}_j(e_i) = \begin{cases} \mathit{inv}_j\big[op(x)w\big] & \text{if } e_i = \mathit{inv}_i\big[op(x)w\big], \\ \mathit{res}_j\big[u\big] & \text{if } e_i = \mathit{res}_i\big[u\big]. \end{cases}$$

Intuitively, $\mathit{reassign}_j(e_i)$ is the same event as e_i, only executed by transaction T_j rather than T_i.

Given some transaction T_i and an event e, let $\mathit{open}_i(e)$ and $\mathit{close}_i(e)$ denote sequences defined as follows:

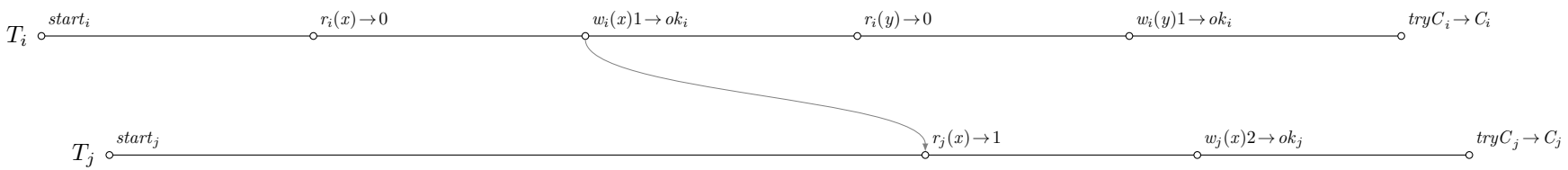
$$\mathit{open}_i(e) = [\mathit{start}_i \to \mathit{ok}_i] \cdot [e], \text{ and}$$
$$\mathit{close}_i(e) = [e] \cdot [\mathit{tryC}_i \to C_i].$$

Note that using *open* on the first event and *close* on the last event of some sequence of events "envelops" them in a transaction.

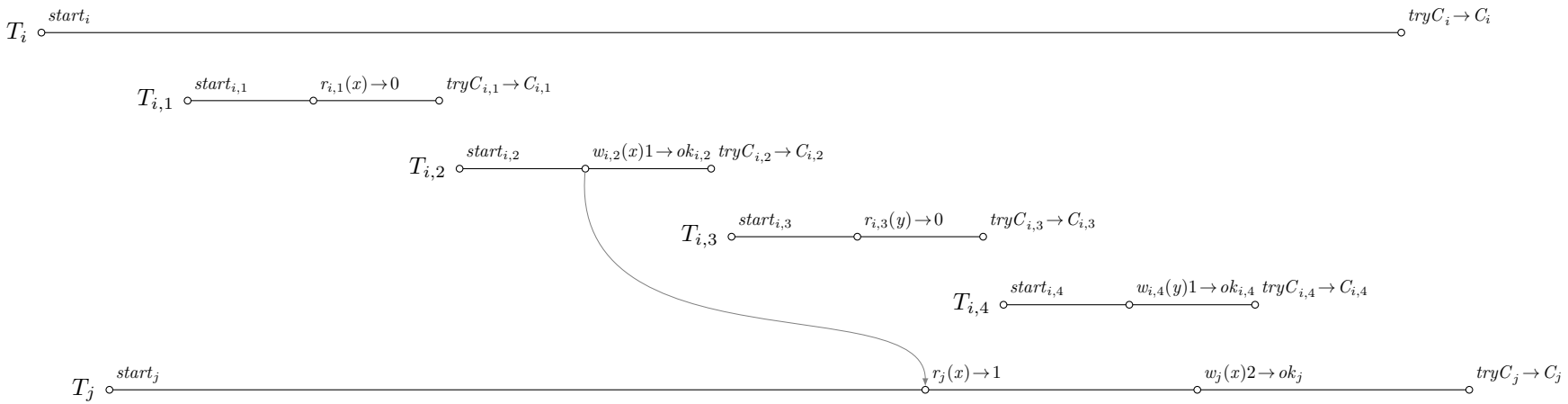
Then, we define *history decomposition* as follows:

Definition 28. *Given a history H, let its decomposition $H_d = \mathit{Decomp}(H)$ be identical to H except that every transaction $T_i \in H$ s.t., $T_i \in \mathbb{T}_{er}$ is transformed as follows:*

a) every complete operation execution in T_i that consists of an invocation event e_i' and a response event e_i'', e_i' is replaced in H_d by $\mathit{open}_j(\mathit{reassign}_j(e_i'))$ and e_i'' is replaced in H_d by $\mathit{close}_j(\mathit{reassign}_j(e_i''))$, where j is fresh, i.e. there is no T_j in H,



(a) Original history.



(b) Decomposed history.

Figure 7.2: Decomposition example.

b) every pending operation execution in T_i that only consists of an invocation event e'_i, e'_i is replaced in H_d by $open_j(reassign_j(e'_i))$, where j is fresh, i.e. there is no T_j in H.

Note that decomposition produces a set of new transactions for each transaction $T_i \in \mathbb{T}_{er}$. We call such a transaction T_i a *decomposed transaction*. The set of all transactions produced by decomposition to execute the events of a decomposed transaction T_i is denoted $\mathbb{T}^i_d$. This set explicitly contains the decomposed transaction T_i. We will refer to any transaction $T_j \in \mathbb{T}^i_d$ s.t. $T_j \neq T_i$ as a *product* of decomposition.

Next, let us show the opacity of a decomposed history, as follows.

Lemma 36. *Given H, a final-state opaque history, and $H_d = Decomp(H)$, H_d is final-state opaque.*

Proof. Let S be a sequential history fulfilling Def. 12 for H. Since H is final-state opaque then, by Def. 12, every transaction T_i in S is legal in S because $Vis(S, T_i)$ is legal.

Let S_d be a sequential history identical to S except that for every event e_i in some transaction T_i in S,

a) if e_i in H was replaced in H_d by $open_j(reassign_j(e_i))$ (given some T_j), then it is also replaced in S_d by $open_j(reassign_j(e_i))$,

b) if e_i in H was replaced in H_d by $close_j(reassign_j(e_i))$ (given some T_j), then it is also replaced in S_d by $close_j(reassign_j(e_i))$,

c) otherwise it remains e_i.

In addition every decomposed transaction directly precedes all of its product transactions.

For every transaction T_k in S_d exactly one of the following is true:

a) T_k is neither a decomposed transaction nor a product transaction. In that case, if $Vis(S_d, T_k) = Vis(S, T_k)$, then, since every transaction T_k in S is legal in S because $Vis(S, T_i)$ is legal, so, by extension, T_k in S_d is legal in S_d. Alternatively, if $Vis(S_d, T_k) \neq Vis(S, T_k)$, then $Vis(S_d, T_k)$ contains operations executed by product transactions of one or more decomposed transaction T_i, where $Vis(S, T_k)$ contains operations executed by T_i. The definition of S_d implies that S_d contains the same read and write operation executions as S, but some of the operations are executed by product transactions. Since the sequential specification $Seq(x)$ of any variable x ignores which transaction executes the operation as long as the written and read values are correct, then, if $Vis(S, T_k)$ is legal, then $Vis(S_d, T_k)$ is also legal. Hence, T_k in S_d is legal in S_d.

b) T_k is a decomposed transaction in S_d. Then, from Def. 28, transaction T_k does not contain any read or write operation executions. Therefore, T_k in S_d is legal in S_d if there is no other transaction T_i s.t., $T_i \prec_{S_d} T_k$. This is because $Vis(S_d, T_k)$ contains no read or write operations, so it is in $Seq(x)$ for any x, and thus $Vis(S_d, T_k)$ is legal. Otherwise, transaction T_k is preceded in S_d by any transaction T_j ($T_j \prec_{S_d} T_k$). For the sake of simplicity, let T_j be such a transaction that there is no other transaction T_i, s.t. $T_i \prec_{S_d} T_k$ and $T_j \prec_{S_d} T_i$. Then, $Vis(S_d, T_k)$ contains the same read and write operation executions as $Vis(S_d, T_j)$. Hence, if $Vis(S_d, T_j)$ is legal, then $Vis(S_d, T_k)$ is also legal. Thus, if the preceding transaction T_j in S_d is legal in S_d, then T_k in S_d is legal in S_d. Since we show in (a) and (c) that other types of transactions in S_d are legal in S_d and since T_k in S_d is legal in S_d if no transaction precedes T_k, then, trivially, T_k in S_d is legal in S_d.

c) T_k is a product transaction such that $T_k \notin S$ and $T_k \in \mathbb{T}^d_j$ for some decomposed transaction T_j. In that case $Vis(S_d, T_k)$ is the same as $Vis(S_d, T_j)$ with the exception

that $Vis(S_d, T_k)$ also contains the operations executed by product transactions $T_i \in \mathbb{T}_d^i$ of the decomposed transaction T_j, but only if $i = k$ or $T_i \prec_{S_d} T_k$. Let T_r be a special case of a product transaction of T_j for which there is no other product transaction $T_i \in \mathbb{T}_d^i$ s.t., $T_i \prec_{S_d} T_k$. Then, note that the definition of S_d implies that $S_d|T_r$ contains the same read and write operation executions as $S|T_j$, with the exception that some of the operations are executed by product transactions. By analogy to (a), since the sequential specification $Seq(x)$ of any variable x ignores which transaction executes the operation as long as the written and read values are correct, then, if $Vis(S, T_j)$ is legal, then $Vis(S_d, T_r)$ is also legal. Hence, T_r in S_d is legal in S_d. Since $Seq(x)$ is prefix closed, then if $Vis(S_d, T_r)$ is legal, then every prefix of $Vis(S_d, T_r)$ is also legal. Since T_k precedes T_r in S_d then $Vis(S_d, T_k)$ is a prefix of $Vis(S_d, T_r)$. Therefore $Vis(S_d, T_k)$ is legal, and thus T_k in S_d is legal in S_d.

Thus, all transactions in S_d are legal in S_d. Trivially, S_d preserves the real-time order of H_d and $S_d \equiv H_d$. Since S_d preserves the real-time order of H and every T_i in S_d is legal in S_d, then, by Def. 12, H_d is final-state opaque. □

Theorem 5. *Given H, a final-state opaque history, and $H_d = Decomp(H)$, H_d is opaque.*

Proof. Since H is final-state opaque, then, from Lemma 36, $H_d = Decomp(H)$ is final-state opaque.

Given $H_d = Decomp(H)$, Def. 28 ensures that if any transaction T_i releases early in H, then it is decomposed in H_d, so all of its write operation executions are reassigned in H_d to a product transaction T_j, s.t. $T_j \in \mathbb{T}_i^d$ and each write is directly followed in H_d by a successful commit operation executed by T_j. Consequently, if any transaction T_j writes value v to variable x, and any transaction T_k reads v from x, then T_j always commits in H_d before T_k reads v from x.

Let P be any finite prefix of H_d. The prefix potentially contains some transactions which are not completed, and therefore are aborted in the completion $P_c = Compl(P)$. Note, from the above, that if any transaction T_j is aborted in P_c, then there is no transaction that reads from T_j, because either any read operation reading from T_j execution would follow T_j's commit operation due to $T_j's$ decomposition, or T_j would not release early.

Hence, there can exist a sequential history $S_c \equiv P_c$ wherein any T_k in S_c is legal in S_c. Since H_d is final state opaque and P preserves the real time order of H_d, then S_c also preserves the real time order of P_c. If any a completion of any P_c is final-state opaque, then H_d is opaque. Thus, H_d is opaque. □

Observational Refinement of Decomposed Histories

Observational refinement [42, 5], intuitively, is a notion that given two programs and an observer who only sees the results of executing these two programs, if both programs always produce the same results, then, effectively, the programs are indistinguishable, and, therefore, interchangeable. The definition depends on what is considered observable behavior, which we assume to be the state of all variables during the execution of a program.

Given the set Var of all variables $Var = \{x_1, x_2, ..., x_w\}$, let *state* $\mathcal{S}$ be a set of variables paired with their values, i.e. $\mathcal{S} = \{(x_1, v_1), (x_2, v_2), ..., (x_w, v_w)\}$. Let the *initial state* $\mathcal{S}_0$ be a state s.t., for any $x_j \in Var$ and $(x_j, v_j) \in \mathcal{S}_0$, $v_j = v_0$. Let $\mathcal{P}(\mathcal{S})$ be a powerset of $\mathcal{S}$ and $\mathbb{E}$ be the set of all possible invocation and response events. Then let $eval : \mathcal{P}(\mathcal{S}) \times \mathbb{E} \mapsto \mathcal{P}(\mathcal{S})$ be a function representing the semantics of a TM system. It

is out of scope of this dissertation to define the complete semantics of TM, so we limit ourselves to presenting the following assumptions about *eval*.

Intuitively, we expect operations to behave deterministically based on the initial state, regardless of transaction. We also expect successful initialization and execution of commitment operations not to modify the state. More formally, given some states $\mathcal{S}$ and $\mathcal{S}'$, processes p_k and p_q variable x, value v, transactions T_i and T_j, we assume the following:

Assumption 1. *If* $(e_1, e_2) = r_i^k(x) \to v$ *and* $(e_1', e_2') = r_j^q(x) \to v$ *then* $eval(\mathcal{S}, e_1) = eval(\mathcal{S}, e_1')$ *and* $eval(\mathcal{S}', e_2) = eval(\mathcal{S}', e_2')$.

Assumption 2. *If* $(e_1, e_2) = w_i^k(x)v \to ok_i$ *and* $(e_1', e_2') = w_j^q(x)v \to ok_j$ *then* $eval(\mathcal{S}, e_1) = eval(\mathcal{S}, e_1')$ *and* $eval(\mathcal{S}', e_2) = eval(\mathcal{S}', e_2')$.

Assumption 3. *If* $(e_1, e_2) = start_i^k \to ok_i$ *then* $eval(\mathcal{S}, e_1) = \mathcal{S}$ *and* $eval(\mathcal{S}', e_2) = \mathcal{S}'$.

Assumption 4. *If* $(e_1, e_2) = tryC_i^k \to C_i$ *then* $eval(\mathcal{S}, e_1) = \mathcal{S}$ *and* $eval(\mathcal{S}', e_2) = \mathcal{S}'$.

We say that history $H = [e_1, e_2, ..., e_m]$ is *observed-state equivalent* to history $H' = [e_1', e_2', ..., e_n']$, which we denote $H \lessapprox H'$, when there exists such an injection $f : H \mapsto H'$, that for each $e_l \in H$ there exists $e_r \in H'$ s.t., if $eval(\mathcal{S}_{l-1}, e_l) = \mathcal{S}_l$ and $eval(\mathcal{S}_{r-1}, e_r) = \mathcal{S}_r$, then $\mathcal{S}_{l-1} = \mathcal{S}_{r-1}$ and $\mathcal{S}_l = \mathcal{S}_r$. Furthermore, it is necessary that if $f(e_l) = e_r$ and $f(e_{l-1}) = e_q$, then $q < r$. The intuition behind this definition is that if the events in both histories were evaluated side-by-side, they would cause the same changes to the state of the system, although one of the histories would contain some operations that were not present in the other history. However, these operations would not modify the state.

We say that transactional memory system M *observationally refines* transactional memory system M' if for any history H allowed by M there exists some history H' allowed by M' s.t., $H \lessapprox H'$.

Finally, we can say that the decomposition is indistinguishable from the original history, and can be used in its place for the purpose of establishing opacity, because the decomposed history reflects exactly the operations, their order, and the effect they have on the state of the system. As such, observing the execution of the original history does not differ from observing the execution of the decomposed history.

Theorem 6. *Given any complete final-state opaque history* H *with unique writes and its decomposed counterpart* $H_d = Decomp(H)$, $H \lessapprox H_d$.

Proof. From Def. 28, for every event e_i in H (for any transaction T_i in H), history H_d contains either the same event e_i, or (for some T_j, s.t. $H|T_j = \varnothing$) either the sequence $open_j(reassign_j(e_i)) = [start_j \to ok_j] \cdot [reassign_j(e_i)]$, or the sequence $close_j(reassign_j(e_i)) = [reassign_j(e_i)] \cdot [tryC_j \to C_j]$. Note that from Assumption 3 and Assumption 4, evaluation of any event e_j' in $[start_j \to ok_j]$ or in $[tryC_j \to C_j]$ does not impact state, i.e., $eval(\mathcal{S}, e_j') = \mathcal{S}$. Note also that Assumption 1 and Assumption 2 imply that $eval(\mathcal{S}, e_i) = eval(\mathcal{S}, reassign_j(e_i))$. Hence we can derive from $Decomp(H)$ a function $f : H \mapsto H_d$ that for any event e_i returns $reassign_j(e_i)$ if $Decomp(H)$ transforms e_i to either $open_j(reassign_j(e_i))$ or $close_j(reassign_j(e_i))$, or e_i otherwise.

Note that this function is an injection from H to H_d, and that if some event e_i is not in the range of f, it is part of a transaction initialization or transaction commitment, so $eval(\mathcal{S}, e_i) = \mathcal{S}$. Furthermore, note that for any event e_i in $H|T_i$ function f returns either the same event e_i or some other event e_j that represents the invocation of or response to the same operation, just executed by a different transaction T_j which is a product transaction of the decomposition of T_i. In that case, from Assumption 1 and Assumption 2 given some state $\mathcal{S}$, for any e_i it is true that $eval(\mathcal{S}, e_i) = eval(\mathcal{S}, f(e_i))$.

Finally, since function *Decomp* preserves the order of events from H in H_d, then for two events e_1 and e_2 in H, s.t. e_1 precedes e_2 in H, $f(e_1)$ precedes $f(e_2)$ in H_d. Thus, since f exists, then by definition of observed-state equivalency, $H \lessapprox H_d$. □

Corollary 19. *Given a TM system M and a (hypothetical) TM system M_d that for every history H allowed by M produces a history $H_d = Decomp(H)$, since $H \lessapprox H_d$, then M observationally refines M_d.*

7.1.2 SVA Opacity Through Decomposition

Lemma 37. *A determined finite SVA history is final-state opaque through decomposition.*

Proof. Since all SVA transactions $T_i \in \mathbb{T}$ access any variable $x \in Var$ only when it is in the appropriate version (denoted $\mathrm{pv}_x(i)$), and since the versions of variables increase monotonically, then they have exclusive access to x. That is, an SVA transaction T_i can access x in history H only after a preceding transaction T_j releases x after last use or commits. Then, for each $x \in Var$, given set $\mathbb{T}_x$ that contains all transactions which access x, there exists a total order $\prec_{\mathbb{T}_x}$ on $\mathbb{T}_x$ s.t., given transactions $T_i, T_j \in \mathbb{T}_x$, $T_j \prec_{\mathbb{T}_x} T_i$ iff $\mathrm{pv}_j(x) < \mathrm{pv}_i(x)$. By extension, given any SVA history H, there exists a partial order $\preceq_H$ on H that agrees with $\prec_{\mathbb{T}_x}$ for each $x \in Var$ (i.e. $\prec_{\mathbb{T}_x} \subseteq \preceq_H$, for each $x \in Var$).

Let H be any finite SVA history that is determined (i.e. one for which $Compl(H) = H$). Let S be a sequential history equivalent to H s.t. transactions in S are ordered in accordance to $\preceq_H$. Then, trivially, S follows the real-time order of H.

Note that given a determined SVA history H, no transaction aborts in H. Note also that given two transactions $T_i, T_j \in H$, s.t. $T_i, T_j \in \mathbb{T}_x$, if T_i accesses x after T_j, then T_i accesses x after T_j releases x or commits. Since SVA transactions release objects after last use, then any transaction always views a consistent state of the system and is the only transaction that executes operations on a given variable between its first and last access of that variable. Hence, each transaction in H behaves as if it were executed sequentially. So each transaction in any sequential history S s.t. $H \equiv S$ conforms to a sequential specification of each variable. Therefore, every transaction T_i in S is legal in S.

Since we can construct a sequential history S equivalent to H that preserves the real time order of H and every transaction T_i in S is legal in S, therefore H is final-state opaque. □

Theorem 7. *Every SVA history is opaque through decomposition.*

Proof. Let H be any finite determined SVA history. Let $H_d = Decomp(H)$. Since H is final-state opaque (Lemma 37), then, by Theorem 5, H_d is opaque. Then, since H_d observationally refines H (Theorem 6), H is indistinguishable from an opaque history H_d.

Let H' be any SVA history that is not determined. Trivially, there exists such a determined SVA history H, that H' is a prefix for H. Since every determined history is final state opaque (Lemma 37), there exists a decomposed history $H_d = Decomp(H)$ that is opaque (Theorem 5). Then, there must exist a $H'_d = Decomp(H')$ that is a prefix of H_d. Since all prefixes of H_d are final-state opaque, then H'_d is final-state opaque (Def. 13). Also, since all prefixes of H_d are final-state opaque, then all prefixes of H'_d are final-state opaque, and in consequence H'_d is opaque. Then, since H'_d observationally refines H' (Theorem 6), H' is indistinguishable from an opaque history. □

7.2 Last-use Opacity of SVA+R

As we mention in Section 4.1.2, since SVA and SVA+R allow transactions to access objects that could have been modified by still-live transactions, it becomes impossible to

demonstrate opacity directly. As we discuss in Section 7.1 in detail, However, it can be shown that since SVA does not abort, any history produced by SVA is equivalent in its effects to an opaque history. We discuss this in detail in Section 7.1. This means that in SVA, the theoretical inconsistent views do not have practical consequences.

However, once we lift the algorithm into the arbitrary abort model, it becomes possible for transactions to view inconsistent state not only from live transactions, but also from ones that will eventually abort, which may lead to cascading aborts. Then, not only is opacity impossible to demonstrate directly, but some histories produced by SVA+R will diverse from what is expected by opacity.

Nevertheless, even though cascading aborts are admitted in some histories, SVA+R carefully limits the inconsistent views that can occur. In particular, it specifically prohibits overwriting and orders commits and aborts to reflect the order in which transaction access objects. Hence SVA+R is still able to provide strong guarantees, and thus satisfies the properties of last-use opacity

In the following sections we present a proof sketch showing that SVA+R is last-use opaque. A complete proof (including demonstrations of the observations we make about the algorithm) is in Appendix A.

7.2.1 Observations

First, we make the following straightforward observations about SVA+R.

Observation 1 (Version Order)**.** *Given the set $\mathbb{T}^x_H$ of all transactions that access x in H there is a total order called a version order $\prec_x$ on $\mathbb{T}^x_H$ s.t. for any $T_i, T_j \in \mathbb{T}^x_H$, $T_i \prec_x T_j$ if $\mathrm{pv}_i(x) < \mathrm{pv}_j(x)$.*

Observation 2 (Access Order)**.** *If $T_i \prec_x T_j$ and T_i performs operation op_i on x, and T_j performs operation op_j on x, then op_i is completed in H before op_j.*

Observation 3 (No Bufferring)**.** *Since transactions operate on variables rather than buffers, any read operation $op = r_i(x) \to v$ in any transaction T_i is preceded in H by some write operation $w_j(x)v \to ok_j$ in some T_j (possibly $i = j$).*

Observation 4 (Read from Released)**.** *If transaction T_i executes a read operation or a write operation op on x in H, then any transaction that previously executed a read or write operation on x is either committed, aborted, or decided on x before op.*

Observation 5 (Do Not Read Aborted)**.** *Assuming unique writes, if transaction T_i executes $w_i(x)v \to u$ and aborts in H, then x will be reverted to a previous value. In consequence, no other transaction can read v from x.*

Observation 6 (Commit Order)**.** *If transaction T_i accesses x in H and commits or aborts in H, any transaction that previously executed a read or write operation on x is either committed or aborted before T_i commits or aborts.*

Observation 7 (Forced Abort)**.** *If transaction T_i reads x from T_j and T_j subsequently aborts, then T_i also aborts.*

7.2.2 Last-use Opacity

Then, the main lemma follows, showing that SVA+R produces final-state opaque histories. For convenience, we assume that the SVA+R program always writes values to variables that are unique and in the domain of the variable.

Lemma 38. *Any SVA+R history H is final-state last-use opaque.*

Proof sketch. Let $H_C = Compl(H)$ be a completion of H if for every $T_i \in H$, if T_i is live or commit-pending in H, then T_i is aborted in H_C. Given H_C we can construct $\hat{S}_H$, a sequential history s.t. $\hat{S}_H \equiv H_C$, where for any two transactions $T_i, T_j \in H_C$:

a) if $T_i \prec_{H_C} T_j$, then $T_i \prec_{\hat{S}_H} T_j$,

b) if $T_i \prec_x T_j$ for any variable x, then $T_i \prec_{\hat{S}_H} T_j$.

Note that if some transaction T_i commits in H, then it commits in $\hat{S}_H$ (and *vice versa*). Otherwise T_i aborts in $\hat{S}_H$.

Let T_i be any transaction committed in H. Thus, T_i also commits in $\hat{S}_H$. From Observation 3, any read operation execution $op_i = r_i(x) \to v$ in $H|T_i$ is preceded in H by $op_j = w_j(x)v \to ok_j$. If op_i is local, then $i = j$, so op_j is in a committed transaction. If op_i is not local, then $i \neq j$. In that case, from Observation 5, T_j cannot be aborted before op_i in H. Consequently, T_j is either committed before op_i in H, live in H, or committed or aborted after op_i. In the former case T_i reads from a committed transaction. In the latter case, since T_i is committed, then from Observation 4 and Observation 6 we know that T_j commits or aborts in H before T_i commits. In addition, from Observation 7 we know that T_j cannot abort in H, because it would have caused T_i to also abort. Thus, any committed T_i reads only from committed transactions.

From Observation 2, if T_i reads from the value written by an operation in T_j then the write in T_j completes before the read in T_i, which implies $T_j \prec_x T_i$. Hence, $T_j \prec_{\hat{S}_H} T_i$. Thus, if T_i is committed in $\hat{S}_H$ and reads from some T_j, then any such T_j is committed and precedes T_i, so $\hat{S}_H|T_j \subseteq Vis(\hat{S}_H, T_i)$. Since all reads in committed transactions read from preceding committed transactions, then for each read in $Vis(\hat{S}_H, T_i)$ reading v from x there will be a write operation execution writing v to x in $Vis(\hat{S}_H, T_i)$. Since, from Observation 2, all accesses on x operations follow $\prec_x$, then $Vis(\hat{S}_H, T_i)$ is legal for any committed T_i. Thus, any T_i that is committed in $\hat{S}_H$ is legal in $\hat{S}_H$.

Let T_i be a transaction that is live or aborts in H, so it aborts in $\hat{S}_H$. From Observation 3 any read operation execution $op_i = r_i(x) \to v$ in $H|T_i$ is preceded in H by $op_j = w_j(x)v \to ok_j$. If op_i is local, then $i = j$, so op_j is always in $Vis(\hat{S}_H, T_i)$ where op_j precedes op_i. If op_i is not local, then $i \neq j$. In that case, from Observation 5, T_j cannot be aborted before op_i in H. Consequently, T_j is either committed before op_i in H, live in H, or committed or aborted after op_i. In the former case T_i reads from a committed transaction. In the latter case, from Observation 4 we know that either T_j commits in H or T_j is decided on x in H. Thus, any committed T_i reads x only from committed transactions or transactions that are decided on x.

From Observation 2, if T_i reads from the value written by an operation in T_j then the write in T_j completes before the read in T_i, which implies $T_j \prec_x T_i$. Hence, $T_j \prec_{\hat{S}_H} T_i$. Thus, if T_i is aborted in $\hat{S}_H$ and reads from some T_j, then any such T_j is either committed and precedes T_i, so $\hat{S}_H|T_j \subseteq LVis(\hat{S}_H, T_i)$, or T_j is decided on any x if T_i reads from x, so $\hat{S}_H\mathring{|}T_j \subseteq LVis(\hat{S}_H, T_i)$. Since all reads in aborted transactions read x from preceding committed transactions or transactions decided on x, then for each read in $LVis(\hat{S}_H, T_i)$ reading v from x there will be a write operation execution writing v to x. Since, from Observation 2 all accesses on x operations follow $\prec_x$, then $LVis(\hat{S}_H, T_i)$ is legal for any aborted T_i. Thus, any T_i that is aborted in $\hat{S}_H$ is last-use legal in $\hat{S}_H$.

Since any committed T_i in $\hat{S}_H$ is legal in $\hat{S}_H$, and any aborted T_i in $\hat{S}_H$ is last-use legal in $\hat{S}_H$, and since $\hat{S}_H$ trivially follows the real time order of H, then from Def. 23 H is final-state last-use opaque. □

Full proof for Lemma 38 is given in Appendix A.

Theorem 8. *Any SVA+R history H is last-use opaque.*

Proof. Since by Lemma 38 any SVA+R history H is final-state last-use opaque, and any prefix P of H is also an SVA+R history, then every prefix of H is also final-state last-use opaque. Thus, by Def. 24, H is last-use opaque. □

7.3 Last-use Opacity of OptSVA+R

In this section, we demonstrate that OptSVA+R meets the same correctness guarantees as SVA+R, by presenting a proof for last-use opacity. In addition to OptSVA+R's correctness this shows that the optimizations used by OptSVA+R to increase parallelism do not sacrifice or otherwise relax correctness.

Given that OptSVA+R divorces the operations performed on shared variables within the code of the transaction from the actual accesses to memory that are executed, and since last-use opacity is defined on operations on shared variables, showing correctness is not straightforward. This is further exacerbated by the fact that last-use opacity is defined explicitly as prefix closed, meaning that it must be demonstrated for all prefixes of a given transactional schedule, not just the schedule itself. Hence, proving it for a complex system is troublesome, just as it is troublesome for opacity, from which the definition was obtained. In fact, markability [52] and graph representation of opacity [33], are both techniques trying to work around the basic definition of opacity. Hence, apart from the proof itself, we contribute *trace harmony*, a proof technique that shows last-use opacity based on interrelationships among memory accesses (and can be easily extended to show related properties like opacity).

Thus, in this section, we first present the preliminary material that defines how operations on memory are represented within traces. Then, we use this abstraction to give the definitions making up trace harmony. Each definition is relatively simple and evaluates a particular single aspect of the relationships between transactions, operations, and memory accesses within a trace. If some trace satisfies all of them in aggregate, it is harmonious. We then claim that any harmonious trace implies a last-use opaque history in general (we provide a demonstration of this proposition in Appendix A). This means that given some trace, it is enough to prove that it satisfies each of the individual definitions making up trace harmony to show that the trace is last-use opaque. Given this, we demonstrate that OptSVA+R traces are harmonious in Section 7.3.4, and so, that OptSVA+R is last-use opaque.

7.3.1 Events

Events are the results of transactions directly interacting with the memory representing shared variables. When during the execution of some program, some transaction accesses a variable's state (either viewing it or updating it), it issues an update event that is logged in the trace resulting from the execution.

A *view event* $g_i(x)v$ is any event that represents some transaction T_i viewing the state of variable x (i.e. reading the memory location where the value of x is stored) and getting the value of v. An *update event* $s_i(x)v$ is any event that represents a modification of the state of variable x by transaction T_i, setting it to the value of v.

Some operations can abort the transaction, rather than doing what they are intended to do. For instance, a write operation may fail with an abort rather than setting a new value of some variable. In such cases the transaction will execute specific code that is meant to clean up after the transaction and revert any variables the transaction modified

to a previous (consistent) state. We will refer to this code as the recovery procedure. Any update events executed as part of a recovery procedure are called *recovery* (update) events. In contrast, all update events that are not recovery events are called *routine* (update) events. For distinction, we denote a routine update $\circ s_i(x)v$ and a recovery update $\natural s_i(x)v$.

Given a view event $g_i(x)v$ (for some T_i), v is specified by the most recent preceding update event on x in a given trace. I.e., if the most recent preceding update event on x is some $s_j(x)v'$ (for some T_j), then $v = v'$. Note, that this distinction does not depend on how the events appear in the trace, but is intrinsic to the code that executes them.

Event $e = s_i(x)v$ is the *ultimate update* event on x in $\mathscr{T}$ iff there is no $e' = s_j(x)v'$ s.t. $e \prec_{\mathscr{T}} e'$. Event $e = s_i(x)v$ is the *ultimate routine update* event on x in $\mathscr{T}$ iff e is routine and there is no $e' = s_j(x)v'$ s.t. $e \prec_{\mathscr{T}} e'$ and e' is routine.

Given a view event $e_v = g_i(x)\square$ in some T_i and an update event $e = s_j(x)\square$ in some T_j, e prefaces e_v in trace $\mathscr{T}$, denoted $e \lessdot_{\mathscr{T}} e_v$ iff $e \prec_{\mathscr{T}} e_v$ and there is no update event $e' = s_k(x)\square$ in any T_k s.t. $e \prec_{\mathscr{T}} e' \prec_{\mathscr{T}} e_v$. Given a read operation execution $op_r \in \mathscr{T}$ s.t., $op_r = r_i(x) \to v$ and op_r that consists of an invocation event e_i and a response event e_r, and a view event $e_v = g_i(x)v'$, op_r *depends on* e_v (denoted $op_r \leftsquigarrow e_v$) iff $v' = v$ and $e_v \lessdot_{\mathscr{T}} e_r$. Given a write operation execution $op_w \in \mathscr{T}$ s.t., $op_w = w_i(x)v \to ok_i$ and op_w consists of an invocation event e_i and a response event e_r, and an update event $e_u = g_i(x)v'$, op_w *instigates* e_u (denoted $op_w \rightsquigarrow e_u$) iff $v' = v$ and $e_i \lessdot_{\mathscr{T}} e_u$.

Transaction T_i *views* transaction T_j, denoted $T_i \curvearrowright T_j$, if $\exists e_u, e_v \in \mathscr{T}$ s.t. $e_v = g_i(x)v$ and $e_u = \circ s_j(x)v$ and $e_u \lessdot_{\mathscr{T}} e_v$. Transaction T_i *virtually views* transaction T_j, denoted $T_i \curvearrowright\!\!\!\!\curvearrowright T_j$, if $\exists e_u, e_v \in \mathscr{T}$ s.t. $e_v = g_i(x)v$ and $e_u = \circ s_j(x)v$ and $e_u \prec_{\mathscr{T}} e_v$.

Event access set $ESet_i$ for some transaction $T_i \in \mathscr{T}$ is such a set of variables such that $x \in ESet_i \iff \exists e \in \mathscr{T}|T_i$ s.t. $e = \circ s_i(x)v$ or $e = g_i(x)v$.

Given $T_i \in \mathscr{T}$, s.t. $e_v = g_i(x)v \in \mathscr{T}|T_i$ and e_v is initial in $\mathscr{T}|T_i$, let $\psi_{\mathscr{T}}(T_i, x)$ be such longest sequence of transactions that: a) if $\exists T_j \in \mathscr{T}$ s.t. $e_a = \natural s_j(x)v \in \mathscr{T}|T_j$ and $e_a \lessdot_{\mathscr{T}} e_v$ then $\psi_{\mathscr{T}}(T_i, x) = \psi_{\mathscr{T}}(T_j, x) \cdot T_i$, otherwise b) $\psi_{\mathscr{T}}(T_i, x) = \varnothing \cdot T_i$.

Let a *view chain* $\xi(\mathscr{T}, T_i, T_j)$ be a sequence of transactions s.t. T_i is the first element, and T_j is the last element, and for each pair of consecutive transactions T_k, T_l, it is true that $T_l \curvearrowright\!\!\!\!\curvearrowright T_k$. Let $H|\xi(\mathscr{T}, T_i, T_j)$ be the longest subsequence of $\mathscr{T}$ s.t. $e \in H|\xi(\mathscr{T}, T_i, T_j)$ iff $e \in \mathscr{T}|T_i$ and $T_i \in \xi(\mathscr{T}, T_i, T_j)$.

7.3.2 Trace Harmony

Since OptSVA+R limits events within a transaction to at most a single routine update event, at most a single recovery update event, and at most a single view event per variable, we limit the method presented below to such a case. This is represented by the definition of *minimalism* below. (However, the method can be extended to allow multiple routine update events and multiple view events per transaction.)

Definition 29 (Minimalism)**.** *Given transaction $T_i \in \mathscr{T}$, for each x, $\mathscr{T}|T_i$ contains:*

a) either none or one view event $g_i(x)\square$,

b) either none or one routine update event $\circ s_i(x)\square$,

c) either none or one recovery update event $\natural s_i(x)\square$.

Trace isolation stipulates, that once a transaction starts accessing the memory of some variable, it has exclusive access to it until it is done performing routine updates and view events on it. Hence a transaction is not interfered with by other transaction when it is performing memory accesses, unless an abort is required. Furthermore, if one transaction accesses the memory of one variable before another transaction, then that other transaction cannot access any other variable before the first transaction does.

Definition 30 (Trace Isolation). *Trace $\mathscr{T}$ is isolated, iff given any two transactions T_i and T_j in $\mathscr{T}$ for every $x \in ESet_i \cap ESet_j$, it is true that given any event e_i s.t. $e_i = g_i(x)v \in \mathscr{T}|T_i$ or $e_i = \circ s_i(x)v' \in \mathscr{T}|T_j$, and any event e_j s.t. $e_j = g_j(x)v' \in \mathscr{T}|T_j$ or a routine update event $e_j = \circ s_j(x)v' \in \mathscr{T}|T_j$, $e_i \prec_{\mathscr{T}} e_j$.*

Isolation order imposes an order on transactions in a trace that respects the order of executing update and view events on variables. Given an isolated trace, there exist the following orders:

Definition 31 (Variable Isolation Order). *Two transactions T_i and T_j are isolation-ordered in trace $\mathscr{T}$ with respect to x, which we denote $T_i \dot{\prec}^x_{\mathscr{T}} T_j$, if given any event e_i s.t. $e_i = g_i(x)v \in \mathscr{T}|T_i$ or $e_i = \circ s_i(x)v' \in \mathscr{T}|T_j$, and any event e_j s.t. $e_j = g_j(x)v' \in \mathscr{T}|T_j$ or a routine update event $e_j = \circ s_j(x)v' \in \mathscr{T}|T_j$, and $e_i \prec_{\mathscr{T}} e_j$.*

Definition 32 (Direct Isolation Order). *Two transactions T_i and T_j are directly isolation-ordered $T_i \ddot{\prec}_{\mathscr{T}} T_j$ if for every $x \in ESet_i \cap ESet_j$, $T_i \dot{\prec}^x_{\mathscr{T}} T_j$.*

Definition 33 (Isolation Order). *Two transactions T_i and T_j are isolation-ordered $T_i \dot{\prec}_{\mathscr{T}} T_j$, if there exists a sequence of transactions $\epsilon = T_i \cdot \ldots \cdot T_j$, where for every pair of consecutive transactions $T_n, T_m \in \epsilon$, $T_n \ddot{\prec}_{\mathscr{T}} T_m$.*

Note that if $T_i \prec_{\mathscr{T}} T_j$ and $x \in ESet_i \cap ESet_j$, then $T_i \dot{\prec}_{\mathscr{T}} T_j$, so the isolation order preserves real-time order.

Consonance describes when a particular event or operation involves a value that can be considered correct, which is determined by other events or operations that either precede or follow the one in question. Specifically, a view event is consonant if it retrieves the value that was written there by a preceding event, or the initial value, if no events preceded. A consonant read operation must then return a value that was retrieved by a view event beforehand. On the other hand, a routine update event must be caused by some write operation. Whereas a consonant recovery update event is one that cleans up after a routine update and reverts the state of a variable to a value that was retrieved by a view event that view the unmodified state of the variable in question.

Definition 34 (View Consonance). *Given some $T_i \in \mathscr{T}$, a view event $e_v = g_i(x)v$ is consonant in $\mathscr{T}$ iff either:*

a) $v = 0$ and $\nexists e_u \in \mathscr{T}$, s.t. $e_u = s_j(x)v'$ for any T_j, and $e_u \prec_{\mathscr{T}} e_r$,

b) $v \neq 0$ and $\exists e_u \in \mathscr{T}$, s.t. $e_u = \circ s_j(x)v$ for some T_j, $i \neq j$, $e_u \lessdot_{\mathscr{T}} e_r$, and e_u is the ultimate routine update on x in $\mathscr{T}|T_j$, or

c) $\exists e_u \in \mathscr{T}$, s.t. $e_u = \boxplus s_j(x)v$ for some T_j, $i \neq j$, $e_u \lessdot_{\mathscr{T}} e_r$.

Definition 35 (Routine Update Consonance). *Given some $T_i \in \mathscr{T}$, a routine update event $e_u = \circ s_i(x)v$ is consonant in $\mathscr{T}$ iff e_u is instigated in $\mathscr{T}$ by a consonant write operation execution.*

Definition 36 (Recovery Update Consonance). *Given some $T_i \in \mathscr{T}$, event $e_a = \boxplus s_i(x)v$ is* consonant *in $\mathscr{T}$ iff:*

a) e_a is conservative *in $\mathscr{T}$, i.e. there exists a consonant non-local view event e_v in $\mathscr{T}|T_i$ that is initial in $\mathscr{T}|T_i$,*

b) e_a is needed *in $\mathscr{T}$, i.e. $\exists e_u = \circ s_i(x)v' \in \mathscr{T}|T_i$, s.t. $e_u \prec_{\mathscr{T}|T_i} e_a$,*

c) e_a is dooming *in $\mathscr{T}$, i.e. $\nexists r \in \mathscr{T}$, s.t. $r = res_i\big[C_i\big]$, $e_a \prec_{\mathscr{T}|T_i} r$,*

d) e_a is ending *in $\mathscr{T}$, i.e. $\nexists e \in \mathscr{T}$, s.t. $e = g_i(x)v'$ or $e = s_i(x)v'$, $e_a \prec_{\mathscr{T}|T_i} e$,*

e) e_a is clean *in $\mathscr{T}$, i.e. given view e_v that justifies that e_a is conservative, there is no event $e'_a = \boxplus s_j(x)v'$ in any T_j s.t. $T_j \dot{\prec}^x_{\mathscr{T}} T_i$ and $e_v \prec_{\mathscr{T}} e'_a \prec_{\mathscr{T}} e_a$.*

Definition 37 (Non-local Read Consonance). *A non-local read operation execution is consonant in trace $\mathscr{T}$ iff it depends in $\mathscr{T}$ on a consonant non-local view event.*

Definition 38 (Local Read Consonance). *Given some $T_i \in \mathscr{T}$, a local read operation execution $op_r = r_i(x) \to v$ is consonant in trace $\mathscr{T}$ iff there exists $op_w = w_i(x)v \to ok_i \in \mathscr{T}|T_i$, s.t. $op_w \lessdot_{\mathscr{T}|T_i} op_r$, and op_w is consonant.*

Definition 39 (Write Consonance). *A write operation execution $w_i(x)v \to ok_i$ in some T_i is consonant in trace $\mathscr{T}$ iff $v \neq 0$ and v is within the domain of x.*

Definition 40 (Trace Consonance). *Trace $\mathscr{T}$ is consonant iff all operation executions, update events, and view events in trace $\mathscr{T}$ are consonant.*

Obbligato ensures that update events required by write operations happen on time, so that the values written to variables by operation executions are actually set in memory by the time the transaction relinquishes control of each variable. This means that a routine update event is required after a write operation by the time a transaction commits (*committed write obbligato*), one is required after a closing write operation, before any other transaction attempts to access that variable (*closing write obbligato*), and one is required if a non-aborted transaction executed write operations and another transaction accesses the variables in question (*view write obbligato*).

Definition 41 (Committed Write Obbligato). *Given $T_i \in \mathscr{T}$, if $\exists op_w \in \mathscr{T}|T_i$ s.t. $op_w = w_i(x)v \to ok_i$, op_w is non-local, and $\exists r \in \mathscr{T}|T_i$ s.t. $r = res_i\big[C_i\big] \in \mathscr{T}|T_i$, then op_w is* in obbligato *iff $\exists e_s \in \mathscr{T}|T_i$, s.t. $e_s = \circ s_i(x)v$ and $op_w \rightsquigarrow e_s$ and $e_s \prec_{\mathscr{T}|T_i} r$.*

Definition 42 (Closing Write Obbligato). *Given $T_i \in \mathscr{T}$, if $\exists op_w \in \mathscr{T}|T_i$, and $\exists T_j \in \mathscr{T}$ s.t. $T_i \dot{\prec}_{\mathscr{T}} T_j$, and there is closing write $op_i = w_i(x)\square \to ok_i \in \mathscr{T}|T_i$, and there is an event $e_v = g_j(x)\square \in \mathscr{T}|T_j$, then op_i is* in closing obbligato *iff $\exists e_u \in \mathscr{T}|T_i$, s.t. $e_u = \circ s_i(x)v$ and $op_i \rightsquigarrow e_u$, and $e_u \prec_{\mathscr{T}} e_v$.*

Definition 43 (View Write Obbligato). *Given $T_i \in \mathscr{T}$, if $\exists T_j \in \mathscr{T}$, s.t. $T_i \dot{\prec}_{\mathscr{T}} T_j$, if there is $op_i = w_i(x)\square \to ok_i \in \mathscr{T}|T_i$, and $e_v = g_j(x)\square \in \mathscr{T}|T_j$, then op_i is* in view write obbligato *iff there is $e_u = s_i(x)\square \in \mathscr{T}|T_i$, s.t. $e_u \prec_{\mathscr{T}} e_v$ or $\exists r = res_i\big[A_i\big] \in \mathscr{T}|T_i$, s.t. $r \prec_{\mathscr{T}} e_v$.*

Definition 44 (Obbligato). *Trace $\mathscr{T}$ is obbligato iff*

a) all non-local writes in all transactions committed in $\mathscr{T}$ are in committed obbligato,

b) all closing writes whose effects are potentially viewed are in closing write obbligato,

c) all writes whose effects are potentially viewed are in view write obbligato.

Decisiveness is achieved, when transactions do not let other transactions to view the values they set to the variables they modify until they commit or perform their closing writes.

Definition 45 (Decisiveness). *Trace $\mathscr{T}$ is* decisive *iff given any pair of transactions $T_i, T_j \in \mathscr{T}$, s.t. $T_i \ddot{\rightsquigarrow} T_j$ for any $e_u = \circ s_j(x)v \in \mathscr{T}|T_j$ and $e_v = g_i(x)v \in \mathscr{T}|T_i$, then either T_j is decided on x, $\exists r = res_j\big[C_j\big] \in \mathscr{T}|T_j$, s.t. $e_u \prec_{\mathscr{T}} r \prec_{\mathscr{T}} e_v$.*

Abort accord is a relation between two transactions, where if one of them views the update events performed by the other, and the other transaction aborts, then the first transaction is not permitted to abort.

Definition 46 (Abort Accord). *Trace $\mathscr{T}$ is in abort accord iff for any two transactions T_i and T_j in $\mathscr{T}$ s.t.:*

a) $T_j \ddot{\curvearrowleft} T_i$, *if* T_i *is aborted in* $\mathscr{T}$,

b) $\exists e_u = \circ s_i(x)\square \in \mathscr{T}|T_i$ *and* $\exists e = \circ s_j(x)\square \in \mathscr{T}|T_j$ *or* $e = g_j(x)\square \in \mathscr{T}|T_j$ *and* $\exists e_a = \natural s_i(x)\square \in \mathscr{T}|T_i$, *and* $e_u \prec_{\mathscr{T}} e \prec_{\mathscr{T}} e_a$,

then T_j *is either live or aborted in* $\mathscr{T}$.

Commit accord is a similar relation, where given two transactions such that one of them views the update events performed by the other, and the latter transaction commits, then the former transaction must have also committed.

Definition 47 (Commit Accord)**.** *trace* $\mathscr{T}$ *is in commit accord iff for any two transactions* T_i *and* T_j *in* $\mathscr{T}$ *s.t.* $T_j \ddot{\curvearrowleft} T_i$, *if* T_j *is committed in* $\mathscr{T}$, *then* T_i *is committed in* $\mathscr{T}$.

Coherence specifies, that if a transaction commits, all preceding transactions according to the isolation order either committed or aborted beforehand.

Definition 48 (Coherence)**.** *Trace* $\mathscr{T}$ *is coherent iff for any two transactions* T_i *and* T_j *in* $\mathscr{T}$, *s.t. for some* x, $T_i \dot{\prec}^x_{\mathscr{T}} T_j$, *if* $\exists r_j = res_j\big[C_j\big] \in \mathscr{T}|T_j$, *then* $\exists r_i = res_i\big[C_i\big]$ *or* $r_i = res_i\big[A_i\big]$ *and* $r_i \prec_{\mathscr{T}} r_j$.

Abort Coda specifies when a recovery event can be expected to be issued. If a transaction updates the state of some variable and eventually aborts, either it or another transaction will issue a recovery event to clean up that update before the transaction in question completes aborting. On the other hand, if the transaction commits, neither it or any other transaction will issue a recovery event to revert the state of that variable to another value.

Definition 49 (Abort Coda)**.** *Trace* $\mathscr{T}$ *has* coda *iff for any transaction* T_i*:*

a) if T_i *aborts in* $\mathscr{T}$ *(so* $r = res_i\big[A_i\big] \in \mathscr{T}|T_i$*), then if* $\exists e_u = \circ s_i(x)v \in \mathscr{T}|T_i$, *then for some* T_j *s.t.* $i = j$ *or* $T_i \dot{\prec}^x_{\mathscr{T}} T_j$ $\exists e_a = \natural s_j(x)v' \in \mathscr{T}$ *s.t.* $e_u \prec_{\mathscr{T}} e_a$,

b) if T_i *commits in* $\mathscr{T}$ *(so* $\exists r = res_i\big[C_i\big] \in \mathscr{T}|T_i$*), then if* $\exists e = \circ s_i(x)v \in \mathscr{T}|T_i$ *or* $e = g_i(x)v \in \mathscr{T}|T_i$, *then for any* T_j *s.t.* $i = j$ *or* $T_i \dot{\prec}^x_{\mathscr{T}} T_j$ $\nexists e_a = \natural s_j(x)v' \in \mathscr{T}$ *s.t.* $e \prec_{\mathscr{T}} e_a$.

Chain consistency describes what events are allowed and barred from a chain of transactions. Specifically, *chain isolation* stipulates that, a chain of transactions executing view and update events is not broken by a recovery event, so a transaction cannot view an inconsistent state where the value of one variable is retrieved before an abort was performed, and another one after. *Chain self-containment* is the situation where the values viewed by a transaction in some chain always come from within that chain.

Definition 50 (Chain Isolation)**.** *Given trace* $\mathscr{T}$, *transactions* $T_i, T_j \in \mathscr{T}$, $\xi(\mathscr{T}, T_i, T_j)$ *is isolated if* $\forall T_k \in \xi(\mathscr{T}, T_i, T_j)$, *s.t.* $e^k = \circ s_k(x)v$, *there is no* T_l *(possibly* $T_l \notin \xi(\mathscr{T}, T_i, T_j)$*) s.t.* $\exists e^l = \natural s_l(x)v'$ *where* $v = v'$ *and* e^l *is between* e^k *and any other event in any transaction in* $\xi(\mathscr{T}, T_i, T_j)$.

Definition 51 (Chain Self-containment)**.** *Given trace* $\mathscr{T}$, *transactions* $T_i, T_j \in \mathscr{T}$, $\xi(\mathscr{T}, T_i, T_j)$ *is self-contained iff given any transactions* $T_k, T_l \in \xi(\mathscr{T}, T_i, T_j)$, *s.t.* $k \neq l$ *and* $\exists e^k_u = \circ s_k(x)v \in \mathscr{T}|T$ $\exists e^l_v = g_l(x)v' \in \mathscr{T}|T$ *and* $e^k_u \prec_{\mathscr{T}} e^l_v$, *then either* $v = v'$ *or* $\exists e^m_u = \circ s_m(x)v' \in \mathscr{T}|T_m$ *for some* $T_m \in \xi(\mathscr{T}, T_i, T_j)$ *s.t.* T_m *precedes* T_l *and follows* T_k *in* $\xi(\mathscr{T}, T_i, T_j)$ *and* $e^k_u \prec_{\mathscr{T}} e^m_u \prec_{\mathscr{T}} e^l_v$.

Definition 52 (Chain Consistency)**.** *An isolated trace* $\mathscr{T}$ *is chain-consistent if given any* $\xi(\mathscr{T}, T_i, T_j)$, *trace* $\mathscr{T}$ *is chain-isolated and self-contained (for some* $T_i, T_j \in \mathscr{T}$*).*

Finally, a trace is *harmonious* if it satisfies all the preceding definitions.

Definition 53 (Harmony)**.** *Trace $\mathscr{T}$ is harmonious iff it satisfies all of the following: a) minimalism, b) consonance, c) obbligato, d) coherence, commit accord, abort accord, and abort coda, e) isolation, f) decisiveness, g) chain consistency, and h) unique writes.*

7.3.3 Last-use Opacity through Trace Harmony

Theorem 9 (Harmonious Trace Last-use Opacity)**.** *Given history H, s.t. $H = Hist(\mathscr{T})$, if $\mathscr{T}$ is harmonious, H is last-use opaque.*

The proof for Theorem 9 is given in Appendix A.

7.3.4 OptSVA+R Trace Harmony

Let $\bar{\mathscr{T}}$ be any trace produced by OptSVA+R.

First, we make the assumption that when write operations are executed, the values written to shared variables comply with their type. It can be assumed that the necessary type checking would be performed by the compiler, and an operation writing a value outside of the variable's domain would never be executed.

Assumption 5 (Writes Within Domain)**.** *Given any operation execution $w_x(v)\square\to\in\bar{\mathscr{T}}$, if D is the domain of $\mathtt{S}_{\lceil x \rfloor}$, then $v \in D$.*

Observation 8 (Memory Access Pattern)**.** *OptSVA+R generates view and update events for variable x precisely as a result of executing the following lines:*

- *in procedure* `:checkpoint` *at line 123—view event,*
- *in procedure* `:read_buffer` *at line 49—view event,*
- *in procedure* `:write_buffer` *at line 65—routine update event,*
- *in procedure* `commit` *at line 85—routine update event,*
- *in procedure* `abort` *at line 127—recovery update event.*

Observation 9 (Closing Write Identification)**.** *If after executing a write operation on x by T_i it is true that $\mathtt{wub}_i(x) = \mathtt{wc}_i(x)$, then that is the closing write operation execution on x in T_i.*

Lemma 39 (Version Order)**.** *Any two transactions $T_i, T_j \in \bar{\mathscr{T}}$ s.t. $\mathtt{ASet}_i \cap \mathtt{ASet}_j \neq \varnothing$ are* isolation ordered*: if $\exists x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$, s.t. $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$, then $\forall y \in \mathtt{ASet}_i \cap \mathtt{ASet}_j, \mathtt{pv}_i(y) < \mathtt{pv}_j(y)$.*

Proof. During `start` every transaction acquires a value of $\mathtt{pv}_i(x)$. Since the acquisition is guarded by locks, it is performed atomically, so that if transaction T_i, starts acquiring $\forall x \in \mathtt{ASet}_i, \mathtt{pv}_i(x)$, then no other T_j acquires $\forall x \in \mathtt{ASet}_j \cap \mathtt{ASet}_i, \mathtt{pv}_j(x)$ until transaction T_i completes acquiring and releases the locks. Hence, if for any two T_i, T_j, if $\exists x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$ $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$, then $\forall y \in \mathtt{ASet}_i \cap \mathtt{ASet}_j, \mathtt{pv}_i(y) < \mathtt{pv}_j(y)$. □

Corollary 20 (Version Order from Isolation Order)**.** *Given transactions T_i, T_j s.t. $T_i \dot{\prec}_{\bar{\mathscr{T}}} T_j$, then $\forall x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j, \mathtt{pv}_j(x) < \mathtt{pv}_i(x)$.*

Lemma 40 (Minimalism)**.** *$\bar{\mathscr{T}}$ is minimalistic.*

Proof. If x is read-only in T_i, then there is exactly one view event on x in T_i (line 49). If x is not read-only, then there is exactly one view event on x in T_i executed as part

of procedure :checkpoint (line 123), either during the first read, the closing write (in :write_buffer), or, if not previously invoked, during commit.

Routine update events are executed only after the closing write (in :write_buffer—line 65), so at most once, or during commit (line 85), if there were writes, but the upper bound on writes was not reached. Hence, routine update events occur at most once per variable.

A recovery update event can occur only during abort (line 127), at most once per variable. □

Lemma 41 (Obligatory Checkpoints)**.** *If T_i issues an update event or a view update event, T_i invoked* :checkpoint*.*

Proof. View events are only executed as part of :checkpoint.

A routine update event is only executed as part of :write_buffer at line 65, which is dominated by lines 61–62, which executes checkpoint if it was not previously executed.

A recovery update event occurs as a result of executing line 127, which is guarded by a condition that $\mathtt{wc}_i(x) > 0$, so a write must have been executed. Furthermore, $\mathtt{pv}_i(x) - 1 > \mathtt{lv}(x)$ must be true, which implies that T_i released x, which means the closing write executed, so :write_buffer was started asynchronously. That procedure executes a :checkpoint if it was not executed beforehand at lines 61–62. □

Lemma 42 (Always View Before Update)**.** *If transaction T_i issues an update event $e_u = s_i(x)\square$ in trace $\bar{\mathscr{T}}$, then there is $e_v = g_i(x)\square \in \bar{\mathscr{T}}|T_i$ s.t. $e_v \prec_{\bar{\mathscr{T}}} e_u$.*

Proof. From Lemma 41, if T_i executes an update event, then it executes :checkpoint before the event is issued. Since :checkpoint issues a view event, then a view event is issued before an update event. □

Lemma 43 (Wait at Access)**.** *Given transactions T_i, T_j s.t. $\mathtt{pv}_j(x) < \mathtt{pv}_i(x)$, T_i does not issue a view or update event on x until T_j executes* :release *on x,* abort*, or* commit*.*

Proof. Let T_k be such that $\mathtt{pv}_k(x) = \mathtt{pv}_i(x) - 1$. Every invocation of :checkpoint is dominated by an instruction that waits until the condition $\mathtt{pv}_i(x) - 1 = \mathtt{lv}(x)$: line 23 by line 22, line 81 by line 79, and line 62 by line 43. Since, from Lemma 41, every view or update event is preceded by the invocation of :checkpoint, then each view or update event is dominated by an instruction that waits until $\mathtt{pv}_i(x) - 1 = \mathtt{lv}(x)$. Hence in order for T_i to issue a event it must be true that $\mathtt{pv}_i(x) - 1 = \mathtt{lv}(x)$.

In order for that condition to be met, some transaction must set $\mathtt{lv}(x)$ to $\mathtt{pv}_i(x) - 1$ (or $\mathtt{pv}_i(x) = 1$, but then there could not be such T_j as assumed). Some transaction T_k modifies a new value of $\mathtt{lv}(x)$ during :release, abort, or commit and the value is there set to $\mathtt{pv}_k(x)$. Hence T_i cannot issue any view or update event until some T_k such that $\mathtt{pv}_k(x) = \mathtt{pv}_i(x) - 1$ executes :release, abort, or commit.

Every invocation of :release (by T_k) is dominated by an instruction that waits until the condition $\mathtt{pv}_k(x) - 1 = \mathtt{lv}(x)$ is met: the invocation at line 50 by line 13, and the one at line 66 by line 43. Furthermore, modifying $\mathtt{lv}(x)$ within commit (line 117) or abort (line 117) also requires that $\mathtt{pv}_k(x) - 1 = \mathtt{lv}(x)$ be first satisfied (at line 116 and line 116, respectively). Hence T_k cannot set $\mathtt{lv}(x)$ to $\mathtt{pv}_k(x)$ view or update event unless $\mathtt{pv}_k(x) = 1$ or until some T_l such that $\mathtt{pv}_l(x) = \mathtt{pv}_k(x) - 1$ executes :release, abort, or commit.

Assuming that $\mathtt{pv}_k(x) > 1$, and that some T_l s.t. $\mathtt{pv}_l(x) = \mathtt{pv}_k(x) - 1$ exists, then, since T_i cannot issue any view or update event until T_k sets $\mathtt{lv}(x)$ in :release, abort, or commit and since T_k cannot set $\mathtt{lv}(x)$ until T_l executes :release, abort, or commit, then T_i cannot issue any view or update events until T_l executes :release, abort, or commit. Since $\mathtt{pv}_i(x) - 1 = \mathtt{pv}_k(x)$ and $\mathtt{pv}_k(x) - 1 = \mathtt{pv}_l(x)$ then $\mathtt{pv}_l(x) < \mathtt{pv}_i(x)$.

It follows by induction then that given any T_j, s.t. $\mathtt{pv}_j(x) < \mathtt{pv}_i(x)$, T_i does not issue a view or update event on x until T_j executes :release, abort, or commit. □

Lemma 44 (Recovery Versions from Version Order)**.** *Given transactions T_i, T_j s.t. $\mathtt{pv}_j(x) < \mathtt{pv}_i(x)$, if T_j executes* abort *before T_i executes* :checkpoint*, $\mathtt{rv}_j(x) \leqslant \mathtt{rv}_i(x)$. otherwise $\mathtt{rv}_j(x) < \mathtt{rv}_i(x)$.*

Proof. Transaction T_i sets $\mathtt{rv}_i(x)$ to $\mathtt{cv}(x)$ only during :checkpoint (line 124). Every invocation of :checkpoint is dominated by an instruction that waits until the condition $\mathtt{pv}_i(x) - 1 = \mathtt{lv}(x)$: line 23 by line 22, line 81 by line 79, and line 62 by line 43.

In order for that condition to be met, some transaction must set $\mathtt{lv}(x)$ to $\mathtt{pv}_i(x)-1$ (or $\mathtt{pv}_i(x) = 1$, but then there could not be such T_j as assumed, so necessarily $\mathtt{pv}_i(x) > 1$). Some transaction T_k can set a new value of $\mathtt{lv}(x)$ during :release, abort, or commit. Hence T_i sets the value of $\mathtt{rv}_i(x)$ only after T_k such that $\mathtt{pv}_k(x) = \mathtt{pv}_i(x) - 1$ executes :release, abort, or commit. Thus, since value of $\mathtt{cv}(x)$ is there set by T_k to $\mathtt{pv}_k(x)$ in case of :release (line 131) and commit (line 120), or $\mathtt{rv}_k(x)$ in case of abort (line 128), $\mathtt{rv}_i(x) = \mathtt{rv}_k(x)$ if T_k aborts before T_i executes :checkpoint and $\mathtt{rv}_i(x) = \mathtt{pv}_k(x)$ otherwise.

Since $\mathtt{rv}_k(x)$ either trivially equals 0 if $\mathtt{pv}_k(x) = 1$, or is acquired by analogy from some T_l s.t. $\mathtt{pv}_l(x) = \mathtt{pv}_k(x) - 1$, then $\mathtt{rv}_k(x) \leqslant \mathtt{rv}_i(x)$.

Furthermore, under the assumption that T_k does not execute abort prior to T_i executing :checkpoint, then value of $\mathtt{cv}(x)$ is there set by T_k only either within :release or commit, and thus $\mathtt{cv}(x) = \mathtt{pv}_k(x)$ during T_i's :checkpoint, so $\mathtt{rv}_i(x) = \mathtt{pv}_k(x)$. Since $\mathtt{pv}_k(x) < \mathtt{pv}_i(x)$, then $\mathtt{rv}_k(x) < \mathtt{rv}_i(x)$.

By extension, given T_j s.t. $\mathtt{pv}_j(x) < \mathtt{pv}_i(x)$, either $k = j$, or $\mathtt{pv}_j(x) < \mathtt{pv}_k(x)$.

In the former case, necessarily $\mathtt{rv}_k(x) \leqslant \mathtt{rv}_i(x)$ if T_j executes abort before T_i executes :checkpoint, or $\mathtt{rv}_k(x) < \mathtt{rv}_i(x)$.

In the latter case, there must be some T_l, s.t. $\mathtt{pv}_l(x) = \mathtt{pv}_k(x)$. Then, if T_l executes abort before T_k executes :checkpoint, $\mathtt{rv}_l(x) \leqslant \mathtt{rv}_k(x)$, otherwise $\mathtt{rv}_l(x) < \mathtt{rv}_k(x)$. Furthermore, either $l = j$, or $\mathtt{pv}_j(x) < \mathtt{pv}_l(x)$. It then follows by induction that given any T_j s.t. $\mathtt{pv}_j(x) < \mathtt{pv}_i(x)$, if T_j executes abort before T_i executes :checkpoint, $\mathtt{rv}_l(x) \leqslant \mathtt{rv}_k(x)$, otherwise $\mathtt{rv}_l(x) < \mathtt{rv}_k(x)$. □

Lemma 45 (Isolation)**.** *Trace $\bar{\mathscr{T}}$ is isolated.*

Proof. Every routine update event, view event, and recovery event is dominated by an access condition ($\mathtt{pv}_i(x) - 1 = \mathtt{lv}(x)$). This condition is satisfied for T_i if $\mathtt{lv}(x) = 0$ and $\mathtt{pv}_i(x) = 0$, or if some transaction T_j s.t. $\mathtt{pv}_j(x) = \mathtt{pv}_i(x) - 1$ releases x by setting $\mathtt{lv}(x)$ to $\mathtt{pv}_j(x)$ during commit or after closing write or after the first non-local read (and thus after any $\circ s_j(x)\square$ or $g_j(x)\square$).

Since events are guarded by access conditions, and variables are released after all view or routine update events are issued by a transaction, and since all transactions are version-ordered, then for any T_i, T_j, $\exists x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$ if $\exists e_i = \circ s_i(x)\square \in \bar{\mathscr{T}}|T_i$ or $e_i = g_i(x)\square \in \bar{\mathscr{T}}|T_i$ and $\exists e_j = \circ s_j(x)\square \in \bar{\mathscr{T}}|T_j$ or $e_j = g_j(x)\square \in \bar{\mathscr{T}}|T_j$, and $e_i \prec_{\mathscr{T}} e_j$ then $\forall y \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$, if $\exists e'_i = \circ s_i(y)\square \in \bar{\mathscr{T}}|T_i$ or $e'_i = g_i(y)\square \in \bar{\mathscr{T}}|T_i$ and $\exists e'_j = \circ s_j(y)\square \in \bar{\mathscr{T}}|T_j$ or $e'_j = g_j(y)\square \in \bar{\mathscr{T}}|T_j$, and $e'_i \prec_{\mathscr{T}} e'_j$. □

Corollary 21 (Isolation Order)**.** *Trace $\bar{\mathscr{T}}$ is isolation-ordered.*

Lemma 46 (Write Consonance)**.** *Any (complete) write operation in $\bar{\mathscr{T}}$ is consonant.*

Proof. From Assumption 5, each write is consonant. □

Lemma 47 (Routine Update Consonance)**.** *Any routine update event in $\bar{\mathscr{T}}$ is consonant.*

Proof. A routine update event $\circ s_i(x)v$ occurs either as a result of executing a closing write operation on x (line 65) or T_i committing (line 85), if the transaction executed writes, but the upper bound for writes was not reached for x. Clearly, then, if there was a routine update event, then T_i executed a write operation on x. In both cases above $v = \texttt{buf}_i(x)$ and $\texttt{buf}_i(x)$ can be set by any write operation, the first non-local read operation, or during start for read-only variables. If there was a write, then x is not read-only, and the first non-local read cannot follow a write, so $\texttt{buf}_i(x)$ is set within (the most recent) write operation executed by T_i and corresponds to the value written by that operation. Thus, $\forall T_i, \forall e_u^i = \circ s_i(x)v \in \bar{\mathscr{T}}|T_i$, $\exists op_i = w_i(x)v \to ok_i \in \bar{\mathscr{T}}|T_i$ s.t. $op_i \rightsquigarrow e_u^i$. Therefore, e_u^i is consonant. □

Lemma 48 (View Consonance)**.** *Any view event in $\bar{\mathscr{T}}$ is consonant.*

Proof. If a view event occurs, it views the current state of a variable. So given transaction T_i, if there is a view event $e_v = g_i(x)v \in \bar{\mathscr{T}}|T_i$, v corresponds to the current state of x. The only way to change the state of x is via an update event on x. Thus, trivially, for some T_j, if there is $e_u = \circ s_j(x)v' \in \bar{\mathscr{T}}|T_j$ or $e_a = \boxplus s_j(x)v' \in \bar{\mathscr{T}}|T_j$, if either e_u or e_a precede e_v so that no other update event on x occurs between either e_u or e_a and e_v, then $v = v'$. Furthermore, from unique routine updates, there is no e_u s.t. $v' = 0$, and since x is initially 0, then $\nexists e_u$ s.t. $v' = 0$ and $e_u \lessdot e_v$. □

Lemma 49 (Local Read Consonance)**.** *Any local read operation execution in $\bar{\mathscr{T}}$ is consonant.*

Proof. If transaction executes a local read on x, then it previously executed a write operation on x, so $\texttt{wc}_i(x) > 0$. Thus, the read procedure returns at line 31, returning $\texttt{buf}_i(x)$. The value of $\texttt{buf}_i(x)$ can be set by any write operation, the first non-local read operation, or during transaction start for read-only variables. If there was a write, then x is not read-only, and the first non-local read cannot follow a write, so $\texttt{buf}_i(x)$ is set within (the most recent) write operation executed by T_i and corresponds to the value written by that operation. Thus, $\forall T_i, \forall op^i = r_i(x) \to v \in \bar{\mathscr{T}}|T_i$ if op_i is local, $\exists op_i' = w_i(x)v' \to ok_i \in \bar{\mathscr{T}}|T_i$ s.t. $v' = v$. Therefore, op_i is consonant. □

Lemma 50 (Non-local Read Consonance)**.** *Any non-local read operation execution in $\bar{\mathscr{T}}$ is consonant.*

Proof. If x is read-only in T_i, then during `start`, a view event occurs within `:read_buffer` (line 49), and the state of x is saved in $\texttt{buf}_i(x)$. Then, subsequent writes return the value of $\texttt{buf}_i(x)$ (line 31) (waiting if necessary). Thus, they depend on that view event.

Otherwise, a non-local read operation on x is one that is not preceded by a write on x, so $\texttt{wc}_i(x) = 0$. The first such read executes `:checkpoint` which initiates a view event (line 123). The value obtained by that event is saved in $\texttt{st}_i(x)$ and later $\texttt{buf}_i(x)$ is set to the same value. Finally, that value is returned at line 31. Subsequent non-local reads use the same value stored in $\texttt{buf}_i(x)$. The value remains unchanged, since it only be overwritten by a write, but the occurrence of a preceding write would mean the read is local (and since x is not read-only, and there was a preceding non-local read). Thus, all non-local reads depend on the view event issued during `:checkpoint`.

Thus, $\forall T_i, \forall op^i = r_i(x) \to v \in \bar{\mathscr{T}}|T_i$ if op_i is non-local, $\exists e_v^i = g_i(x)v' \in \bar{\mathscr{T}}|T_i$ s.t. $v' = v$. □

Lemma 51 (Conservative Recovery Update Events)**.** *Any recovery update event in $\bar{\mathscr{T}}$ is conservative.*

Proof. The recovery update event in T_i occurs as a result of executing line 127, which updates the state of x to $\mathtt{st}_i(x)$. This is done only if $\mathtt{rv}_i(x) \neq \mathtt{cv}(x)$ and $\mathtt{wc}_i(x) > 0$. Since $\mathtt{rv}_i(x)$ is set to the value of $\mathtt{cv}(x)$ only during :checkpoint and :read_buffer, and since the requirement that $\mathtt{wc}_i(x) > 0$ excludes the latter, this condition checks whether the current transaction previously made a checkpoint. Executing :checkpoint entails a view event that sets $\mathtt{st}_i(x)$ to the current value of x. Hence, if $e_a = \text{¤}s_i(x)v \in \mathscr{T}|T_i$ then there exists $e_v = g_i(x)v \in \mathscr{T}|T_i$ s.t. $e_a \prec e_v$. □

Lemma 52 (Clean Recovery Update Events)**.** *Any recovery update event in $\bar{\mathscr{T}}$ is clean.*

Proof. Assume by contradiction that there exists $e_a = \text{¤}s_i(x)v$ in $\bar{\mathscr{T}}|T_i$ and $e_v = g_i(x)v$ that justifies that e_s is conservative, and $e'_a = \text{¤}s_j(x)v'$ in $\bar{\mathscr{T}}|T_j$ s.t. $T_j \dot{\prec}^x_{\mathscr{T}} T_i$ and $e_v \prec_{\mathscr{T}} e'_a \prec_{\bar{\mathscr{T}}} e_a$. This implies that T_j executes abort (and satisfies the condition $\mathtt{rv}_j(x) = \mathtt{cv}(x)$) between the point at which T_i executes :checkpoint and abort If that is the case, as a result of executing abort, T_j sets $\mathtt{cv}(x)$ to $\mathtt{rv}_j(x)$.

Given that $T_j \dot{\prec}^x_{\mathscr{T}} T_i$, then $\mathtt{pv}_j(x) < \mathtt{pv}_i(x)$. Since any execution of :checkpoint for some T_k is guarded by the condition $\mathtt{pv}_k(x) - 1 = \mathtt{lv}(x)$, then T_j executes :checkpoint before T_i. Hence, T_j acquires $\mathtt{rv}_j(x)$ from $\mathtt{cv}(x)$ before T_i acquires $\mathtt{rv}_i(x)$ from $\mathtt{cv}(x)$.

The value of $\mathtt{rv}_j(x)$ is equal to the value of $\mathtt{cv}(x)$ at the point when T_j executed :checkpoint (i.e. when $\mathtt{pv}_i(x) - 1 = \mathtt{lv}(x)$). The value of $\mathtt{cv}(x)$ is set to $\mathtt{pv}_k(x)$ when T_k executes :release or commit, or to $\mathtt{rv}_k(x)$ when T_k aborts. Thus, when T_j executes :checkpoint, since $\mathtt{pv}_i(x) - 1 = \mathtt{lv}(x)$, then either:

a) $\mathtt{cv}(x) = \mathtt{pv}_k(x) = \mathtt{pv}_j(x) - 1$ (if T_k released x or committed),

b) $\mathtt{cv}(x) = \mathtt{rv}_k(x)$ and $\mathtt{rv}_k(x) < \mathtt{pv}_j(x)$ (if T_k aborted), or

c) $\mathtt{cv}(x) = 0$ (if there is no such T_k).

In any case, $\mathtt{rv}_j(x) < \mathtt{pv}_i(x)$.

T_i is capable of executing :checkpoint after T_j commits, aborts, or releases x. Since T_j executes abort between T_i's :checkpoint and abort, then only the third option remains. If T_j executes :release for x, then it sets $\mathtt{cv}(x)$ to $\mathtt{pv}_j(x)$. Following the logic from the previous paragraph, this means that when T_i assigns $\mathtt{cv}(x)$ to $\mathtt{rc}_i(x)$, $\mathtt{pv}_j(x) \leqslant \mathtt{cv}(x)$, so $\mathtt{pv}_j(x) \leqslant \mathtt{rv}_i(x)$, and thus $\mathtt{rv}_j(x) < \mathtt{rv}_i(x)$.

Hence, after $\mathtt{cv}(x)$ to $\mathtt{rv}_j(x)$ during abort, it is not true that $\mathtt{cv}(x) = \mathtt{rv}_i(x)$. Thus, e_a cannot occur once e'_a occurs, which is a contradiction. □

Lemma 53 (Needed Recovery Update Events)**.** *Any recovery update event in $\bar{\mathscr{T}}$ is needed.*

Proof. The recovery update event occurs as a result of executing line 127, which is guarded by a condition that $\mathtt{wc}_i(x) > 0$, so a write must have been executed. Furthermore, $\mathtt{pv}_i(x) - 1 > \mathtt{lv}(x)$ must be true, which implies that T_i released x, which means the closing write executed, so :write_buffer was started asynchronously. If that is the case, the recovery update event cannot execute until :write_buffer, which means a routine update event on x will have executed before the recovery update event on x. □

Lemma 54 (Dooming Recovery Update Events)**.** *Any recovery update event in $\bar{\mathscr{T}}$ is dooming.*

Proof. Trivially, since any recovery update event occurs only within abort. □

Lemma 55 (Ending Recovery Update Events)**.** *Any recovery update event in $\bar{\mathscr{T}}$ is ending.*

Proof. Trivially, since any recovery update event occurs only within abort, and there are no other update or view events on the same variable in abort. □

Lemma 56 (Recovery Update Consonance)**.** *Any recovery update event in $\bar{\mathscr{T}}$ is consonant.*

Proof. Since each recovery update is conservative (from Lemma 51), needed (from Lemma 53), dooming (from Lemma 54), ending (from Lemma 55), and clean (from Lemma 52), then each recovery update is consonant. □

Lemma 57 (Trace Consonance)**.** *Trace $\bar{\mathscr{T}}$ is consonant.*

Proof. From Lemmas 46–50 and 56. □

Lemma 58 (Comitted Write Obbligato)**.** *Given T_i that is committed in $\bar{\mathscr{T}}$, every non-local write operation execution $op_i = w_i(x)v \to ok_i \in \bar{\mathscr{T}}|T_i$ is in committed obbligato.*

Proof. If T_i executes a write corresponding to op_i, then, if at the end of the execution it is true that $\mathtt{wc}_i(x) = \mathtt{wub}_i(x)$, :write_buffer is executed, which causes a routine update event to execute, writing the value of $\mathtt{buf}_i(x)$ to x.

Since $\mathtt{wc}_i(x) = \mathtt{wub}_i(x)$ no other writes follow, and since x is not read-only in T_i, then the value written to x in :write_buffer is the value passed to the write operation. In that case there is $e_u = \circ s_i(x)v \in \bar{\mathscr{T}}|T_i$. Since commit will not return until :write_buffer finishes executing, then trivially $inv_i\big[w_i(x)v\big] \prec_{\bar{\mathscr{T}}|T_i} e_u \prec_{\bar{\mathscr{T}}|T_i} res_i\big[C_i\big]$.

If it is true that $\mathtt{wc}_i(x) = \mathtt{wub}_i(x)$, then :write_buffer is not executed, but during commit, the same condition is checked again, and if it is not satisfied, T_i writes the value from $\mathtt{buf}_i(x)$ to x. Thus, by analogy to the paragraph above, there is $e_u = \circ s_i(x)v \in \bar{\mathscr{T}}|T_i$. Since this is executed within commit, then $inv_i\big[w_i(x)v\big] \prec_{\bar{\mathscr{T}}|T_i} e_u \prec_{\bar{\mathscr{T}}|T_i} res_i\big[C_i\big]$. □

Lemma 59 (Closing Write Obbligato)**.** *Given T_i that is decided on x in $\bar{\mathscr{T}}$, every non-local write operation execution $op_i = w_i(x)v \to ok_i \in \bar{\mathscr{T}}|T_i$ is in closing write obbligato.*

Proof. If T_i executes a write corresponding to op_i, then, at the end of the execution, if op_i is a closing write it is necessarily true that $\mathtt{wc}_i(x) = \mathtt{wub}_i(x)$. This causes :write_buffer to be executed (line 43), which causes a routine update event to execute, writing the value of $\mathtt{buf}_i(x)$ to x.

Since $\mathtt{wc}_i(x) = \mathtt{wub}_i(x)$ no other writes follow, and since x is not read-only in T_i, then the value written to x in :write_buffer is the value passed to the write operation. In that case there is $e_u = \circ s_i(x)v \in \bar{\mathscr{T}}|T_i$. Since commit will not return until :write_buffer finishes executing, then trivially $inv_i\big[w_i(x)v\big] \prec_{\bar{\mathscr{T}}|T_i} e_u \prec_{\bar{\mathscr{T}}|T_i} res_i\big[C_i\big]$. Hence $op_i \rightsquigarrow e_u$. T_i executes :release only following issuing e_u at line 66.

If $T_i \dot{\prec}_{\bar{\mathscr{T}}} T_j$, then $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$ (Corollary 20). From Lemma 43, to issue any $e_v = g_j(x)\Box$, since $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$, T_i must have executed abort, commit, or :release. Hence T_j does not issue e_v before T_i executes :release, which requires that T_i issues e_u so that $e_u \prec_{\bar{\mathscr{T}}} e_v$. □

Lemma 60 (View Write Obbligato)**.** *Given $T_i \in \bar{\mathscr{T}}$, if $\exists T_j \in \mathscr{T}$, s.t. $T_i \dot{\prec}_{\bar{\mathscr{T}}} T_j$, if there is $op_i = w_i(x)\Box \to ok_i \in \bar{\mathscr{T}}|T_i$, and $e_v = g_j(x)\Box \in \bar{\mathscr{T}}|T_j$, then op_i is in view write obbligato.*

Proof. If $T_i \dot{\prec}_{\bar{\mathscr{T}}} T_j$, then $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$ (Corollary 20). From Lemma 43, e_v occurs only after T_i releases x, commits, or aborts. Since according to the assumption, T_i cannot abort prior to T_j issuing e_v, T_i either releases x or commits prior to T_j issuing e_v.

If T_i releases x it executes :release. This can occur as a result of executing line 50 or line 66. Since T_i executes op_i, then line 50 cannot be executed, since it can only be reached if T_i only ever reads x (condition at line 11). Hence T_i must execute line 66, which is dominated by line 65, which issues a write event $e_u = s_i(x)v$, where v is the value of $\mathtt{buf}_i(x)$.

Since :release was executed at line 66, :write_buffer must have been executed at line 43. Then the value written to x in :write_buffer is the value passed to the write operation. In that case there is $e_u = \circ s_i(x)v \in \bar{\mathscr{T}}|T_i$. Since commit will not return until :write_buffer finishes executing, then trivially $inv_i\big[w_i(x)v\big] \prec_{\bar{\mathscr{T}}|T_i} e_u \prec_{\bar{\mathscr{T}}|T_i} res_i\big[C_i\big]$. Hence $op_i \rightsquigarrow e_u$. T_i executes :release only following issuing e_u at line 66. □

Lemma 61 (Obbligato)**.** *Trace $\bar{\mathscr{T}}$ is obbligato.*

Proof. From Lemmas 58, 59, and 60. □

Lemma 62 (Decisiveness)**.** *Trace $\bar{\mathscr{T}}$ is decisive.*

Proof. If $T_j \dot{\curvearrowleft} T_i$, then for any $x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$, $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$ (Lemma 45). Before any view event occurs, T_j must pass the condition $\mathtt{pv}_k(x) - 1 = \mathtt{lv}(x)$ in :read_buffer or :checkpoint. Hence, before any $e_v = g_j(x)v$ can occur, some T_k s.t. $\mathtt{pv}_i(x) - 1 = \mathtt{pv}_k(x)$ must set $\mathtt{pv}_k(x)$ to $\mathtt{lv}(x)$. Transaction T_j issues a routine update event $e_u = \circ s_j(x)v'$, whenever it commits or releases x. If it releases, it means that $\mathtt{wc}_j(x) = \mathtt{wub}_j(x)$, which implies that op_i is closing. Otherwise, T_j will update on commit, meaning that it will issue $res_j\big[C_j\big]$ once it returns from the commit procedure. Before returning from the closing write or commit, T_j sets $\mathtt{lv}(x)$ to $\mathtt{pv}_j(x)$. In either case, this happens only afterward e_u is issued. Since there is no waiting between e_u and either a commit or a last write returning, no other transaction may execute anything on x in the meantime. Thus, any transaction T_k s.t. $\mathtt{pv}_k(x)$ that waits until $\mathtt{pv}_k(x) - 1 = \mathtt{lv}(x)$ and $\mathtt{pv}_k(x) - 1 = \mathtt{pv}_j(x)$ will wait until T_j returns from the closing write or commit procedure, and so will any subsequent transactions according to version order. Thus, if $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$ and T_j commits, then either op_j is closing or $e_u \prec_{\bar{\mathscr{T}}} res_j\big[C_j\big] \prec e_v$. □

Lemma 63 (Abort Accord)**.** *Trace $\bar{\mathscr{T}}$ is in abort accord.*

Proof. Let T_i, T_j be two transactions in $\bar{\mathscr{T}}$, s.t.:

a) $T_j \dot{\curvearrowleft} T_i$, T_i is aborted in $\bar{\mathscr{T}}$.

Assume by contradiction that T_j commits in $\bar{\mathscr{T}}$, meaning it executes commit successfully. Thus it passes $\forall x, \mathtt{cv}(x) \geqslant \mathtt{rv}_j(x)$.

Since $T_j \dot{\curvearrowleft} T_i$, $\exists e_v = g_j(x)v \in \bar{\mathscr{T}}|T_j$ and $\exists e_u = \circ s_i(x)v \in \bar{\mathscr{T}}|T_i$.

If T_i aborted before e_v was issued, then from abort coda, $\exists e_a = \natural s_k(x)\square$ in some T_k that precedes the abort, which contradicts that $T_j \dot{\curvearrowleft} T_i$. Hence T_i aborts only after e_v is issued.

Since $T_j \dot{\curvearrowleft} T_i$, then $T_i \dot{\prec}^x_{\mathscr{T}} T_j$, so from Corollary 20, $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$. From Lemma 43, e_v cannot occur until T_i aborts, commits, or releases x. Since T_i aborts after e_v, then it must therefore release x prior to e_v.

Since $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$, then from Lemma 44, $\mathtt{rv}_i(x) < \mathtt{rv}_j(x)$. When T_i aborts, it sets $\mathtt{cv}(x)$ to $\mathtt{rv}_i(x)$. From coherence, T_j commits after T_i aborts. Thus, when T_j commits, $\mathtt{cv}(x) = \mathtt{rv}_i(x)$, so since $\mathtt{rv}_i(x) < \mathtt{rv}_j(x)$, then $\mathtt{cv}(x) < \mathtt{rv}_j(x)$, which contradicts the condition that $\mathtt{cv}(x) > \mathtt{rv}_j(x)$.

Thus T_j cannot commit.

b) $\exists e_u = \circ s_i(x)\square \in \bar{\mathscr{T}}|T_i$ and $e = \circ s_j(x)\square \in \bar{\mathscr{T}}|T_j$ or $e = g_j(x)\square \in \bar{\mathscr{T}}|T_j$ and $e_a = \natural s_i(x)\square \in \bar{\mathscr{T}}|T_i$, and $e_u \prec_{\bar{\mathscr{T}}} e \prec_{\bar{\mathscr{T}}} e_a$.

Assume by contradiction that T_j commits in $\bar{\mathscr{T}}$, meaning it executes commit successfully. Thus it passes $\forall x, \mathtt{cv}(x) \geqslant \mathtt{rv}_j(x)$.

Since $e_u \prec_{\bar{\mathscr{T}}} e$, then from isolation it follows that $T_i \dot{\prec}^x_{\mathscr{T}} T_j$. Hence, from Corollary 20, $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$. Since e_a must be issued during abort, then from coherence, T_j cannot commit prior to e_a occurring. Furthermore, T_j cannot commit until T_i returns from abort.

If T_i returned from `abort`, then it executed line 128, so $\mathtt{cv}(x) = \mathtt{rv}_i(x)$ prior to T_j committing.

From Lemma 44, since $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$, then $\mathtt{rv}_i(x) < \mathtt{rv}_j(x)$. When T_i aborts, it sets $\mathtt{cv}(x)$ to $\mathtt{rv}_i(x)$. From coherence, T_j commits after T_i aborts. Thus, when T_j commits, $\mathtt{cv}(x) = \mathtt{rv}_i(x)$, so since $\mathtt{rv}_i(x) < \mathtt{rv}_j(x)$, then $\mathtt{cv}(x) < \mathtt{rv}_j(x)$, which contradicts the condition that $\mathtt{cv}(x) > \mathtt{rv}_j(x)$.

Thus T_j cannot commit.

□

Lemma 64 (Commit Accord)**.** *Trace $\bar{\mathscr{T}}$ is in commit accord.*

Proof. Let T_i, T_j be transaction in $\mathscr{T}$ s.t. $T_j \circlearrowleft T_i$ and T_j is committed in $\bar{\mathscr{T}}$.

Let us assume by contradiction that T_i is not committed in $\bar{\mathscr{T}}$. So T_i is either aborted or live in $\bar{\mathscr{T}}$. From coherence, if T_j cannot commit until T_i commits or aborts. Thus T_i is not live in $\bar{\mathscr{T}}$, so it is aborted in $\bar{\mathscr{T}}$.

Since $T_j \circlearrowleft T_i$, $\exists e_v = g_j(x)v \in \bar{\mathscr{T}}|T_j$ and $\exists e_u = \circ s_i(x)v \in \bar{\mathscr{T}}|T_i$.

If T_i aborted before e_v was issued, then from abort coda, $\exists e_a = \natural s_k(x)\square$ in some T_k that precedes the abort, which contradicts that $T_j \circlearrowleft T_i$. Hence T_i aborts only after e_v is issued.

Since $T_j \circlearrowleft T_i$, then $T_i \dot{\prec}^x_{\mathscr{T}} T_j$, so from Corollary 20, $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$. From Lemma 43, e_v cannot occur until T_i aborts, commits, or releases x. Since T_i aborts after e_v, then it must therefore release x prior to e_v.

Since $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$, then from Lemma 44, $\mathtt{rv}_i(x) < \mathtt{rv}_j(x)$. When T_i aborts, it sets $\mathtt{cv}(x)$ to $\mathtt{rv}_i(x)$. From coherence, T_j commits after T_i aborts. Thus, when T_j commits, $\mathtt{cv}(x) = \mathtt{rv}_i(x)$, so since $\mathtt{rv}_i(x) < \mathtt{rv}_j(x)$, then $\mathtt{cv}(x) < \mathtt{rv}_j(x)$, which contradicts the condition that $\mathtt{cv}(x) > \mathtt{rv}_j(x)$.

Thus T_i cannot abort.

□

Lemma 65 (Abort Coda)**.** *Trace $\bar{\mathscr{T}}$ has coda.*

Proof. Let T_i be a transaction in $\bar{\mathscr{T}}$.

a) If T_i aborts in $\bar{\mathscr{T}}$ (so $r = res_i[A_i] \in \bar{\mathscr{T}}|T_i$) then if $\exists e_u = \circ s_x(v) \in \bar{\mathscr{T}}|T_i$ then for some T_j (possibly $i = j$) s.t. $j = i$ or $T_j \dot{\prec}^x_{\mathscr{T}} T_i$, $\exists e_a = \natural s_j(x)\square \in \bar{\mathscr{T}}|T_j$ s.t. $e_u \prec_{\bar{\mathscr{T}}} e_a \prec_{\bar{\mathscr{T}}} r$.

If there is such e_u, then T_i executes :`checkpoint` (Lemma 41). Since there is such e_u there is also a write operation execution on x in $\bar{\mathscr{T}}|T_i$ (Lemma 47), so $\mathtt{wc}_i(x) > 0$ (line 39).

If there is such e_u, then T_i executes :`release` for x or `commit`. Since T_i aborts, then `commit` is not possible, so T_i executes :`release` for x. Therefore T_i sets $\mathtt{lv}(x)$ to $\mathtt{pv}_x(i) - 1$.

i) If no other transaction modified $\mathtt{cv}(x)$ between the point at which T_i executed :`checkpoint` and `abort`, then $\mathtt{cv}(x) = \mathtt{rv}_i(x)$, thus during abort T_i satisfies the condition on line 105 and executes line 127, issuing the recovery event e_a. Since e_a is issued during `abort`, then $e_u \prec_{\bar{\mathscr{T}}} e_a \prec_{\bar{\mathscr{T}}} r$.

ii) If there is T_j s.t. T_j modifies $\mathtt{cv}(x)$ between the points at which T_i executed :`checkpoint` and `abort`, s.t. $T_j \dot{\prec}^x_{\mathscr{T}} T_i$, then T_j executes :`release`, `commit`, or `abort` between the points at which T_i executed :`checkpoint` and `abort`. For the sake of simplicity we assume that there is no other T_k that modifies $\mathtt{cv}(x)$ between those two points s.t. $T_k \dot{\prec}^x_{\mathscr{T}} T_i$.

Since T_i executes :checkpoint, it issues a view event $e_v = g_i(x)\square$. In addition, since $T_j \dot{\prec}^x_{\mathscr{T}} T_i$, then from Corollary 20, $\mathtt{pv}_j(x) < \mathtt{pv}_i(x)$. From Lemma 43, e_v cannot occur until T_j aborts, commits, or releases x. Since T_j is supposed to execute :release, abort, or commit after T_i executes :checkpoint, hence after e_v, then T_j must therefore release x prior to e_v. Hence T_j executes commit, or abort between the points at which T_i executed :checkpoint and abort.

If T_i executes commit, then in order to set $\mathtt{cv}(x)$ to $\mathtt{pv}_j(x)$, it must be true that $\mathtt{rv}_i(x) = \mathtt{cv}(x)$. But if T_j executed :release, then $\mathtt{cv}(x)$ is set to $\mathtt{pv}_j(x)$. Since $\mathtt{rv}_j(x) \neq \mathtt{pv}_j(x)$, then $\mathtt{rv}_i(x) \neq \mathtt{cv}(x)$, so T_j cannot set $\mathtt{cv}(x)$ as a result of a commit. Hence, T_j executes abort between the points at which T_i executed :checkpoint and abort.

If T_j executes abort, then this implies that T_j executes line 128, and therefore also line 127, thus T_j issues recovery event e_a during abort. Thus, $e_u \prec_{\bar{\mathscr{T}}} e_a \prec_{\bar{\mathscr{T}}} r$.

b) If T_i commits in $\bar{\mathscr{T}}$ (so $r = res_i\big[C_i\big] \in \bar{\mathscr{T}}|T_i$) then if $\exists e = \circ s_x(v) \in \bar{\mathscr{T}}|T_i$ or $e = g_x(v) \in \bar{\mathscr{T}}|T_i$ then for no T_j s.t. $j = i$ or $T_j \dot{\prec}^x_{\mathscr{T}} T_i$, there exists $e_a = \natural s_j(x)\square \in \bar{\mathscr{T}}|T_j$ s.t. $e_u \prec_{\bar{\mathscr{T}}} e_a \prec_{\bar{\mathscr{T}}} r$.

If there is such e, then T_i executes :checkpoint (Lemma 41). If T_i successfully commits, then this means that T_i satisfies the condition $\mathtt{cv}(x) \geqslant \mathtt{rv}_i(x)$.

Assume by contradiction that there is such e_a in some T_j. Since $e_a \in \mathscr{T}|T_j$, then T_j must execute line 127, which also means that it executes line 128 and therefore sets $\mathtt{cv}(x)$ to $\mathtt{rv}_j(x)$.

Since $T_j \dot{\prec}^x_{\mathscr{T}} T_i$, then from Corollary 20 $\mathtt{pv}_j(x) < \mathtt{pv}_i(x)$ and from Lemma 44, $\mathtt{rv}_j(x) < \mathtt{rv}_i(x)$. Thus, $\mathtt{cv}(x) < \mathtt{rv}_i(x)$ which contradicts that $\mathtt{cv}(x) \geqslant \mathtt{rv}_i(x)$. Thus there is no such T_j.

□

Lemma 66 (Coherence)**.** *Trace $\bar{\mathscr{T}}$ is coherent.*

Proof. If $T_i \dot{\prec}^x_{\mathscr{T}} T_j$, then $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$. In order to commit or abort, any T_k must pass the condition $\mathtt{pv}_k(x) - 1 = \mathtt{ltv}(x)$. In addition, each T_k sets $\mathtt{ltv}(x)$ to $\mathtt{pv}_k(x)$ only at the end of either committing or aborting. Hence, if T_k cannot commit or abort until some T_l s.t. $\mathtt{pv}_k(x) - 1 = \mathtt{pv}_l(x)$ finishes committing or aborting. Hence if T_j committed, it must have passed the condition $\mathtt{pv}_k(x) - 1 = \mathtt{ltv}(x)$, and since $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$, T_i must have committed or aborted before T_j committed. Thus, given $r_j = res_j\big[C_j\big] \in \bar{\mathscr{T}}|T_j$, then there is $r_i = res_i\big[C_i\big] \in \bar{\mathscr{T}}|T_i$ or $r_i = res_i\big[A_i\big] \in \bar{\mathscr{T}}|T_i$ and $r_i \prec_{\mathscr{T}} r_j$. □

Lemma 67 (Chain Isolation)**.** *Given trace $\bar{\mathscr{T}}$ and transactions $T_i, T_j \in \bar{\mathscr{T}}$ s.t. there is $\xi(\mathscr{T}, T_i, T_j)$, $\forall T_k \in \xi(\mathscr{T}, T_i, T_j)$ s.t. $e^k_u = \circ s_k(x)v$, there is no T_l s.t. $\exists e^l_a = \natural s_l(x)v'$ where $v = v'$ and e^l is between e^k and any other event in any transaction in $\xi(\mathscr{T}, T_i, T_j)$.*

Proof. Assume by contradiction that there exists T_l such that $\exists e^l = \natural s_l(x)v'$ and e^l_a is between e^k_u and any other event $e \in \bar{\mathscr{T}}|\xi(\bar{\mathscr{T}}, T_i, T_j)$. This means that either $e \in \bar{\mathscr{T}}|T_l$ and $e^k_u \prec_{\bar{\mathscr{T}}} e^l_a \prec_{\bar{\mathscr{T}}} e$, or $\exists T_n$ s.t. $e \in \bar{\mathscr{T}}|T_n$.

a) Assume $e \in \bar{\mathscr{T}}|T_l$ and $e^k_u \prec_{\bar{\mathscr{T}}} e^l_a \prec_{\bar{\mathscr{T}}} e$.

From Lemma 42 there is a view event $e^k_v = g_k(x)\square \in \bar{\mathscr{T}}|T_k$, and from minimalism there is only one such event in $\bar{\mathscr{T}}|T_k$, so e must be a recovery event $e = \natural s_k(x)v$. If T_l executes e^l_a, then from Lemma 53, $\exists e^l_u = \circ s_l(x)\square$ in $\bar{\mathscr{T}}|T_l$ s.t. $e^k_u \prec_{\bar{\mathscr{T}}} e^l_a$. Hence either $e^l_u \prec_{\bar{\mathscr{T}}} e^k_u$ or $e^k_u \prec_{\bar{\mathscr{T}}} e^l_u$. So either $T_l \dot{\prec}^x_{\mathscr{T}} T_k$ or $T_k \dot{\prec}^x_{\mathscr{T}} T_l$.

If $T_l \dot{\prec}^x_{\mathscr{T}} T_k$, then $e^l_u \prec_{\bar{\mathscr{T}}} e^k_u$, so from Lemma 43, e^k_u cannot occur until e^l_u executes :release, commit, or abort, and since $e^k_u \prec_{\bar{\mathscr{T}}} e^l_a$, then only :release is viable.

If $T_l \dot{\prec}^x_{\bar{\mathscr{T}}} T_k$, then from version order $\mathtt{pv}_l(x) < \mathtt{pv}_k(x)$, so from Lemma 44, $\mathtt{rv}_l(x) < \mathtt{rv}_k(x)$. In order for T_l to issue e_a, it must execute `abort` and satisfy the condition at line 105. This means that line 128 is executed, so $\mathtt{cv}(x) = \mathtt{rv}_l(x)$.

If subsequently T_k issues e_a, then it must also satisfy the condition at line 105, so $\mathtt{rv}_k(x) = \mathtt{cv}(x)$. But since $\mathtt{rv}_l(x) < \mathtt{rv}_k(x)$, then $\mathtt{cv}(x) < \mathtt{rv}_k(x)$, which contradicts that $\mathtt{cv}(x) = \mathtt{rv}_l(x)$.

The execution of a recovery event on x by T_l is dominated by line 102, which cannot be passed until $\mathtt{pv}_l(x) - 1 = \mathtt{ltv}(x)$. Any transaction T_n sets $\mathtt{ltv}(x)$ to $\mathtt{pv}_n(x)$ as a last action during `commit` (line 96) or `abort` (line 111). Hence T_l cannot proceed to abort until T_n finishes committing or aborting. Since T_n cannot execute line 96 or line 111 if line 88 or line 102 was passed, then T_n cannot proceed to commit or abort until some other T_m s.t $\mathtt{pv}_n(x) - 1 = \mathtt{pv}_m(x)$ committed or aborted. Hence T_l cannot execute a recovery event until any T_m s.t. $\mathtt{pv}_m(x) < \mathtt{pv}_l(x)$ committed or aborted.

If $T_k \dot{\prec}^x_{\bar{\mathscr{T}}} T_l$, then from version order $\mathtt{pv}_k(x) < \mathtt{pv}_l(x)$. Hence, T_l executes any events resulting from `abort` only after T_k returns from `abort` or `commit`. Hence if T_k executes e^k_a , then $e^k_a \prec_{\bar{\mathscr{T}}} e^l_a$, which contradicts that $e^l_a \prec_{\bar{\mathscr{T}}} e^k_a$.

Thus, regardless of whether $T_l \dot{\prec}^x_{\bar{\mathscr{T}}} T_k$ or $T_k \dot{\prec}^x_{\bar{\mathscr{T}}} T_l$ there is a contradiction. Therefore, T_l cannot issue such e^l_a between e^k_u and another event in $\bar{\mathscr{T}}|T_l$.

b) Assume $\exists T_n$ s.t. $e \in \bar{\mathscr{T}}|T_n$.

We assume without loss of generality that $T_n \ddot{\sim} T_k$. Thus, there is a view event e^n_v and possibly a routine update event e^n_u in $\bar{\mathscr{T}}|T_n$. From minimalism and Lemma 42: $e^n_v \prec_{\bar{\mathscr{T}}} e^n_u$. Also, since $T_n \ddot{\sim} T_k$, then $T_k \dot{\prec}^x_{\bar{\mathscr{T}}} T_n$, so from Corollary 20, $\mathtt{pv}_k(x) < \mathtt{pv}_n(x)$.

If T_l executes e^l_a, then from Lemma 53, $\exists e^l_u = \circ s_l(x)\square$ in $\bar{\mathscr{T}}|T_l$ s.t. $e^k_u \prec_{\bar{\mathscr{T}}} e^l_a$. Hence either $e^l_u \prec_{\bar{\mathscr{T}}} e^k_u$ or $e^k_u \prec_{\bar{\mathscr{T}}} e^l_u$. So either $T_k \dot{\prec}^x_{\bar{\mathscr{T}}} T_l \dot{\prec}^x_{\bar{\mathscr{T}}} T_n$ or $T_k \dot{\prec}^x_{\bar{\mathscr{T}}} T_n \dot{\prec}^x_{\bar{\mathscr{T}}} T_l$. Thus, from Corollary 20, either $\mathtt{pv}_k(x) < \mathtt{pv}_l(x) < \mathtt{pv}_n(x)$ or $\mathtt{pv}_k(x) < \mathtt{pv}_n(x) < \mathtt{pv}_l(x)$.

If $\mathtt{pv}_k(x) < \mathtt{pv}_l(x) < \mathtt{pv}_n(x)$, then either $e^l_a \prec_{\bar{\mathscr{T}}} e^n_v$ or $e^n_v \prec_{\bar{\mathscr{T}}} e^l_a$.

If $e^l_a \prec_{\bar{\mathscr{T}}} e^n_v$, then since e^l_a sets x to v' s.t. $v' \neq v''$ for any v'' s.t. $\exists \circ s_l(x)v'' \in \bar{\mathscr{T}}|T_l$ prior to the occurrence of e^n_v. Thus when T_n subsequently executes `:checkpoint` it issues $e^n_v = g_n(x)v'$, and since $v' \neq v''$, this contradicts that $T_n \ddot{\sim} T_k$.

If $e^n_v \prec_{\bar{\mathscr{T}}} e^l_a$, then since $T_l \dot{\prec}^x_{\bar{\mathscr{T}}} T_n$m then from version order $\mathtt{pv}_l(x) < \mathtt{pv}_n(x)$, so from Lemma 44, $\mathtt{rv}_l(x) < \mathtt{rv}_n(x)$. In order for T_n to issue e_a, it must execute `abort` and satisfy the condition line 105. This means that line 128 is executed, so $\mathtt{cv}(x) = \mathtt{rv}_l(x)$.

If subsequently T_l issues e^n_u then it either executes `:write_buffer` or `commit`. Issuing an update event at line 65 or line 85 is dominated by checking whether $\mathtt{cv}(x) = \mathtt{rv}_n(x)$ (for all variables) at line 63 or line 82, respectively. If the condition is failed, the transaction aborts instead. From Lemma 44, $\mathtt{rv}_l(x) < \mathtt{rv}_n(x)$, so if $\mathtt{cv}(x) = \mathtt{rv}_l(x)$, then $\mathtt{cv}(x) \neq \mathtt{rv}_n(x)$. Hence, T_i will `abort` rather than issue an update event. Since during `abort` only a recovery event may be issued, and only if $\mathtt{rv}_n(x) = \mathtt{cv}(x)$, then, similarly, no recovery event is issued. Hence T_n cannot issue events on x following e^l_a.

Since each occurrence of a routine update event or a view event checks $\forall y, \mathtt{cv}(y) = \mathtt{rv}_n(y)$, then no other such event in T_n can follow e^l_a. This is a contradiction.

The execution of a recovery event on x by T_l is dominated by line 102, which cannot be passed until $\mathtt{pv}_l(x) - 1 = \mathtt{ltv}(x)$. Any transaction T_n sets $\mathtt{ltv}(x)$ to $\mathtt{pv}_o(x)$ as a last action during `commit` (line 96) or `abort` (line 111). Hence T_l cannot proceed to abort until T_o finishes committing or aborting. Since T_o cannot execute line 96

or line 111 if line 88 or line 102 was passed, then T_o cannot proceed to commit or abort until some other T_m s.t $\mathrm{pv}_o(x) - 1 = \mathrm{pv}_m(x)$ committed or aborted. Hence T_l cannot execute a recovery event until any T_m s.t. $\mathrm{pv}_m(x) < \mathrm{pv}_l(x)$ committed or aborted.

If $\mathrm{pv}_k(x) < \mathrm{pv}_n(x) < \mathrm{pv}_l(x)$, then $\mathrm{pv}_n(x) < \mathrm{pv}_l(x)$, so T_l executes `abort` only after T_k returns from `abort` or `commit`. Hence, since $e_u^n \prec_{\bar{\mathscr{T}}} res_n\big[A_n\big]$ or $e_u^n \prec_{\bar{\mathscr{T}}} res_n\big[C_n\big]$, either $res_n\big[A_n\big] \in \bar{\mathscr{T}}|T_n$ or $res_n\big[C_n\big] \in \bar{\mathscr{T}}|T_n$, and since $res_n\big[C_n\big] \prec_{\bar{\mathscr{T}}} e_a^l$ or $res_n\big[A_n\big] \prec_{\bar{\mathscr{T}}} e_a^l$, then $e_u^n \prec_{\bar{\mathscr{T}}} e_a^l$. This contradicts that $e_a^l \prec_{\bar{\mathscr{T}}} e_u^n$.

Thus, regardless of whether there is a contradiction. Therefore, T_l cannot issue such e_a^l between e_u^k and another event in $\bar{\mathscr{T}}|T_n$. By extension, it cannot issue e_a^l between e_u^k and another event in $\bar{\mathscr{T}}|T_m$ for any $T_m \in \xi(\bar{\mathscr{T}}, T_i, T_j)$.

□

Lemma 68 (Chain Self-containment)**.** *Given $\mathscr{T}$, and any transactions $T_i, T_j \in \mathscr{T}$, s.t. there exists $\xi(\mathscr{T}, T_i, T_j)$, $\xi(\mathscr{T}, T_i, T_j)$ is self-contained.*

Proof. Given transaction T_q s.t. $\exists e_v^q = g_q(x)v^q \in \bar{\mathscr{T}}|T_q$, assuming that there are any update events on x in $\bar{\mathscr{T}}$ prior to e_v^q, then $\exists e_u^r = \circ s_r(x)v^q \in \bar{\mathscr{T}}|T_r$ where $e_u^r \lessdot e_v^q$ or $\exists e_a^r = \boxplus s_r(x)v^q \in \bar{\mathscr{T}}|T_r$ where $e_a^r \lessdot e_v^q$ for some T_r. In addition, from Lemma 42, $\exists e_v^r = g_r(x)v^r \in \bar{\mathscr{T}}|T_r$ s.t. $e_v^r \prec_{\bar{\mathscr{T}}} e_u^r$ and $e_v^r \prec_{\bar{\mathscr{T}}} e_a^r$ (as applicable).

Then, similarly, assuming that there are any update events on x in $\bar{\mathscr{T}}$ prior to e_v^r, then $\exists e_u^s = \circ s_s(x)v^r \in \bar{\mathscr{T}}|T_s$ where $e_u^s \lessdot e_v^r$ or $\exists e_a^s = \boxplus s_s(x)v^r \in \bar{\mathscr{T}}|T_s$ where $e_a^s \lessdot e_v^r$. And by analogy to T_r, from Lemma 42, $\exists e_v^r = g_r(x)v^r$ s.t. $e_v^r \prec_{\bar{\mathscr{T}}} e_u^r$ and $e_v^r \prec_{\bar{\mathscr{T}}} e_a^r$ (as applicable).

It is then clear that as long as there are update events on x preceding a view event in some transaction, another transaction exists that both views and updates x before that view event.

Thus, given T_k and $T_l \in \xi(\mathscr{T}, T_i, T_j)$ such that $k \neq l$ and $\exists e_u^k = \circ s_k(x)v \in \mathscr{T}|T$ $\exists e_v^l = g_l(x)v' \in \mathscr{T}|T$ and $e_u^k \prec_{\mathscr{T}} e_v^l$, there is a sequence of transactions $\mathscr{S}$ s.t.:

1. the first transaction is T_k,
2. the last transaction is T_l, and
3. given some transaction $T_m \in \mathscr{S}$, where $m \neq k$, T_m is preceded in $\mathscr{S}$ by some transaction T_n, s.t. for $e_v^m = g_m(x)v^m \in \bar{\mathscr{T}}|T_m$, $\exists e_u^n = \circ s_n(x)v^m \in \bar{\mathscr{T}}|T_n$ where $e_u^n \lessdot e_v^m$ or $\exists e_a^n = \boxplus s_n(x)v^m \in \bar{\mathscr{T}}|T_n$ where $e_a^n \lessdot e_v^m$.

Given such $\mathscr{S}$, given some T_m, $m \neq k$, there is some T_n that precedes T_m in $\mathscr{S}$.

If for $e_v^m = g_m(x)v^m \in \bar{\mathscr{T}}|T_m$, $\exists e_u^n = \circ s_n(x)v^m \in \bar{\mathscr{T}}|T_n$ where $e_u^n \lessdot e_v^m$, then $T_m \curvearrowleft T_n$.

If, on the other hand, for $e_v^m = g_m(x)v^m \in \bar{\mathscr{T}}|T_m$, $\exists e_a^n = \boxplus s_n(x)v^m \in \bar{\mathscr{T}}|T_n$ where $e_a^n \lessdot e_v^m$, then from chain isolation there cannot be a recovery event $e_a^k = \boxplus s_k(x)\square$ s.t. $e_a^k \prec_{\bar{\mathscr{T}}} e_v^l$, so it follows that $k \neq n$. Since e_a^n is conservative, $\exists e_v^n = g_n(x)v^m \in \bar{\mathscr{T}}|T_n$.

Since $k \neq n$ and $T_n \in \mathscr{S}$, then there is some T_o preceding $T_n$in $\mathscr{S}$. Then:

a) If $o = k$, then $T_m \curvearrowleft T_k$.

b) If $o \neq k$ and $\exists e_u^n = \circ s_o(x)v^m \in \bar{\mathscr{T}}|T_o$ where $e_u^o \lessdot e_v^n$ then $T_m \curvearrowleft T_o$.

c) If $o \neq k$ and $\exists e_a^n = \boxplus s_o(x)v^m \in \bar{\mathscr{T}}|T_o$ where $e_a^o \lessdot e_v^n$ then by analogy, either case a), b) or c) applies to T_o as it does to T_n. So, by analogy, either a) $T_m \curvearrowleft T_k$, b) $T_m \curvearrowleft T_o$, or c) there is another preceding transaction in $\mathscr{S}$, etc.

Note, however, that since $\mathscr{S}$ is finite, and e_a^k cannot precede e_v^l in $\bar{\mathscr{T}}$, then eventually for some such preceding $T_q \in \mathscr{S}$ case a) or b) and not c) will apply. Thus, there will be some $T_q \in \mathscr{S}$ s.t. $T_m \curvearrowleft T_q$ (where either $q = k$ or $q \neq k$).

Therefore, $\forall T_m \in \mathscr{S}$, s.t. $m \neq k$, $\exists T_n \in \mathscr{S}$ s.t. $T_m \tilde{\sim} T_n$.

In addition, for each such pair T_m, T_n , there is therefore $\xi(\bar{\mathscr{T}}, T_n, T_m)$. Furthermore, if $T_m \tilde{\sim} T_n$ and for some other T_o, $T_n \tilde{\sim} T_o$, then there is $\xi(\bar{\mathscr{T}}, T_o, T_m)$. Thus, there is also $\xi(\bar{\mathscr{T}}, T_k, T_l)$, such that if $T_l \tilde{\sim} T_m$ and $T_m \in \mathscr{S}$, then $T_m \in \xi(\bar{\mathscr{T}}, T_k, T_l)$. Since $T_k, T_l \in \xi(\bar{\mathscr{T}}, T_i, T_j)$, then if $T_m \in \xi(\bar{\mathscr{T}}, T_k, T_l)$, $T_m \in \xi(\bar{\mathscr{T}}, T_i, T_j)$.

If $\mathscr{S} = T_k \cdot T_l$ then trivially $v = v'$.

Otherwise, since $T_l \in \mathscr{S}$ and $l \neq k$, then $\exists T_m \in \mathscr{S}$ s.t. $T_l \tilde{\sim} T_m$, so $\exists e_u^m = \circ s_m(x) v' \in \mathscr{T}|T_m$ for some $T_m \in \xi(\mathscr{T}, T_i, T_j)$, s.t. T_m precedes T_l and follows T_k in $\xi(\mathscr{T}, T_i, T_j)$ and $e_u^k \prec_{\mathscr{T}} e_u^m \prec_{\mathscr{T}} e_v^l$. □

Lemma 69 (Tree Chain Consistency)**.** *Trace $\bar{\mathscr{T}}$ is chain consistent.*

Proof. From Lemma 67 and Lemma 68. □

Lemma 70 (Trace Harmony)**.** *Trace $\bar{\mathscr{T}}$ is harmonious.*

Proof. Trace $\bar{\mathscr{T}}$ satisfies all of the following: a) minimalism from Lemma 40 b) consonance from Lemma 57, c) obbligato from Lemma 61, d) coherence, commit accord, abort accord, and abort coda from Lemmas 66, 64, 63, and 65, e) isolation from Lemma 45, f) decisiveness from Lemma 62, g) chain consistency from Lemma 69, and h) unique writes (assumed). □

Corollary 22. *History $H = Hist(\bar{\mathscr{T}})$ is last-use opaque. In consequence OptSVA+R is last-use opaque.*

7.4 Last-use Opacity of OptSVA-CF+R

Given that OptSVA-CF+R is a more restricted variant of OptSVA+R, it follows that it guarantees safety properties that are at least as strong as OptSVA+R's.

Theorem 10. *Every OptSVA-CF+R history is last-use opaque.*

Proof sketch. For the purpose of the proof assume that $Obj = Var$, Then the requirements in synchronizing complex objects in OptSVA-CF+R transactions mean that OptSVA-CF+R allows a subset of histories allowed by OptSVA+R. Since any history allowed by OptSVA+R is last-use opaque, then any history allowed by OpSVA-CF+R is also last-use opaque. □

8 Implementation and Evaluation

This chapter discusses two distributed TM implementations of versioning algorithms. We explain the architecture, API, and specific mechanisms these systems employ to comply with particular versioning algorithms. Each implementation is also comprehensively evaluated against other distributed concurrency control mechanisms.

In the first section we present Atomic RMI, an implementation of SVA+R/RSVA+R that aims to introduce an easy-to-use interface that allows the execution of atomic transactions on top of Java RMI. The idea of Atomic RMI builds on previous work on the calculus of distributed atomic tasks [96, 98] and was first presented in its current form in [75, 78].

In the second section we discuss Atomic RMI 2, an implementation of OptSVA-CF+R/ROptSVA-CF+R that builds on Atomic RMI but exhibits much improved performance thanks to the high degree of parallelism of the underlying concurrency control algorithm. This implementation was first presented in [82].

8.1 Atomic RMI

In this section, we present *Atomic RMI*, a programming framework that extends Java RMI with support for distributed transactions in the control flow model. Atomic RMI is a fully-pessimistic CF distributed TM with support for programmatic abort (rollback) whose goal is to provide a simple-to-use and powerful interface for executing atomic code in distributed systems. It implements SVA+R as the underlying concurrency control algorithm. We first present an overview of our system architecture, followed by an in-depth look into specific features of the system, including a discussion of its strengths and limitations of the implementation. We then evaluate the system and show its efficiency compared to 2PL-based locking schemes and an optimistic distributed TM.

8.1.1 Overview

The Atomic RMI architecture builds on the architecture of Java RMI, as shown in Fig. 8.1. *Java Virtual Machines (JVMs)* running on network nodes can host a number of discrete shared remote objects, each of which is uniquely identifiable within the system and registered in an RMI registry located on the same node. Each remote object

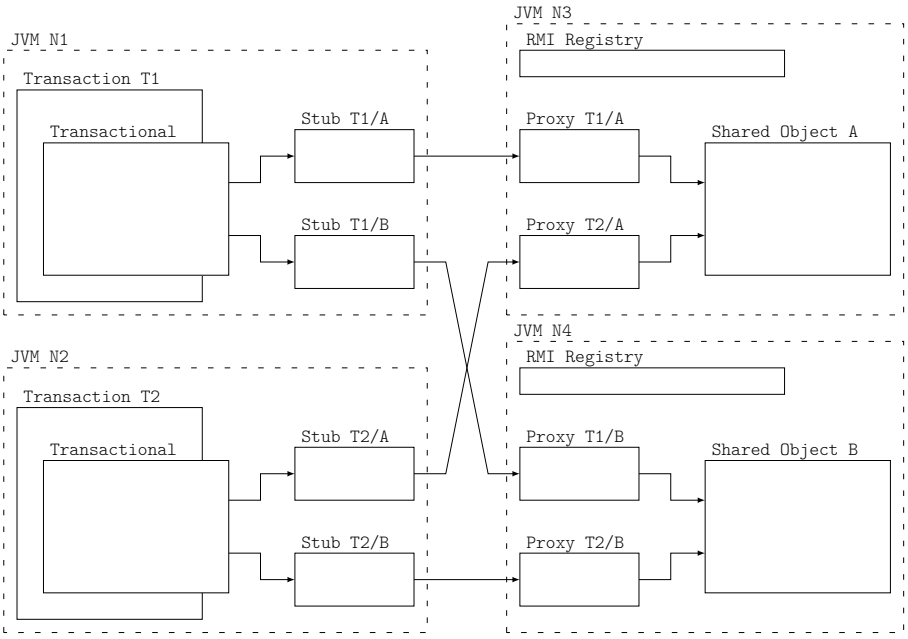


Figure 8.1: Atomic RMI architecture.

specifies an interface of methods that can be called remotely, hence adhering to the heterogeneous object model. A client application running on any JVM can ask any registry for a reference to a specific object. Then, the client can use the reference to call the object's methods. Each shared object is located at exactly one specific node (as opposed to the object being copied or moved to other nodes, or being replicated on several nodes) and all operations invoked on that object cause the appropriate method's code to executes on the object's host node and return the result to the client in accordance with the control flow model. The semantics of the methods are defined by the programmer and can be anything from simple gets and sets, to complex methods executing arbitrary server-side code, accessing a database, or even invoking other remote objects.

Atomic RMI introduces transaction-based concurrency control to this model. Clients calling multiple remote objects in parallel can resort to atomic transactions to enforce consistent accesses to fields of the objects, and the system makes sure that concurrent transactions are executed correctly and efficiently. For this, Atomic RMI employs SVA+R, a TM concurrency control algorithm described in Section 6.2.2. SVA+R is our main focus in the following discussion, but Atomic RMI can also switch to RSVA+R, a variant of SVA+R.

In order to give SVA+R the means to guide execution, so that correctness is guaranteed, Atomic RMI introduces remote object proxies into the RMI architecture. For each shared remote object there is an automatically-generated proxy on the host node that has a wrapper method for each of the original object's methods (those available remotely). Clients are required to access remote objects via proxies, so all calls of the original object's methods first pass through wrapper methods. The wrapper methods are then used to enforce SVA+R: establish whether a given operation can be executed at a given time, or whether it must be deferred or canceled. Once the algorithm establishes that a call may proceed, the proxy calls the original method of the remote object. In theory, proxy objects could be located either on the server side or the client side, but since the communication between the proxy and the shared object is much more frequent than that between the transaction and the proxy, placing them on the server-side incurs lower

```java
interface Transaction {
  Transaction();
  Transaction(boolean reluctant);

  <T> T accesses(T obj);
  <T> T accesses(T obj, int supr);

  void start();
  void commit();
  void retry();
  void abort();

  void start(Transactional runnable);
}

interface Transactional {
  void run(Transaction t);
}
```

(a) Atomic RMI transaction interface.

```java
interface Account extends Remote {
  int balance();
  void deposit(int amount);
  void withdraw(int amount);
  void reset();
}

class AccountImpl
    extends TransactionalUnicastRemoteObject
    implements Account {

  private int balance = 0;

  public int balance() {
    return balance;
  }
  public void deposit(int amount) {
    balance += amount;
  }
  public void withdraw(int amount) {
    balance -= amount;
  }
  public void reset() {
    balance = 0;
  }
}
```

(b) Shared object definition.

```java
Registry r = LocateRegistry.getRegistry(...);

Transaction t = new Transaction();
Account a = t.accesses(r.lookup("A"), 2);
Account b = t.accesses(r.lookup("B"), 1);

t.start(new Transactional{
  void run(Transaction t) {
    a.withdraw(100);
    b.deposit(100);

    if (a.balance() < 0)
      t.abort();
  }
});
```

(c) Transaction definition (`Transactional` object).

```java
Registry r = LocateRegistry.getRegistry(...);

Transaction t = new Transaction();
Account a = t.accesses(r.lookup("A"), 2);
Account b = t.accesses(r.lookup("B"), 1);

t.start();

a.withdraw(100);
b.deposit(100);

if (a.balance() < 0)
  t.abort();
else
  t.commit();
```

(d) Transaction definition (in-line).

Figure 8.2: Atomic RMI API and examples.

overheads. In addition, if the proxy is placed on the server, it can easily manage copy and log buffers, which must be placed on the server to preserve the CF model—methods executed using buffers should have side effects on the same node as the original object.

Shared Remote Objects

Shared remote objects used with Atomic RMI are plain unicast (stateful) RMI objects, except that instead of `UnicastRemoteObject` (which handles the Java Remote Method Protocol) they subclass the `TransactionalUnicastRemoteObject` class. We show an example of a bank account object defined in this way in Fig. 8.2b. This class creates proxy objects when necessary, in effect injecting SVA+R support code into remote method invocations. The methods of remote objects are not limited: as well as simple operations like reading and writing to a field, they can contain blocks of code which include side effects, system calls, I/O operations, network communication, etc. that execute on the server. This freedom is possible in large part due to the pessimistic approach to concurrency control used by SVA+R—since these operations often produce visible effects on the system, they cannot be repeated in case of conflicts, as in the optimistic approach. The pessimistic approach will only let them execute (up to) once in the course of nor-

mal operation, although allowances must be made when the user triggers an abort by manually rolling back some transaction.

In particular, remote methods can also contain method calls to other remote objects, further distributing the execution of the transaction. Note, however, that if these are to be accessed transactionally (i.e., with the same correctness guarantees), references to the objects have to be made known to the transaction *a priori* for SVA+R to operate correctly.

Note that Atomic RMI uses the control flow model of execution, since it allows transactions to execute code on the server where the remote object is located, rather than limiting them to executing code always on their own node, just using remote data, as in the data flow model. Our intention is to orientate Atomic RMI towards this model, since it provides greater freedom and expressiveness to the programmer, who can balance the load between servers and clients by defining the level of processing that is done on remote objects. Additionally, the control flow model is more versatile, because it can emulate the data flow model if remote objects only provide methods which simply write or retrieve data from the host.

Transaction Objects

The programmer declares a transaction using the API provided by Atomic RMI (Fig. 8.2a), by creating a `Transaction` object, which is responsible for starting and stopping transactional execution. We show example transaction definitions in Fig. 8.2c and 8.2d. A transaction can be defined as reluctant at this point, meaning it will never be forced to abort, because it will not access objects that are released early, instead waiting for the preceding transaction to commit or abort. Specifically, if transactions are defined as reluctant, Atomic RMI switches to RSVA+R to execute them.

Once a transaction object is created, it is used to declare the transaction's *preamble*, where the programmer specifies which objects will be used by the transaction and how, by passing the reference retrieved from the RMI registry to method `accesses`. The programmer can use a variant of this method to also provide suprema for any object used by the transaction. The suprema indicate the maximum number of times the transaction will execute methods on each shared object throughout the execution of the code. In the examples in Fig. 8.2c and 8.2d, the preamble declares the transaction will invoke a method on object `A` at most twice (line 4), and at most once on `B` (line 5).

In practice, suprema do not have to be inferred manually, but instead static analysis (see Chapter 9) or the type system [96] can be used to do it automatically. If suprema are not given, infinity is assumed (and the system maintains correctness guarantees). If suprema are provided though, the underlying concurrency control algorithm uses them to effect early release, and in this way increase the level of parallelism between concurrent transactions, which has a positive effect on performance.

Instrumentation

When accesses are declared within the preamble, an object stub is created. This stub is then used within the code of the transaction to invoke methods on a shared object, as with ordinary RMI stubs. The difference between an ordinary RMI stub and an Atomic RMI stub is that the latter does not forward method calls to the shared object directly, but instead uses a proxy object. Proxy objects are created dynamically on the node hosting the shared object in question at the same time as the stub is created by the transaction. Each proxy object links one specific shared object on the server side with one specific transaction (object) on the client side. Proxy objects implement the interfaces of the shared objects they are linking, and their rôle is to inject concurrency control code before

and after the invocation of specific methods of the shared object. The injection is done via reflection, which supplies the necessary flexibility, allowing arbitrary methods easily to be supplemented with concurrency control code. Proxy objects can be decommissioned once a transaction that created them finishes executing.

Transactional Code

Once the preamble of the transaction is in place, the transaction's code can be specified. This is done in one of two ways: by implementing the `Transactional` interface, or in-line. An example of the former is given in Fig. 8.2c. Here, transactional code is specified by creating an object implementing the `Transactional` interface (line 7), whose `run` method then defines the logic of the transaction (lines 9–15). In general, a transaction can contain virtually any code between its start and either commit or abort. This specifically means that apart from operations on shared objects, any local operations, e.g., irrevocable operations, can be present within.

As an example of a simple transaction, that transfers an amount of 100 currency from one bank account to another. Thus, an anonymous `Transactional` object is created (line 7) and within the `run` method, the `withdraw` method is called on object `a` (the stub for shared object `A`), after which the `deposit` method is called on `b` (the stub for `B`). The programmer can rest assured the concurrency control algorithm will synchronize the execution of this code so that no other transaction in the system interferes with the execution in a way that would violate its consistency. If the transaction reaches the end of its code it attempts to commit. The programmer is also given the option to abort or retry the entire transaction manually by using the transaction object and invoking either the `abort` or `retry` method. Here, the transaction is rolled back (line 13) if it turns out that the balance on account `A` fell below 0 as a result of executing `withdraw` (line 12).

We define the same transaction in-line in Fig. 8.2d. Here, instead of defining an object to run transactional code, the transaction is delimited by the invocation of methods `start`, `commit` and `abort`. Thus the transaction begins at line 7 and finishes at line 13 or 15, depending on whether the condition at line 12 evaluates to true or false. Note that in-line transactions must explicitly call `commit`. In addition, `Transactional` objects handle the retry operation automatically, by aborting and restarting the transaction, while in-line transactions require that this be handled programmatically (an exception must be caught). Thus, `Transactional` objects are easier to use, while in-line transactions are more flexible. Otherwise, both methods of defining transactions are equivalent.

Suprema

The main requirement that Atomic RMI poses for its users is the need to provide the set of objects used by transactions *a priori* and a strong suggestion to also provide upper bounds (suprema) on the number of accesses of remote objects accessed by each transaction. The former is required to acquire versions on each object. The latter allows SVA+R to decide when objects can be released early—if this information is inexact or omitted (equivalent to setting the upper bound to infinity) SVA+R will only release objects when transactions commit or abort. In such cases, although the execution will nevertheless be correct, Atomic RMI will be less efficient since transactions will wait more on one another.

However, while it is acceptable for upper bounds to be too high, it is essential that they are never lower than the actual number of calls a transaction does to a given object. If the specification is lower then the actual number of accesses, the guarantees provided by SVA+R cannot be upheld, because a transaction could release an object and then attempt to access it again. In order to alleviate such situations, transactions throw an

```
1 t = new Transaction();
2 a = t.accesses(a);
3 b = t.accesses(b);
4 t.start();
5 for (i = 0; i < n; i++) {
6   a.foo();
7   b.foo();
8 }
9 t.release(a);
10 t.release(b);
11 // local operations
12 t.commit();
```

(a) Early release at end of block.

```
1 t = new Transaction();
2 a = t.accesses(a);
3 b = t.accesses(b);
4 t.start();
5 for (i = 0; i < n; i++) {
6   a.foo();
7   if (i == n)
8     t.release(a);
9   b.foo()
10 }
11 t.release(b);
12 // local operations
13 t.commit();
```

(b) Conditional early release.

```
1 t = new Transaction();
2 a = t.accesses(a, n);
3 b = t.accesses(b, n);
4 t.start();
5 for (i = 0; i < n; i++) {
6   a.foo(); // nth call: release
7   b.foo(); // nth call: release
8 }
9 // local operations
10 t.commit();
```

(c) Early release by supremum.

```
1 t = new Transaction();
2 for (h : hotels)
3   trHotels.add(t.accesses(h, 2));
4 t.start();
5 for (h : trHotels) {
6   if (h.hasVacancies())
7     h.bookRoom();
8   else
9     t.release(h);
10 }
11 t.commit();
```

(d) Complementary manual release.

Figure 8.3: Early release examples.

exception when the number of accesses for some object exceeds its supremum. It is then left to the programmer to resolve the issue by handling the exception. A typical solution would be to roll back the offending transaction. A more sophisticated technique would be to roll back the transaction, then modify the suprema and retry.

The upper bounds can be collected manually by the programmer by inspecting the code and creating the preamble. They can also be inferred automatically by various means, including a type system (see e.g., [96]) or static analysis. In particular, Atomic RMI comes with a precompiler tool which statically analyzes transactions to discover which objects they use, and to derive the upper bounds on accesses to them. With this information, the precompiler generates the appropriate code and inserts it into the program. The idea behind the static analysis is described in detail in Chapter 9. The tool itself is a command-line utility implemented on top of the Soot framework [87].

Manual Early Release

The early release mechanism in Atomic RMI can be triggered automatically (via the supremum early release mechanism) or manually.

Manual early release is an extension of SVA+R that we introduce in Atomic RMI, where, if the programmer has good knowledge of when an object stops being used in a transaction from the semantics of the program, she can allow that remote object to be released by invoking the release operation. The release operation simply waits for the access condition on a given remote object to become true, and then releases it as if its upper bound was reached. This mechanism must be used carefully, so that a released object is not accessed again later on (causing an exception). On the other hand, the mechanism can be used to complement the early release mechanism supplied by SVA+R, as we explain below.

Note the simple example in Fig. 8.3a, where a transaction calls methods on shared objects `a` and `b` in a loop. If manual release was to be used, the simplest way to use it is to insert release instructions at the end of the loop at lines 9–10. However, it will mean

that before a is released, the transaction unnecessarily waits until b executes as well. If a and b are remote objects, each such call can take a long time, so this simple technique impairs efficiency.

Instead, the programmer should strive to write transactions like in Fig. 8.3b. Here, a is released at lines 7–8, in the last iteration of the loop before the method call on b is started. An earlier release improves parallelism, but the solution requires that the programmer spend time on optimizing concurrency (which the TM approach should avoid) and to clutter up the code with instructions irrelevant to the application logic. In addition, the release in both examples sends an additional network message to a and b (because the release method requires it), which can be relatively expensive.

However, if the SVA+R algorithm is given the maximum number of times each object is accessed by the transaction, i.e., that a and b will be accessed at most n times each, then Atomic RMI can determine which access is the last one as it is happening. Then, the transaction's code looks like in Fig. 8.3c, where suprema are specified in lines 2–3, but the instructions to release objects are hidden from the programmer, so there is no need for supplementary code. Additionally, since release is done as part of the nth call on each object, there is no additional network traffic. Furthermore, object a does not wait for the method `b.foo()` to execute.

However, releasing by suprema alone is not always the best solution, since there are scenarios when deriving precise suprema is impossible. In those cases the manual early release complements the suprema-based mechanism in increasing the parallelism of transactional executions. One such case is shown in Fig. 8.3d, where a transaction searches through objects representing hotels, and books a room if there are vacancies. Each interaction with a hotel can take up to two method calls: vacancy check (line 6) and booking (line 7). However the supremum will only be precise for one hotel, the first one with vacancies. Other hotels that do not have vacancies, will not be asked to book a room, so there is only one access. This means that the supremum will not be met for those cases until the end of the transaction, so those hotel objects will only be released on commit. This is particularly paradoxical, since the transaction will retain exclusive access longer to objects it does not intend to use. Hence, they are manually released on line 9, so the objects are not needlessly retained and can be accessed by other transactions as soon as possible.

Irrevocable Operations

The greatest advantage of Atomic RMI is its pessimistic algorithm, which allows any operation to be used within transactions. In particular, irrevocable operations pose no problem. These are operations that have visible effects on the system and cannot be easily reverted, e.g., system calls, sending network messages. This is not true for optimistic transactions, because conflicts cause aborts, which then cause irrevocable transactions to be repeated.

For the same reason, Atomic RMI allows transactions to include locking or to start new threads within transactions. This is also often not possible in optimistic transactions, where aborts can cause threads to be restarted or locks to be acquired and not released (especially, if conflicts are detected eagerly). However, not only does allowing these sorts of operations improve expressiveness, but it also makes working with legacy code easier.

While it is true that Atomic RMI transactions operating according to SVA+R do not typically forcibly abort (as we remark in Section 8.1.2, even if programmatic aborts occur, cascading aborts do not happen in benchmarks), it is possible for transactions to abort *in potentia*. In order to make forced aborts impossible for specific transactions, Atomic RMI provides an interface to make transactions reluctant as per RSVA+R, meaning that

they are never forced abort by the TM, and therefore are completely safe for irrevocable operations.

Buffering

In order to buffer remote objects, Atomic RMI introduces a universal mechanism that deep-copies shared remote objects. The copy is created and kept on the same JVM as the original object, to avoid the cost of moving it across the network. The copy is created by reflection: a new object of the same type is created, and all of its fields (barring concurrency-control–related data) are set to the same values as the original object. This method of copying is used to implement both the `st` and the `buf` buffers used by SVA+R.

Nesting and Recurrency

Atomic RMI supports transaction nesting, albeit with limitations. The programmer can create a transaction within another transaction, but in such cases it is vital to ensure that they do not share objects. Otherwise, the inner transaction will wait for the outer to release the objects, while the outer will not release them until the inner finishes. In effect, a deadlock occurs.

Atomic RMI also supports transaction recursion. That is, a transaction may call itself within itself (for instance, if a transaction is defined within a method). The recursion will be treated as a single transaction. The execution will proceed until the `commit` and `abort` methods are called, in which case the transactional method is exited and the transaction finishes as normal. Keep in mind, however, that the suprema for object accesses must still be defined for the entire execution of the transaction.

Fault Tolerance Mechanisms

In distributed environments partial failures are a fact of life, so any DTM system must have mechanisms to deal with them. Atomic RMI handles two basic types of failures: remote object failures and transaction failures.

Remote object failures are straightforward and the responsibility for detecting them and alarming Atomic RMI falls onto the mechanisms built into Java RMI. Whenever a remote object is called from a transaction and it cannot be reached, it is assumed that this object has suffered a failure and an exception is thrown. The programmer may then choose to handle the exception by, for example, rerunning the transaction, or compensating for the failure. Remote object failures follow a *crash-stop* model of failure: any object that crashed is removed from the system.

On the other hand, a client performing some transaction can crash causing a transaction failure. Such failures can occur before a transaction releases all its objects and thus make them inaccessible to all other transactions. The objects can also end up in an inconsistent state. For these reasons transaction failures need also to be detected and mitigated. Atomic RMI does this by having remote objects check whether a transaction is responding. If a transaction fails to respond to a particular remote object (i.e., if it times out), it is considered to have crashed, and the object performs a rollback on itself: it reverts its state and releases itself. If the transaction actually crashed, all of its objects will eventually do this and the state will become consistent. On the other hand, if the crash was illusory and the transaction tries to resume operation after some of its objects rolled themselves back, the transaction will be forced to abort when it communicates with one of these objects.

8.1.2 Evaluation

In this section we present the results of a practical evaluation of Atomic RMI. First, we compare the performance of Atomic RMI to other distributed concurrency control mechanisms, including another distributed TM. In the second test, we check the performance of Atomic RMI under different *Java Runtime Environments* (*JREs*).

Benchmarks

For our comprehensive evaluation we used a micro-benchmark and three complex benchmarks. We based our implementation of the benchmarks on the one included in HyFlow [68].

The *distributed hash table benchmark* (DHT) is a micro-benchmark containing a number of server nodes acting as a distributed key-value store. Each node is responsible for storing values for a slice of the key range. There are two types of transactions. A write transaction selects 2 nodes and atomically performs a write on each. A read transaction selects 4 nodes and performs an atomic read on them. The benchmark is characterized by small transactions (2–4 remote operations, few local operations) and low contention (few transactions try to access the same resource simultaneously).

The *bank benchmark* simulates a straightforward distributed application using the bank metaphor. Each node hosts a number of bank accounts that can be accessed remotely by clients. Bank accounts allow write operations (*withdraw* and *deposit*) and a read operation (*get balance*). Clients perform either write or read transactions. In the former type, a transfer transaction, two random accounts are selected and some sum is withdrawn from one account and deposited on the other. In the latter type, an audit transaction, all the accounts in the bank are atomically read by the transaction and a total is produced. The benchmark has both short and long transactions and medium to high contention, depending on the number of read-only transactions.

The *loan benchmark* presents a more complex distributed application where the execution of the transaction is also distributed. Each server hosts a number of remote objects that allow write and read operations. Each client transaction atomically executes two reads or two writes on two objects. When a read or write method is invoked on a remote object, then it also executes two reads or writes (respectively) on two other remote objects. This recursion continues until it reaches a depth of five. Thus, each client transaction "propagates" through the network and performs 30 operations on various objects. Hence, the benchmark is characterized by long transactions and high contention, as well as relatively high network congestion.

Finally, the *vacation benchmark* is a complex benchmark (originally a part of STAMP [58]), representing a distributed application with the theme of a travel agency. Each server node supplies three types of objects: cars, rooms, and flights. Each of these represents a pool of resources that can be checked, reserved, or released by a client. When some resource is reserved, associated reservation and customer objects are also created on the server. Clients perform one of three types of transactions. *Update tables* selects a number of random objects and changes their price to a new value. *Delete customer* removes a random customer object along with any associated reservations. Note that the *delete customer* operation requires some transactions to execute speculatively and (programmatically) abort when the list of objects reserved for deletion becomes out of date. *Make reservation* is a read-dominated transaction that searches through a number of objects, chooses one of each type (car, room, flight) that meet some price criterion. Once the objects are chosen, the transaction may create a reservation. The benchmark has medium to large transactions with a lot of variety, and medium to high contention.

Frameworks

We evaluate Atomic RMI with specified precise suprema (where possible). All versions of Atomic RMI use manual early release in the vacation benchmark to improve efficiency while making reservations (while searching through remote objects). In all other cases, transactions release objects when they reach their supremum.

We compare Atomic RMI with standard Java RMI with mutual exclusion locks (or mutexes, denoted *Exclusion Locks*) and Java RMI with read/write locks (denoted *R/W Locks*). They feature fine grained locking: there is one lock per remote object. The locks are used like a CR2PL transaction: all locks are acquired at the start (although we simplify the lock acquisition procedure by using a predefined locking order, which is sufficient to avoid deadlocks), and they are all released on commit. We use this locking scheme with mutual exclusion locks as a baseline algorithm, which is very simple to use and can be expected to be seen in applications written by conventionally trained software engineers. On the other hand, read/write locks present one of the most popular types of performance optimizations in concurrent systems: parallelizing reads.

We also compare Atomic RMI with HyFlow [68], another Java RMI-based implementation of distributed transactional memory, implementing the Distributed Transactional Locking (DTL) algorithm, a variant of TL2, a well-known optimistic TM. Since the technology used in both Atomic RMI and HyFlow is the same, the comparison should show the performance difference between the pessimistic and optimistic approaches to TM.

Testing Environment

In each of the benchmarks every node performs the rôle of a server hosting a number of publicly accessible remote objects, as well as a client running various randomly chosen types of transactions using remote objects from any server.

We perform our tests on a 10-node cluster connected by a private 1 Gb network. Each node is equipped with two quad-core Intel Xeon L3260 processors at 2.83 GHz with 4 GB of RAM each and runs a OpenSUSE 13.1 (kernel 3.11.10, x86_64 architecture). We use the 64-bit IcedTea 2.4.4 OpenJDK 1.7.0_51 Java runtime (suse-24.13.5-x86_64) for tests involving comparison between multiple frameworks. We also use this JRE (denoted OpenJDK 1.7.0_51 on the graphs) alongside Oracle's 64-bit 1.7.0_55-b13 Java Runtime Environment with Java HotSpot build 24.55-b03 (denoted Oracle 1.7.0_55), and Oracle's 64-bit 1.8.0_05-b13 Java Runtime Environment with Java HotSpot build 25.5-b02 (denoted Oracle 1.8.0_05) for evaluating behavior when running on different JREs. We also attempted to run the benchmarks on the last version of Oracle JRockit (1.6.0_45-b06), but were unsuccessful due to compatibility issues with the libraries used for the implementation of the benchmarks.

Each of the benchmarks is run on 2–10 nodes. Every node hosts one server with as many objects as specified by the benchmark. In addition, every node hosts one client with 24 threads each. So, for example, on 10 nodes there are 240 simultaneous transactions accessing objects on 10 nodes. Threads execute transactions selected at random. In one batch of tests there are 20% of read transactions and 80% of write transactions in each benchmark. In the other batch the ratio is reversed.

Results

The results of the comparison between concurrency control mechanisms are presented in Fig. 8.4. Each benchmark is presented on two graphs: one for a 20% read-to-write operation ratio, and the other for an 80% read-to-write operation ratio. Points on the graph represent the mean throughput (on the y-axis) from the given benchmark run on

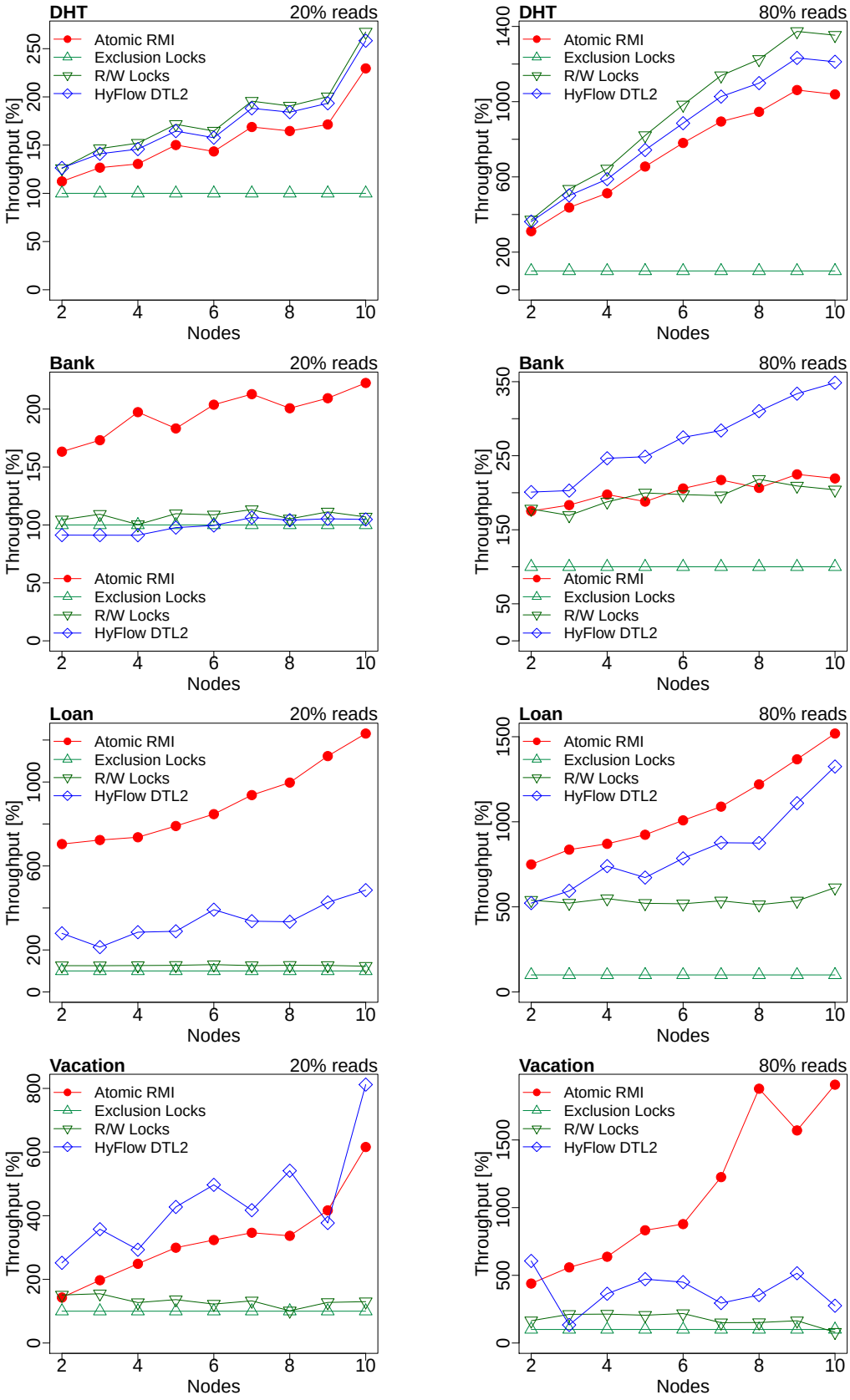


Figure 8.4: Evaluation results.

a particular number of nodes (on the x-axis). The results are shown as a percentage improvement in relation to the execution of Exclusion Locks.

The results of the DHT benchmark show that in a low-contention environment with short transactions Atomic RMI is comparable to performance obtained by using both fine grained R/W Locks and HyFlow, and all three are much better than fine grained Exclusion Locks. Atomic RMI's advantage over Exclusion Locks comes from early release and allowing some transactions to execute in part in parallel. Thus, there are more transactions executing at once, so more of them can go through the system per unit of time. R/W Locks and HyFlow attain a very similar result, by allowing reads to execute in parallel with other reads, therefore also allowing some transactions to execute in part in parallel. The gain from treating reads specially is very similar to what is gained from early release, so the shapes of the graphs are very similar. However, the overhead of maintaining all of the distributed TM mechanisms in Atomic RMI and HyFlow—including rollback support (so making copies of objects) and fault tolerance (extra network communication)—is greater than the overhead of R/W Locks. Hence, Atomic RMI and HyFlow perform consistently worse in DHT than R/W Locks. Note also that the advantage that the more subtle frameworks have over Exclusion Locks increases in proportion to the number of nodes in the network, hinting at better scalability.

The results for Bank show a case with a higher contention, where the higher cost of setting up HyFlow's and Atomic RMI's more complex concurrency control pays off, so both frameworks tend to outdo R/W Locks (and Exclusion Locks) on average. The benchmark also shows the impact of the two approaches to parallelizing transactions. Since R/W Locks and HyFlow allow executing reads in parallel, they both gain a significant boost in the 80% read case, since any number of transactions can simultaneously read from the same object. To the contrary, Atomic RMI parallelizes transactions on the basis of early release rather than reads, so it is forced to wait for a preceding transaction to release the right object for two transactions to be able to execute simultaneously. Since transactions here contain operations in random order, Atomic RMI's SVA+R algorithm is often forced to wait for a preceding transaction to release the right object. This still gives performance similar to that of R/W Locks, but it means Atomic RMI is outperformed by HyFlow. On the other hand, in the 20% read case, R/W Locks and HyFlow have fewer reads to parallelize, so they execute on a par with Exclusion Locks, HyFlow performs particularly poorly in this scenario because of the high number of aborts caused by speculative execution of write operations. Here, HyFlow transactions abort in between 15.5% and 51% cases (as opposed to between 4.25% and 8.9% in the 80% read case), while other frameworks do not perform aborts in this scenario at all. In contrast, Atomic RMI performs significantly better than the other frameworks, since its early release mechanism does not depend on a large read-to-write ratio. In fact, Atomic RMI performs similarly in both the 20% and the 80% read case, reliably achieving a throughput of around 200% in both scenarios. Nevertheless, it is clear that Atomic RMI could benefit from introducing support for read/write differentiation in addition to the existing mechanisms.

The Loan benchmark shows that Atomic RMI is also much better at handling long transactions and high contention than all other types of the concurrency control mechanisms. Again, since Atomic RMI does not distinguish between reads and writes, both scenarios are effectively the same in terms of performance and an increase in throughput comes from releasing objects as early as possible. However, as opposed to Bank, in the Loan benchmark, Atomic RMI can effect an early release while about half of the transaction still remains to be executed. Hence, Atomic RMI transactions run in parallel in part to the transaction preceding it, and in part to the one following it. This creates a significant performance gain compared to R/W Locks and HyFlow. These, again, differ

in performance between the 20% read case and the 80% read case, but in this high a contention their advantage over Exclusion Locks is not as great as Atomic RMI's.

Finally, Vacation shows the behavior of more complex transactions in a high contention environment. In this scenario, there is an actual advantage to being able to roll back and this is a component in the performance gain. Atomic RMI makes copies for abort on the server-side, so there is no network overhead associated with either making a checkpoint or reverting objects to an earlier copy. On the other hand, the rollback mechanism in the locking schemes requires that clients copy objects and store them on the client side, which makes them costly operations. Atomic RMI, therefore, has a big advantage over both locking mechanism, from rollback support alone. In particular, the transaction that requires rollback is *delete customer*, which makes up for 10% and 40% of transactions in Vacation (in the 80% and 20% read case, respectively). Furthermore, the complexity of this real-world–like benchmark makes transactions increasingly difficult to parallelize using locks, since often it is necessary to lock a superset of a transaction's read set and write set, and since read-only transactions are less likely to occur. Hence, R/W Locks struggle with performance, even falling behind Exclusion Locks at times.

The comparison of HyFlow and Atomic RMI in Vacation is much more involved. Since read transactions do not imply read-only transactions here, there is much less to be gained by parallelizing reads. Here, even a read transaction can write, so cause conflicts and therefore effect aborts in HyFlow. It is the order of operations and the implementation of the read transaction that explains why Atomic RMI does better than HyFlow in the 80% read case and not in the 20% read case. First of all, the *make reservation* transaction initially performs a sequence of reads to a large set of remote objects, until one is found that fits some specified criterion. Since the object can be released instantly if the criterion is not met, then Atomic RMI allows many parallel transactions to work on the same objects. And since reads in Vacation are done in the same order in each transactions, any two Atomic RMI transactions can execute almost entirely in parallel. On the other hand, since there are writes at the end of *make reservation*, if HyFlow executes these in parallel, a conflict can occur and cause an abort. This is why there is the advantage of Atomic RMI over HyFlow in an 80% read scenario, where the *make reservation* transaction is prevalent. On the other hand, the necessary rollback in *delete customer* is more problematic in Atomic RMI, since it may cause a cascade of aborts. Furthermore, *update tables* performs reads in random order, so Atomic RMI encounters the same problems as in the 80% read case in Bank. HyFlow avoids both these problems through speculation and avoiding cascading aborts. Hence HyFlow outperforms Atomic RMI in the 20% read case where these particular transactions are a bigger part of the workload. However, both Atomic RMI and HyFlow perform quite well, achieving a typical speedup of at least 200% in comparison to Exclusion Locks.

On the whole, Atomic RMI is able to perform just as well as fine grained locks in all environments, with only small penalty for additional features in environments particularly hostile to versioning algorithms (low contention, short transactions). On the other hand, in environments for which versioning algorithms were intended (high contention, long transactions, mixed reads and writes) Atomic RMI gains a significant performance advantage over fine grained locking. In comparison to HyFlow, both TM systems perform variously in different environments. On average, Atomic RMI tends to perform better than HyFlow in high contention, while it tends to be outperformed by HyFlow in cases where read-only transactions can be treated specially. Hence Atomic RMI is preferable to R/W Locks and Exclusion Locks in all cases, while the decision to use Atomic RMI in place of an optimistic TM like HyFlow should be made depending on the workload.

The results under different Java Runtime Environments are presented in Fig. 8.5.

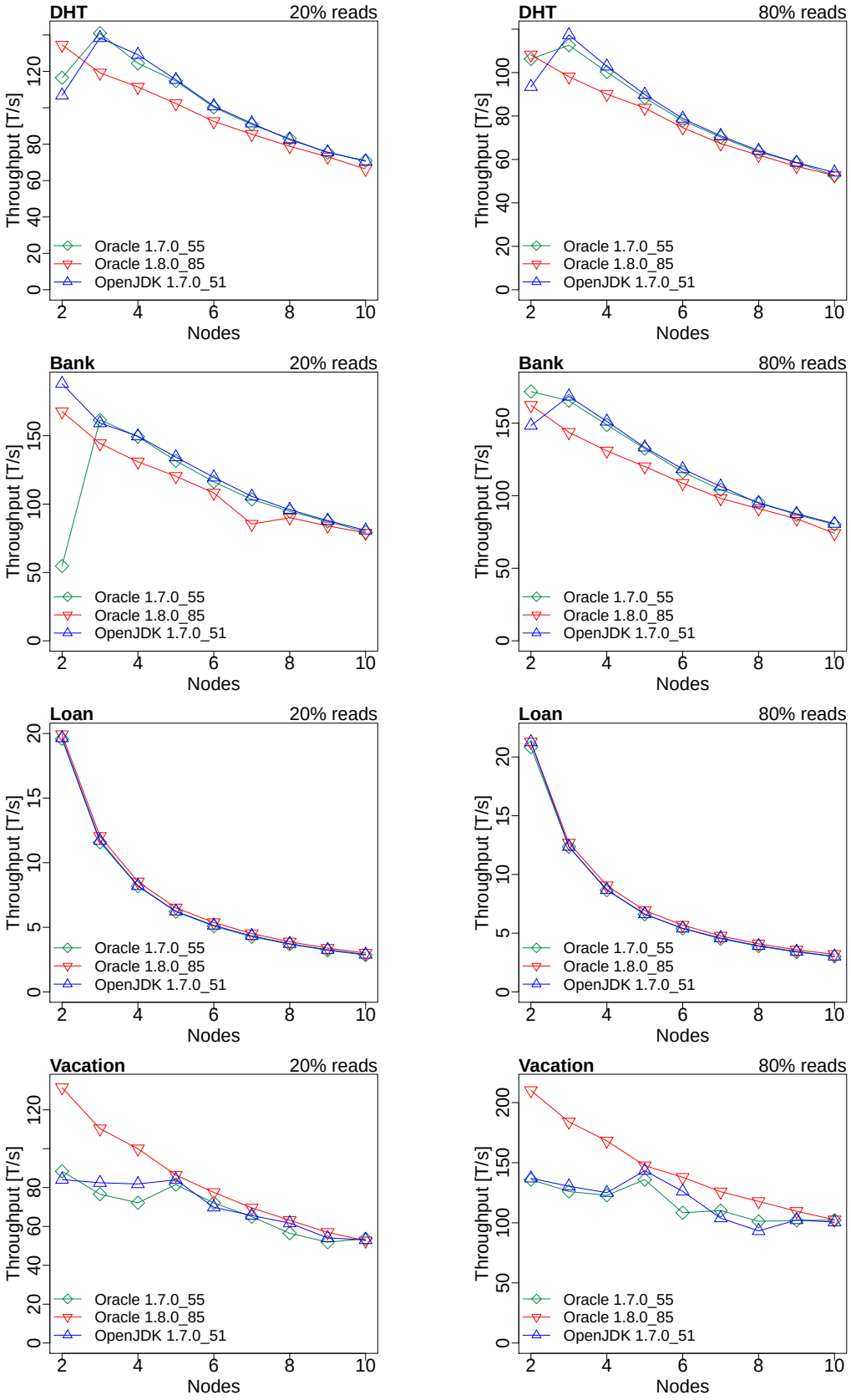


Figure 8.5: Evaluation results of Atomic RMI under various JREs.

Points on the graph represent the mean throughput in transactions per second (on the y-axis) from the given benchmark run on a particular number of nodes (on the x-axis). The benchmarks indicate that Atomic RMI performs in a relatively similar manner. The most significant difference can be seen when relatively few nodes are involved. This is best visible in Vacation for tests with 4 nodes or fewer, where Oracle 1.8.0_85 significantly outperforms either of the Java 7 implementations. The results also show a decline in throughput as more nodes are added. This is because each added node increases the rate of conflicts between transactions, as well as network congestion.

8.1.3 Discussion

The transaction abstraction is easy for programmers to use, while hiding complex synchronization mechanisms under the hood. We use that to full effect by employing SVA+R, an algorithm based on solid theory that allows high parallelism. Additionally, the pessimistic approach that is used in the underlying algorithm allows our system to present fewer restrictions to the programmer with regard to what operations can be included within transactions. Apart from limited transaction nesting, very little is forbidden within transactions.

Supremum-based early release makes our programming model efficient and relatively burden-free (especially, when static analysis is employed). Upper bounds on object calls are hard to estimate but the effort pays off since they allow to release objects as early as possible in certain cases. Our evaluation shows that due to the early release mechanism, Atomic RMI has a significant performance advantage over fine grained locks.

However, given the results of our evaluation, it is necessary to implement a different versioning algorithm that will distinguish between reads and writes, while retaining the early release mechanism. Combining the two optimizations should improve the efficiency of the system even further. We proceed to test this prediction by introducing Atomic RMI 2 an improved version of Atomic RMI that implements exactly that kind of a concurrency control algorithm.

8.2 Atomic RMI 2

In this section we present *Atomic RMI 2*, an implementation of OptSVA-CF+R (described in Section 6.4) that builds on Atomic RMI. The creatively named Atomic RMI 2 provides a simple-to-use API that allows programmers to implement consistent transactions as straightforwardly as if using much simpler mechanisms, such as distributed coarse-grained locking. However, instead of employing an operation-type agnostic concurrency control algorithm like its predecessor, Atomic RMI 2 uses one which recognizes three classes of operation types: reads, writes, and updates, and applies optimizations with respect to the execution of these operations. Atomic RMI 2 uses the same API as Atomic RMI, only extended to allow the identification of these operation types in the objects' interfaces, as well as to allow transactions to specify separate suprema for each operation type.

As a result of distinguishing operation types, the algorithm allows for the execution of transactional code to be highly parallelized, which leads to the improved performance of the DTM system. We demonstrate this in a comprehensive evaluation of Atomic RMI 2, showing that it produces a significant performance increase over its predecessor as well as various lock-based distributed concurrency control solutions. In addition, we show that Atomic RMI 2 performs better than, or comparably to HyFlow2 (depending on contention

```java
interface Transaction {
  Transaction();
  Transaction(boolean reluctant);

  <T> T reads(T obj);
  <T> T writes(T obj);
  <T> T updates(T obj);
  <T> T accesses(T obj);

  <T> T reads(Tobj, int rub);
  <T> T writes(T obj, int wub);
  <T> T updates(T obj, int uub);
  <T> T accesses(T obj, int rub,
                        int wub,
                        int uub);

  void start();
  void commit();
  void retry();
  void abort();

  void start(Transactional runnable);
}

interface Transactional {
  void run(Transaction t);
}
```

(a) Atomic RMI 2 transaction interface.

```java
interface Account extends Remote {
  @Access(Mode.READ) int balance();
  @Access(Mode.UPDATE) void deposit(int amount);
  @Access(Mode.UPDATE) void withdraw(int amount);
  @Access(Mode.WRITE) void reset();
}

class AccountImpl
    extends TransactionalUnicastRemoteObject
    implements Account {

  private int balance = 0;

  public int balance() {
    return balance;
  }
  public void deposit(int amount) {
    balance += amount;
  }
  public void withdraw(int amount) {
    balance -= amount;
  }
  public void reset() {
    balance = 0;
  }
}
```

(b) Shared object definition.

```java
Registry r = LocateRegistry.getRegistry(...);

Transaction t = new Transaction();

Account a = t.accesses(r.lookup("A"), 1, 0, 1);
Account b = t.updates(r.lookup("B"), 1);

t.start(new Transactional() {
  void run(Transaction t) {
    a.withdraw(100);
    b.deposit(100);

    if (a.balance() < 0)
      t.abort();
  }
});
```

(c) Transaction definition (`Transactional` object).

```java
Registry r = LocateRegistry.getRegistry(...);

Transaction t = new Transaction();

Account a = t.accesses(r.lookup("A"), 1, 0, 1);
Account b = t.updates(r.lookup("B"), 1);

t.start()

a.withdraw(100);
b.deposit(100);

if (a.balance() < 0)
  t.abort();
else
  t.commit();
```

(d) Transaction definition (in-line).

Figure 8.6: Atomic RMI 2 API and examples.

and operation length), but does so while avoiding aborting transactions altogether, thus allowing the use of irrevocable operations. Given this, we show that a pessimistic system can be as well-performing as an optimistic one. In this way we also introduce a CF DTM with competitive performance that was lacking.

8.2.1 Overview

Atomic RMI 2 is a re-implementation of the Atomic RMI framework. Because of this, the two systems share the same architecture and most of the same API. The differences stem from the requirements of the improved concurrency control algorithm, as follows. Since OptSVA-CF+R requires that read, write, and update methods be handled differently, interfaces of shared objects are declared differently in Atomic RMI 2. Furthermore, since OptSVA-CF+R uses separate upper bound values for the three types of operations, the definition of transaction preambles changes in Atomic RMI 2 to allow for separate definition of suprema for each operation type. Finally, Atomic RMI 2 introduces new buffer types and new modules for controlling threads, to meet the requirements of OptSVA-CF+R for buffering remote objects without synchronization and for transaction-local

asynchronous execution. We discuss each of these features below.

Shared Remote Objects

Atomic RMI 2 shared remote objects are unicast RMI objects, that implement an arbitrary interface and subclass the `TransactionalUnicastRemoteObject`, just like in Atomic RMI. However, since OptSVA-CF+R requires that operations be categorized as either reads, writes, or updates, the programmer should provide annotations in the interface to indicate each method's category. This is done via the `@Access` annotation which takes either the `Mode.READ`, `Mode.WRITE`, or `Mode.UPDATE` argument to indicate specific operation categories. These categories accord with the categories defined by OptSVA-CF+R: a read operation cannot modify the state of the object and may return a value, a write operation cannot view the state of the object and cannot return a value, and an update operation may both modify and view the state of the object, and may return a value. We show an example of the bank account objects' interface with annotations in Fig. 8.6b. If a method is not otherwise annotated, it is conservatively assumed to be an update method.

Transaction Objects

Clients execute operations on shared objects as part of transactions using the transaction interface provided by Atomic RMI 2 (see Fig. 8.6a). The API is analogous to that used in Atomic RMI (see Section 8.1.1): transactions can be either defined in-line or via `Transactional` interface (see Fig. 8.6c and 8.6d), and they can also be prevented from aborting by being defined as reluctant (which makes Atomic RMI 2 switch its concurrency control algorithm from OptSVA-CF+R to ROptSVA-CF+R).

The one significant difference introduced into Atomic RMI 2 transactions' API is that in the preamble the programmer specifies which objects will be used by the transaction and how, by passing the reference retrieved from the RMI registry to method `reads`, `writes`, `updates`, or `accesses`. The read method is used to specify that only read operations will be executed on that object by this transaction. Similarly, the `writes` and `updates` methods are used to specify that only writes or updates, respectively, will be executed on the object by this transaction. Finally, `accesses` is used to declare that methods of any kind may be executed on this object within this transaction.

The programmer can use variants of these methods to also provide suprema for any object used by the transaction. There are three types of suprema that indicate the maximum number of times the transaction will execute read, write, and update methods, on each shared object throughout the execution of the code. In the example in Fig. 8.6c the preamble declares the transaction will invoke at most one read method, no write methods, and at most one update method on shared object `A` (line 5). It also declares that the transaction will execute at most one update method, and no read or write methods on `B` (line 6).

Buffering

Atomic RMI 2 implements copy buffers just like Atomic RMI, but also introduces an implementation of the log buffer `log` used by OptSVA-CF+R. A log buffer for some object implements the same interface as the original object, but when an operation is supposed to be executed on the log buffer, the code of the method is not executed right away. Instead, a method object is created, and this object as well as the invocation's arguments are appended to a linked list associated with the buffer. This does not require any synchronization with the actual object that the log buffer represents. However, the log

buffer in our implementation does not execute any methods until it is applied explicitly to an actual object, by the concurrency control algorithm (which does require synchronization). When the log buffer is applied, each of the methods on its list is executed on the actual object in sequence.

Executor Thread

OptSVA-CF+R calls for asynchronous execution of certain procedures using separate threads. Given the overhead that starting a thread creates, Atomic RMI 2 uses one executor thread per JVM. The executor thread is always running and transactions assign it tasks. Each task consists of a condition and code. The code of the task is meant to be executed only when the condition is satisfied. Once the thread receives a task, it checks whether it can be immediately executed. If not, it queues up the task and waits until any of the two counters that can impact the condition change value (`lv` and `ltv`). When any of the counters change, the thread re-evaluates the relevant conditions and executes the task, if the condition so allows.

8.2.2 Evaluation

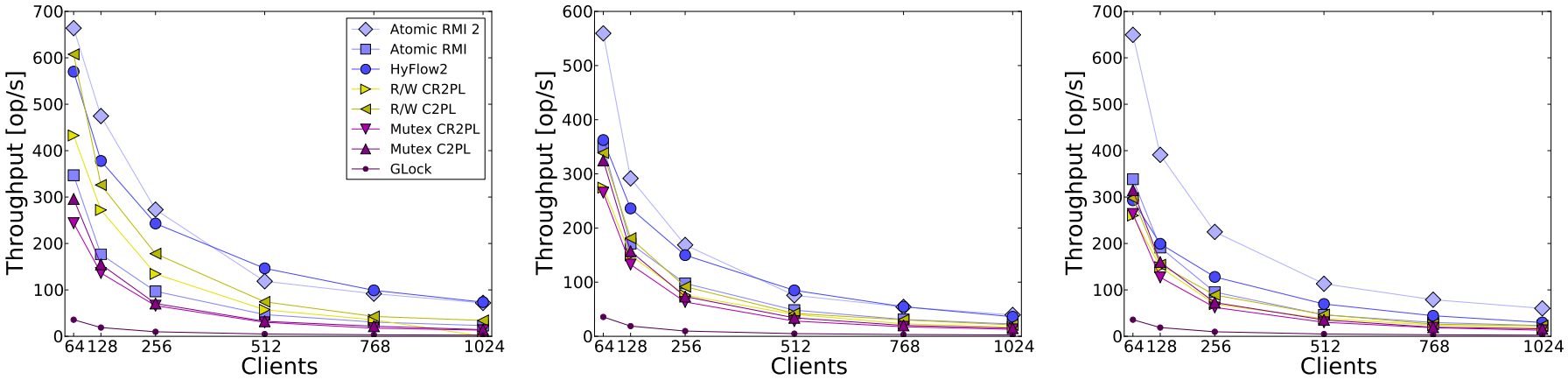
In this section we present the results of a practical evaluation of Atomic RMI 2 in the context of other distributed TM concurrency control mechanisms operating in a similar system model.

Benchmark

We perform our evaluation using a 16-node cluster connected by a 1Gb network. Each node had two quad-core Intel Xeon L3260 processors at 2.83 GHz with 4 GB of RAM each and runs OpenSUSE 13.1 (kernel 3.11.10, x86_64 architecture). We use Groovy version 2.3.8 with the 64-bit Java HotSpot(TM) JVM version 1.8 (build 1.8.0_25-b17).

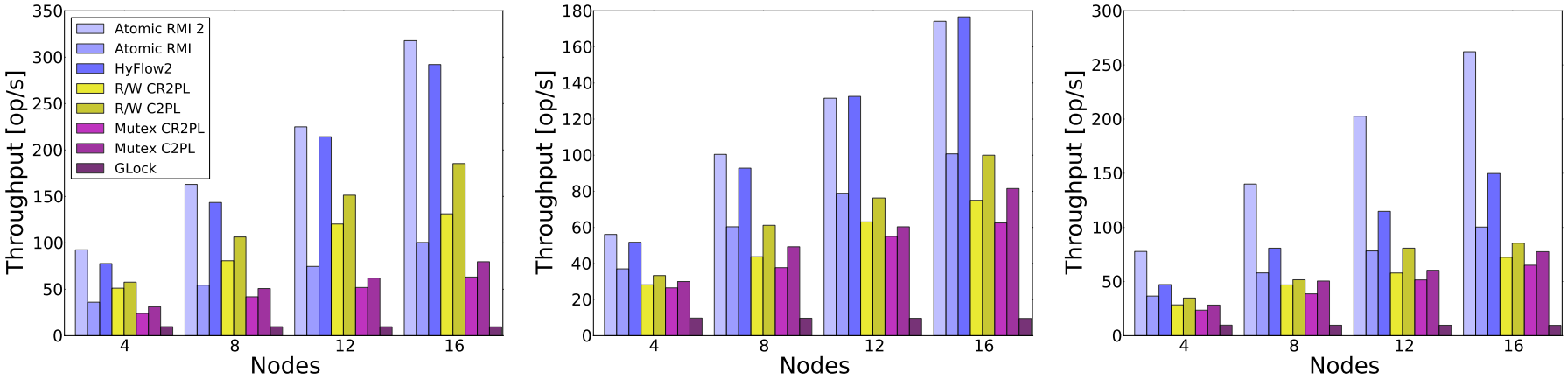
The evaluation is performed using our own distributed implementation of Eigenbench [47]. Eigenbench is a flexible, powerful, and lightweight benchmark that can be used for comprehensive evaluation of multicore TM systems by simulating a variety of transactional application characteristics.

Eigenbench uses three arrays of shared objects, each of which is accessed with a different level of contention. The *hot array* contains some number n of objects that can be accesses by transaction in any thread. The access to objects in the hot array is controlled by the TM. The *mild array* contains n objects per thread. The access to these objects is also controlled by the TM, but the objects are partitioned in such a way, that no two transactions ever conflict on them. The third, *cold array* is populated and partitioned like the mild array, but it is only accessed non-transactionally. Each object within any of the three arrays is a *reference cell*, i.e., an object that holds a single value, that can be either read or written to—an object-oriented equivalent of a variable. These arrays are accessed by client transactions. Each transaction accesses semi-randomly selected objects in all three arrays in random order, with the exception that the number of accesses to each type of array is specified, and the ratio of read operations to write operations on each type of array is also specified. The benchmark has a specified locality, which is a probability with which transactions will access the same object several times. When an object is being selected by a transaction, a random number is generated, and if it is below the locality probability, the object is selected at random from the transaction's history of objects accessed thus far. Otherwise, the object is selected randomly from the pool of all shared objects. Locality and the length of the history are parameters of the benchmark.



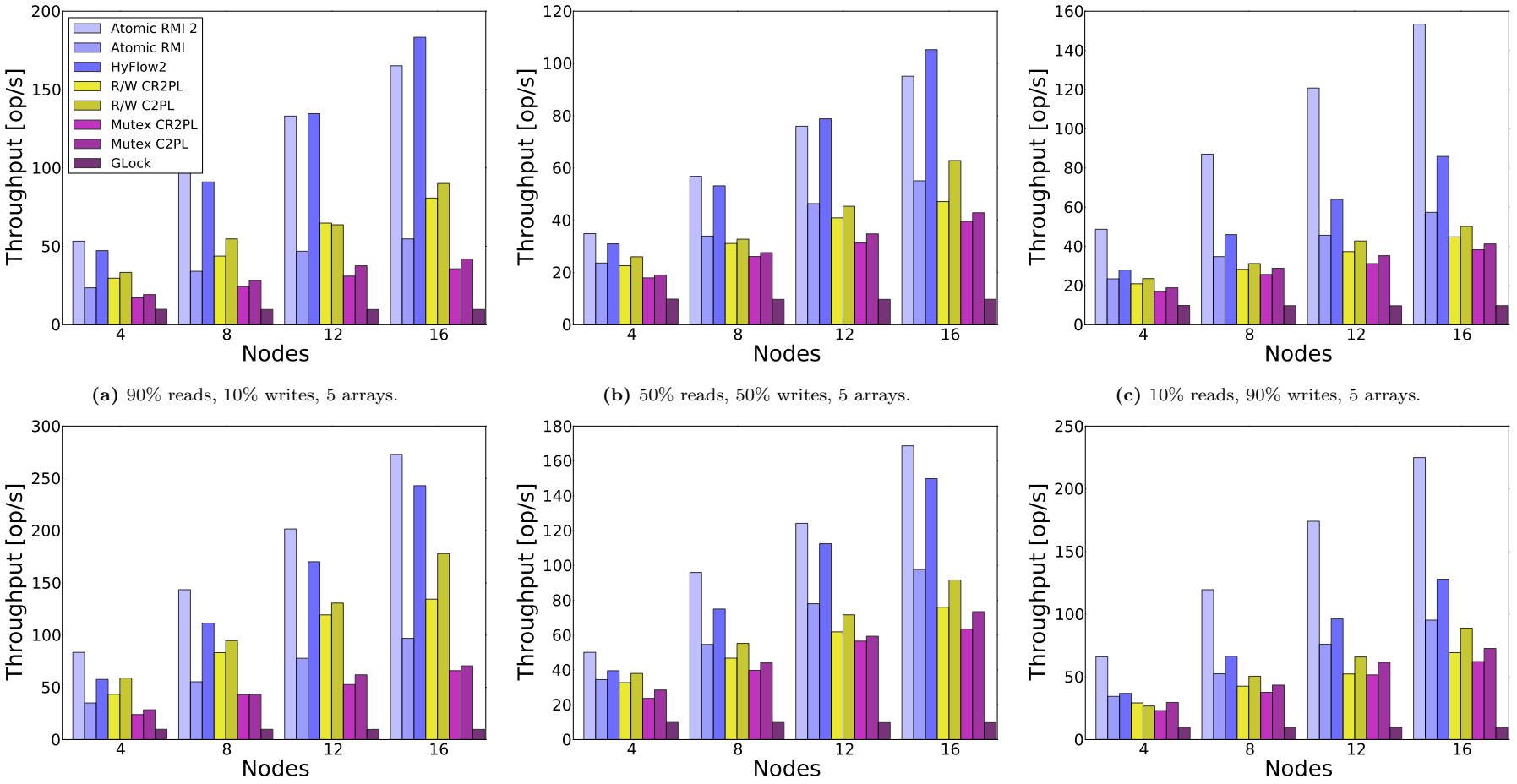
(a) 90% reads, 10% writes. **(b)** 50% reads, 50% writes. **(c)** 10% reads, 90% writes.

Figure 8.7: Throughput vs client count (hot array accesses).



(a) 90% reads, 10% writes. **(b)** 50% reads, 50% writes. **(c)** 10% reads, 90% writes.

Figure 8.8: Throughput vs node count (hot and mild array accesses).



(a) 90% reads, 10% writes, 5 arrays.

(b) 50% reads, 50% writes, 5 arrays.

(c) 10% reads, 90% writes, 5 arrays.

(d) 90% reads, 10% writes, 10 arrays.

(e) 50% reads, 50% writes, 10 arrays.

(f) 10% reads, 90% writes, 10 arrays.

Figure 8.9: Throughput vs node count (hot array accesses).

Frameworks

The first framework we use for evaluation is Atomic RMI. Since Atomic RMI and Atomic RMI 2 use the same basic technology and provide the same guarantees, the comparison between them shows off the optimizations introduced in OptSVA-CF+R over pure SVA+R. The second framework we compare Atomic RMI 2 against is HyFlow2 [86], a state-of-the-art distributed TM system implemented in Scala. HyFlow2 implements the optimistic Transactional Forwarding Algorithm (TFA) (see Section 4.2.2) and operates in the data flow model. TFA is opaque but does not have provisions for irrevocable operations.

We also compare all three TM systems against distributed concurrency control solutions based on locks and C2PL algorithms. Specifically, we use distributed mutual exclusion locks (marked Mutex) and read-write locks (marked R/W Locks), both custom-tailored and implemented on top of Java RMI. In both solutions a lock is created for every shared object in the system. Each locking solution has two variants. The first variant is a straightforward usage where every transaction locks every object from its access set when it commences, and releases each of object on commit. This is equivalent to a *conservative rigorous two-phase locking* solution and satisfies opacity. We denote this variant as *CR2PL*. The second variant represents *conservative two-phase locking* (*C2PL*), and is a more advanced implementation from the programmer's point of view. Here, each transaction also initially locks each of the objects in its access set, but the programmer determines the last access on each object in the transaction's access set and manually releases the lock early (prior to commit). C2PL locking satisfies last-use opacity under the assumption that the last access is always determined correctly. Finally, we also use a solution with a *single global mutual exclusion lock* (*GLock*) that is acquired by each transaction for the duration of the transaction's entire execution. This produces a completely sequential execution and acts as a baseline for the purpose of the comparison.

Results

Fig. 8.7 illustrates the change of throughput (measured as the number of executed operations on shared data per second) as the number of clients increases from 64 (4 per node) to 1024 (64 per node). We show three scenarios, each executed on 16 nodes, with 10 arrays of each type per node. Each client executes 10 consecutive transactions, each with 10 operations on the hot array per transaction, with a 9÷1, 5÷5, or 1÷9 read-to-write operation ratio. Each operation takes around 3ms to execute, not counting the overhead from synchronization, network communication, or serialization overhead. This means operations are fairly long, which represents the complex computations. The locality of operations is set to 50% with a history of 5 operations.

The graphs show that all frameworks' throughput falls as the number of clients, and therefore contention, increases. The decline is steep until 256 clients, and it levels out by 1024 clients. All systems significantly outperform the serial execution forced through GLock. In the 90% read scenario HyFlow2 and Atomic RMI 2 outperform other frameworks by a significant margin of between 9 and 267% (not counting GLock), with the exception of R/W C2PL outperforming HyFlow2 at 64 clients. Atomic RMI 2 outperforms HyFlow2 initially (by 9–25%), but after 512 clients are introduced, HyFlow2 takes the lead (by 2–23%), and both frameworks throughputs eventually converge at the 1024 client mark.

In the other two scenarios, all frameworks suffer a decrease in throughput, but Atomic RMI 2 remains relatively efficient, outperforming all other frameworks, including HyFlow2, by 9–359%. The difference stems from the write-oriented optimizations in Atomic RMI 2 that allow the framework to tighten the executions in the presence

Scenario	Clients					
	64	**128**	**256**	**512**	**768**	**1024**
9÷1 ratio	66	74	79	86	84	89
5÷5 ratio	60	70	75	83	87	87
1÷9 ratio	66	74	79	86	84	89

Table 8.1: HyFlow2 abort rates for Fig. 8.7 [%].

of larger contingents of write operations, just as much as is possible in read-dominated schedules: objects are acquired for writing as late as possible and released prior to commit. Meanwhile other frameworks typically do not optimize write operations to the same extent. Specifically, HyFlow2 does not release early on writes, and R/W C2PL cannot perform any optimizations on writes, apart from early release on last write. In addition a degradation in Atomic RMI 2's performance is also partly explained by the need to introduce new threads to handle asynchrony, which can become a bottleneck and offset the gain from Atomic RMI 2's optimizations if other threads are also running on the same node (like client threads here).

Among the remaining frameworks, any C2PL always performs better than the apposite CR2PL variant, and R/W performs better than Mutex. Atomic RMI performs on par with Mutex C2PL and significantly below Atomic RMI 2.

Fig. 8.9 shows a change in throughput with constant contention as new nodes are introduced. In this scenario, we vary the number of nodes from 4 to 16 with 5 or 10 arrays of each type hosted on each node (yielding lower and higher contention respectively), and 16 clients running per node. Transactions only perform operations on the hot array. The remainder of parameters is as above. As more processors are introduced into the system, the number of transactions running in parallel increases, causing the throughput of all frameworks to increase as well.

In the 5-array scenarios in Fig. 8.9a–c the comparison shows that Atomic RMI 2 significantly and consistently outperforms Atomic RMI and all remaining frameworks, with the exception of HyFlow2. Specifically, Atomic RMI 2 achieves at least a 47% better throughput over Atomic RMI due to the introduced optimizations. The impact of read-only optimizations is visible in the 90% read scenario, where Atomic RMI 2 achieves up to a 201% advantage in throughput. Furthermore, the write optimizations give Atomic RMI 2 a performance boost of up to 167% over Atomic RMI in the 90% write scenario. In a more balanced scenario optimizations can be applied less often, leading to a slightly lower performance improvement of up to 72%. Note, that Atomic RMI's performance does not change with respect to the differences in workloads among scenarios, since Atomic RMI is agnostic of operation types. HyFlow2 and Atomic RMI 2 perform similarly in read dominated and balanced scenarios, with HyFlow2 outperforming Atomic RMI 2 by up to 10% in a 16-node system, and Atomic RMI 2 outperforming HyFlow2 by as much in a 4 node system. The similarities in performance stem from special handling of read-only variables in both systems. However, in a write dominated scenario, Atomic RMI 2 has a 77% percent throughput advantage, which we again attribute to extensive write-oriented optimizations employed in OptSVA-CF.

The 10-array scenarios in Fig. 8.9a–c yield similar results, but here, Atomic RMI 2 manages to consistently outperform HyFlow2, as well as other evaluated frameworks. This is because transactions have more objects from which to randomly select, so transactions tend to contain shorter subsequences of operations on the same objects, which allows Atomic RMI 2 to release more objects earlier.

Fig. 8.8 shows changes in throughput as above, but with longer transactions, that

Scenario	Nodes			
	4	**8**	**12**	**16**
9÷1 ratio, 5 arrays, hot	73	69	73	74
5÷5 ratio, 5 arrays, hot	81	83	80	82
1÷9 ratio, 5 arrays, hot	77	81	85	81
9÷1 ratio, 10 arrays, hot	65	68	67	63
5÷5 ratio, 10 arrays, hot	78	77	79	77
1÷9 ratio, 10 arrays, hot	75	76	80	79
9÷1 ratio, 10 arrays, hot & mild	66	67	67	66
5÷5 ratio, 10 arrays, hot & mild	78	79	82	81
1÷9 ratio, 10 arrays, hot & mild	74	81	81	81

Table 8.2: HyFlow2 abort rates for Fig. 8.8 and 8.9 [%].

perform mild array accesses in addition to hot array accesses. Hence each transaction performs 10 operations on the hot array and 10 operations on the mild array, in the same read-to-write ratios. Since accesses on mild arrays never lead to conflicts, the average contention is much lower in this scenario than the previous. Because of this, throughput increases for each framework. Atomic RMI 2 performs similarly to HyFlow2 in the balanced scenario (up to 2% reduction or 8% improvement), slightly better in the read dominated (8–19% improvement), and significantly better in the write dominated scenario (64–76%). Both HyFlow2 and Atomic RMI 2 perform significantly better than all other frameworks, including Atomic RMI. The results are similar to those in the previous scenario, but show that Atomic RMI 2's advantage decreases in lower contention, which we attribute to the overhead introduced by the instrumentation and asynchronous executions. Instrumentation requires that new objects are created on-the-fly, which takes time and uses processing power. Asynchronous execution requires that new threads are created and maintained by each JVM to handle various computations imposed by OptSVA-CF+R (usually buffering) which puts strain on the processor, especially since these threads compete with client threads and server threads running on the same JVM.

The abort rates of Atomic RMI 2 and Atomic RMI remain at 0% throughout the evaluation (despite none of the transactions being reluctant), while 60–89% of HyFlow2 transactions abort and retry at least once due to conflicts, depending on the scenario (see Tables 8.1 and 8.2). This means, that irrevocable operations are likely to be aborted and re-executed. On the other hand, Atomic RMI 2 manages to rival the efficiency of an optimistic TM system while bypassing problems with irrevocable operations completely.

8.2.3 Discussion

Throughout we see that Atomic RMI 2 significantly outperforms Atomic RMI and other lock-based distributed concurrency control mechanisms, and performs similarly to or better than a state-of-the-art optimistic distributed TM, all without the need to use aborts and, thus, without complicating irrevocable operation executions, and while employing the reflection-based mechanisms that allow to use CF model. We also see that Atomic RMI 2 performs best in read-dominated scenarios, but becomes really competitive in write-dominated scenarios, where the buffering- and asynchrony-related write-oriented optimizations make a real difference to throughput. Given this, we successfully demonstrate that a pessimistic system can be as well-performing as an optimistic one.

9 Precompiler

In this chapter we present a precompiler for Atomic RMI which can be used to statically derive the *a priori* information required by Atomic RMI's underlying concurrency control algorithm: the access set of each transaction, and suprema—upper bounds on the number of times each transaction will access each object in its access set throughout the transaction's execution. This extends our research in [72, 73].

First, we describe the static analysis algorithm used by the precompiler. The precompiler uses a static analysis algorithm based on data flow analysis to establish information about values and paths and region analysis to tally method calls. We expand regions with additional properties so that the final, vital part of the analysis becomes straightforward. We also describe a use of a natural positive set extended by an absorbing value to count uncertain executions. In effect, we infer upper bounds (either concrete or infinite) conservatively but safely.

In the following sections we discuss the implementation of the precompiler itself, and discuss the effectiveness as well as other applications of the static analysis. In the final section we briefly survey related work.

9.1 Static Analysis

To derive the suprema the algorithm performs multiple passes over the input code in the form of an intermediate language. Three passes correspond to the three phases that form our algorithm: *value analysis*, *region analysis*, and *call count analysis*. In addition, another pass is performed before value analysis to identify loops. Value analysis predicts possible values of variables in the code. It also identifies unfeasible or dead code, and unfolds loops. Region analysis uses the results of value analysis to convert the input code into regions. Finally, call count analysis examines these regions to produce the upper bounds on method call counts. We describe our use of Jimple and each of the phases of the algorithm in detail below.

9.1.1 Translation to Jimple

In order to analyze a program in Java we first translate it into an intermediate representation called *Jimple* [88] using the Soot framework [87]. We use Jimple as an intermediate

Identifiers	$j \in \mathit{Ident}$	
Constants	$c \in \mathit{Const}$	
Labels	$l \in \mathit{Lab}$	
Types	$t \in \mathit{Type}$	
Fields	$f \in \mathit{Field}$	$::= j : t$
Immediates	$i \in \mathit{Imed}$	$::= j \mid c$
Right-hand values	$r \in \mathit{Rval}$	$::= i \mid i[i] \mid i.[f] \mid [f]$
Methods	$m \in \mathit{Meth}$	$::= \texttt{invoke}\ i.[j(j_1, ..., j_n)](i_1, ..., i_n)\{b_1, ..., b_n, \}$
Conditions	$p \in \mathit{Cond}$	$::= i == i \mid i \geqslant i \mid i > i \mid i \leqslant i \mid i < i \mid i \neq i$
Expressions	$e \in \mathit{Expr}$	$::= i + i \mid i\ /\ i \mid i\ *\ i \mid i\%i \mid -i \mid i - i \mid i \mid i \mid i\ \&\ i$ $\mid i\ \texttt{xor}\ i \mid i \gg i \mid i \ll i \mid (t)i \mid i\ \texttt{instanceof}\ t$ $\mid \texttt{new}\ t \mid \texttt{new}\ t[i_1]...[i_n] \mid \texttt{length}\ i \mid p$
Statements	$s \in \mathit{Stmt}$	$::= \texttt{switch}(i)\{\texttt{case}\ c_1 : l_1; ...; \texttt{case}\ c_n : l_n; \texttt{default:}\ l_0\}$ $\mid \texttt{if}\ p\ \texttt{goto}\ l_1\ \texttt{else}\ l_2 \mid l \mid j = m \mid j = r \mid m$ $\mid \texttt{goto}\ l \mid \texttt{return}\ i$
Blocks	$b \in \mathit{Bloc}$	$::= l : b_1; ...; b_n; \mid b_1; ...; b_n; \mid s$

Figure 9.1: Jimple syntax (altered from [88]).

language because it is much better suited for analysis than either Java source code or bytecode. The reason for this is that Jimple is a 3-address code representation with a very limited instruction set consisting of 17 statements. In our earlier attempts to perform similar analyses using Java source code [72] we learned that such analyses become convoluted and the implementation costly in effort due to the number of constructs needing handling and the complexity of their semantics.

The part of Jimple syntax that is pertinent to our further discussion is presented in Fig. 9.1. The semantics are mostly straightforward, the reader is referred to [88] for details and the complete language. The constructs most important to us are the conditional statements, switch statements, method invocations, assignments, and labeled blocks. We introduce superficial alterations to the syntax to suit further description of the algorithm. We treat labels as statements and place them at the beginning of labeled blocks. We modify the conditional statement to define target labels for both outcomes instead of having a succeeding block of code called if the condition is false. We do not distinguish among different sorts of method invocations—interface, special, virtual, and static—and we remove type information from invocations while adding a direct definition of the methods' arguments and a set of possible bodies. We also fix method invocations nested in other statements by defining a separate assignment statement instead where the results of the invocation are assigned to an identifier. We show an example Java program using Atomic RMI distributed transactions in Fig. 9.2a translated to the altered form of Jimple in Fig. 9.2b (lines 2, 3 are omitted because they are generated from Jimple later—see Section 9.2.2 for details).

For the purposes of analysis the input program is represented as *Control Flow Graphs* (CFGs) and each method's body is a separate graph. Most statements in Jimple will have one incoming and outgoing edge. The conditional statement will have 2 outgoing edges, and the `switch` statement will have one more outgoing edge than it has conditions. Loop headers and labeled blocks will have more incoming edges. Invoke statements point to CFGs of other method bodies.

9.1.2 Value Analysis

As a preliminary to the value analysis we find loops in code. A loop consists of a head and a body. A loop head is a statement s that dominates any other statement s' (all paths

```
1 Transaction t = new Transaction();
2 a = t.accesses(a, 2); // generated, upper bound: 2
3 b = t.accesses(b, 1); // generated, upper bound: 1
4 t.start();
5 int balance = a.balance();
6 if (balance >= sum) {
7   a.withdraw(sum);
8   b.deposit(sum);
9   k.commit();
10 } else
11   k.abort();
```

(a) Java.

```
1  k = new soa.atomicrmi.Transaction;
2  invoke k.[<init>()]()(){$b0};
3  invoke k.[start()]()(){$b1};
4  balance = invoke a.[balance()]()(){$b4};
5  if balance < sum goto label1 else label0;
6 label0:
7  invoke a.[withdraw(@parameter0)](sum){$b5};
8  invoke b.[deposit(@parameter0)](sum){$b6};
9  invoke k.[commit()]()(){$b2}; return null;
10 label1:
11  invoke k.[abort()]()(){$b3};
```

(b) (Altered) Jimple.

Figure 9.2: Example Atomic RMI code.

$$
\begin{aligned}
&\mathbb{G}(s) \triangleq \mathbb{S} \triangleq (\mathbb{S}_V, \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)\\
&\mathbb{G}(s) = \mathsf{eval}(\mathsf{join}(\{\mathbb{G}(p) \mid s \ \mathbf{succ}\ p\}), s)\\
&\mathsf{eval}(\mathbb{S}, j = r) \triangleq (\mathbb{S}_V[j \mapsto \{\mathsf{val}(r, \mathbb{S}_V)\}], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)\\
&\mathsf{eval}(\mathbb{S}, j = m) \triangleq \mathbb{S}' = \mathsf{eval}(\mathbb{S}, m), (\mathbb{S}_V \oplus \mathbb{S}'_V[j \mapsto \{\mathsf{val}(m, \mathbb{S}_V)\}], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)\\
&\mathsf{eval}(\mathbb{S}, \texttt{invoke}\ i.[j(j_1, ..., j_n)](i_1, ..., i_n)\{b_1, ..., b_m\}) \triangleq\\
&\quad \text{case } \mathsf{depth}(i.j) \rightarrow \mathbb{S}_V[k \mapsto \omega k \in \mathsf{defs}(b_1) \cup ... \cup \mathsf{defs}(b_m)], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)\\
&\quad \text{otherwise} \rightarrow \mathbb{S}' = (\mathbb{S}_V[j_1 \mapsto \mathsf{val}(i_1, \mathbb{S}_V), ..., j_n \mapsto \mathsf{val}(i_n, \mathbb{S}_V)], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I),\\
&\qquad \mathsf{join}(\mathsf{eval}(\mathbb{S}', b_1), ..., \mathsf{eval}(\mathbb{S}', b_m))\\
&\mathsf{eval}(\mathbb{S}, l) \triangleq (\mathbb{S}_V \oplus \mathbb{S}_P(l), \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I)\\
&\mathsf{eval}(\mathbb{S}, s : \texttt{return}\ i) \triangleq (\mathbb{S}_V, \mathbb{S}_P, \mathbb{S}_D \cup \{(s, s') \mid s \ \mathbf{pdom}\ s', s' \in \mathit{Stmts})\}, \mathbb{S}_I)\\
&\mathsf{eval}(\mathbb{S}, s : \texttt{if}\ p\ \texttt{goto}\ l_1\ \texttt{else}\ l_2) \triangleq\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{true} \rightarrow (\mathbb{S}_V, \mathbb{S}_P\left[l_1 \mapsto \mathbb{S}_P[p \mapsto \texttt{true}]\right], \mathbb{S}_D \cup \{(s, l_2)\}, \mathbb{S}_I)\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{false} \rightarrow (\mathbb{S}_V, \mathbb{S}_P\left[l_2 \mapsto \mathbb{S}_P[p \mapsto \texttt{false}]\right], \mathbb{S}_D \cup \{(s, l_1)\}, \mathbb{S}_I)\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \omega \rightarrow (\mathbb{S}_V, \mathbb{S}_P\left[l_1 \mapsto \mathbb{S}_P[p \mapsto \texttt{true}], l_2 \mapsto \mathbb{S}_P[p \mapsto \texttt{false}]\right], \mathbb{S}_D, \mathbb{S}_I)\\
&\mathsf{eval}(\mathbb{S}, s : \texttt{switch}(i)\{\texttt{case}\ c_1\texttt{:}\ l_1\texttt{;} ...\texttt{;}\ \texttt{case}\ c_n\texttt{:}\ l_n\texttt{;}\ \texttt{default:}\ l_0\}) \triangleq (\mathbb{S}_V,\\
&\quad \mathbb{S}_P\left[l_1 \mapsto \mathbb{S}_P(l_1)[j = \mathsf{val}(c_1)], ..., l_n \mapsto \mathbb{S}_P(l_n)[j = \mathsf{val}(c_n)]\right], \mathbb{S}_D\\
&\quad \cup\{(s, l_k) \mid \mathsf{pred}(c_k = j, \mathbb{S}) = \texttt{false} \vee \mathsf{pred}(c_r = j, \mathbb{S}) = \texttt{true}, k = 1, ..., n, r = 1, ..., k\}\\
&\quad \cup\{l_0 \mid \mathsf{pred}(\exists k, c_k = j, \mathbb{S}) = \texttt{true}, k = 1, ..., n\}, \mathbb{S}_I)\\
&\mathsf{eval}(\mathbb{S}, s \in \mathbb{H}) \triangleq \mathsf{evalloop}(s, \mathbb{G}, \mathbb{L}(\mathsf{id}(s)), 1, L)\\
&\mathsf{eval}(\mathbb{S}, s \in \mathbb{B} \wedge s \notin \mathbb{H}) \triangleq \mathbb{S}\\
&\mathsf{join}(\mathbb{S}^1, ..., \mathbb{S}^n) \triangleq \big(\{k \mapsto \mathbb{S}^1_V(k) \cup ... \cup \mathbb{S}^n_V(k) \mid k \in (\mathrm{dom}\ \mathbb{S}^1_V \cup \mathrm{dom}\ \mathbb{S}^n_V)\}, \{l \mapsto\\
&\quad \{k \mapsto \mathbb{S}^1_P(l)(k) \cup ... \cup \mathbb{S}^n_P(l)(k)\} \mid l \in \mathrm{dom}\ \mathbb{S}^1_P \cup ... \cup \mathbb{S}^n_P, k \in \mathrm{dom}\ \mathbb{S}^1_P(l) \cup ... \cup \mathbb{S}^n_P(l)\},\\
&\quad \mathbb{S}^1_D \cup ... \cup \mathbb{S}^n_D, \{k \mapsto \max_{i=1,...,n}(\mathbb{S}^i_I(k)) \mid k \in (\mathrm{dom}\ \mathbb{S}^1_I \cup \mathrm{dom}\ \mathbb{S}^n_I)\}\big)\\
&\mathbb{S}'_V \oplus \mathbb{S}''_V \triangleq \{k \mapsto \mathbb{S}'_V(k) \cup \mathbb{S}''_V(k) \mid k \in (\mathrm{dom}\ \mathbb{S}'_V(k) \cup \mathrm{dom}\ \mathbb{S}''_V(k))\}
\end{aligned}
$$

Figure 9.3: Value analysis.

from the start to s' lead through s [2], denoted s **dom** s') while simultaneously being the successor of s' (there is a path from s' to s). A loop body is a sequence of statements all of which are dominated by a loop head and have that loop head as their successor. We gather the heads in set $\mathbb{H}$ and create map $\mathbb{L}$ which contains a unique identifier of each statement h from $\mathbb{H}$ as a key mapped to a set of statements whose elements are all dominated by h.

The first phase of the analysis is a forward data flow analysis performed on the CFG. Its main purpose is threefold: to establish the possible values of variables at each node of the CFG representing the program, to count the maximum number of loop iterations through loop unfolding, and to establish which nodes of the CFG are dead or unfeasible (will not be executed). There are two principal functions in value analysis, eval and join. These functions are used to compute members of global state $\mathbb{G}$, a data structure that results from the analysis. We present all of those elements in Fig. 9.3 and describe them below.

Global state $\mathbb{G}$ maps Jimple statements to states which apply to them. Global state is constructed during value analysis by constructing a state for each statement using a transfer function eval and an aggregation of states for the predecessors of a given statement using join. We designate individual states $\mathbb{S}$, such that $\mathbb{S}$ is a quadruple consisting of a value map $\mathbb{S}_V$, an inferred value map $\mathbb{S}_P$, a dead edge set $\mathbb{S}_D$, and a loop iteration map $\mathbb{S}_I$. $\mathbb{S}_V$ is a map of locals (identifiers and constants) to sets of values—it indicates what values a given variable or constant may take at this point in the program. $\mathbb{S}_P$ maps labels (names of blocks) to value maps and indicates assumptions about values of variables and constants inferred from conditions that will apply at a particular succeeding statement. $\mathbb{S}_D$ contains pairs of statements indicating edges that will definitely not be used in the execution of the program. $\mathbb{S}_I$ is a map of loop heads to numbers indicating the maximum estimated iteration count of the loop, or an unknown value. All components of the state are initially empty.

Transfer function eval is the key function of the analysis. It analyzes each Jimple statement and establishes the state of the program that holds after the statement is evaluated. The resulting state depends on the type of statement and the state before that statement.

When encountering an assignment of a right-hand side expression r to an identifier j, a new mapping is added to $\mathbb{S}_V$ that maps j to the set of possible values of expression r. When eval encounters an assignment of the results of method invocation m to identifier j, first m is evaluated separately and state after its evaluation $\mathbb{S}'$ is extended by the mapping of j to the result of m. A method invocation itself is analyzed by first extending the value map by parameter identifiers mapped to the values of arguments. Then all possible bodies are evaluated and the results are joined (the particular bodies are identified from the type hierarchy and arguments but we leave the details to Soot). But if recursion exceeds a depth L all the values defined within possible method bodies are set to unknown (this degrades precision but maintains safety). L must be tuned to a given application. A label l extends the value map with predictions from the inferred map. A `return` statement adds all other statements it dominates to $\mathbb{S}_D$.

When analyzing an `if` statement the expression that is the condition is checked. If the condition yields true then the edge in the CFG from the current statement to label l_2 is added to dead edges, and predictions about variables are made under the assumption that the condition will be true at label l_1. Conversely, if the condition yields false the edge from the statement to l_1 will be dead and predictions will be made for l_2 under the assumption that the condition is false. If the condition yields an unknown, no edges will be added to the dead edge set, but predictions for both l_1 and l_2 will be made. A `switch` statement is analyzed by creating a prediction for each constant $c_1, ..., c_n$ that the local i is equal to it at an appropriate label $l_1, ..., l_n$. Furthermore, if any of the constants c_k is definitely equal to i, edges from this statement to labels subsequent to that constant $l_{k+1}, ..., l_n$ and the default label l_0 are added to the dead edge set $\mathbb{S}_D$.

Function join (Fig. 9.3) is responsible for joining states and is used when a statement has two or more incoming edges. Each component of the state is joined with its counterpart in the second state. Sets $\mathbb{S}_D$ are added together. The keys and values are copied to a new map, and if a key is present in both maps, the values are added ($\mathbb{S}_V$, $\mathbb{S}_P$) or the higher one is selected ($\mathbb{S}_I$).

We use the following helper functions within eval. Function val substitutes values from a value map for identifiers and constants (where possible) in a given expression and evaluates it to establish a set of values that the expression may yield. The returned set may consist of a single value, any number of elements or contain the unknown value ω. We use the function pred in a similar manner, except that only conditional expressions are evaluated and a single ternary value is returned—`true`, `false`, or ω. We use depth

$$
\begin{array}{l}
\mathsf{evalloop}(s, \mathbb{G}', \mathbb{U}, i, L) \triangleq \\
\quad \mathbb{G}'' = \mathbb{G}', \quad \mathbb{G}''(u) = \mathsf{eval}(\mathsf{join}(\{\mathbb{G}''(u) \mid u \ \mathbf{succ}\ p \wedge u \in \mathbb{U}\})), \\
\quad \mathbb{E} = \{e \mid s \ \mathbf{succ}\ e \wedge s \notin \mathbb{U} \wedge e \in \mathbb{U}\}, \\
\quad \mathbb{E}' = \mathbb{E} \setminus \{d \mid \mathbb{G}''(d) = \mathbb{S}', \mathsf{unpredecessed}(\mathbb{S}'_D, d) \wedge d \in \mathbb{E}\}, \\
\quad \mathbb{S}^e = \mathsf{join}(\{\mathbb{G}''(e) \mid e \in \mathbb{E}'\}), \\
\quad \mathbb{Z} = \{(b,h) \mid h \ \mathbf{dom}\ b \wedge h \ \mathbf{succ}\ b \wedge \nexists s \in \mathbb{U}, h \ \mathbf{succ}\ s \ \mathbf{succ}\ b\} \\
\quad \mathbb{S}^z = \mathsf{join}(\{\mathbb{G}''(b) \mid (b,h) \in \mathbb{Z}\}), \\
\quad \text{case } \mathbb{Z} \subseteq \mathbb{S}^z_D \vee (\forall (b,h) \in \mathbb{Z},\ \mathsf{unpredecessed}(\mathbb{S}^z_D, b)) \rightarrow (\mathbb{S}^e_V, \mathbb{S}^e_P, \mathbb{S}^e_D, \mathbb{S}^e_I[h \mapsto i]) \\
\quad \text{case } i > L \rightarrow (\mathbb{S}^e_V[k \mapsto \omega, k \in \mathsf{defs}(\mathbb{U})], \mathbb{S}^e_P, \mathbb{S}^e_D, \mathbb{S}^e_I[h \mapsto \omega]) \\
\quad \text{case } i \leqslant L \rightarrow \mathsf{evalloop}(h, \mathbb{G}'', \mathbb{U}, i+1, L) \\
\mathsf{unpredecessed}(\mathbb{S}_D, s) \triangleq \forall s\ \mathsf{succ}\ p,\ (p,s) \in \mathbb{S}_D \vee \mathsf{unpredecessed}(p) \\
\mathsf{defs}(\mathbb{U}) \triangleq \left\{ j \mid s \in \{j = m, j = r\} \wedge s \in \mathbb{U} \right\}
\end{array}
$$

Figure 9.4: Loop unfolding.

Unit regions	$U \in$	*Units*	$::=$ `unit`
Statement regions	$S \in$	*Statements*	$::=$ `statement` s
Invocation regions	$I \in$	*Invocations*	$::=$ `invoke` $j, R_1, ..., R_m, s$
Block regions	$B \in$	*Blocks*	$::=$ `block` $[R_1, ..., R_n]$
Condition regions	$C \in$	*Conditions*	$::=$ `condition` p, R_1, R_2
Loop regions	$L \in$	*Loops*	$::=$ `loop` h, R
Regions	$R \in$	*Regions*	$::= U \mid S \mid I \mid B \mid C \mid L$

Figure 9.5: Region-based intermediate representation.

to find out the depth of a method's recursion. Function id produces a unique identifier of a statement. Operators **succ**, **dom** and **pdom** denote the succession, domination and post-domination relation of two statements in the CFG.

When encountering a statement that was identified as a head of a loop, function evalloop is used where the statements that form the body of the loop are taken from $\mathbb{L}$ and evaluated. During evaluation a collection of states $\mathbb{G}'$ is created and used to find those exit statements $\mathbb{E}'$ and back edges $\mathbb{Z}$ that may be executed during this iteration. If no back edge could be used during this iteration we know the loop exits, so we aggregate the states after all exit statements and finish evaluating the loop. It can also be deduced at this point that the loop will be executed at most as many times as we performed iterations. Otherwise, if we have not reached an arbitrary limit of iterations we conduct another iteration using evalloop. If the limit was reached we do not proceed but assume that this loop will continue indefinitely and set all the values that are defined within its body to unknown ω. Upon evaluation exit statements from the loop body are derived from the dead edge set of the resulting state. If there is only one exit from the loop then the loop exits in the current iteration and both the state of the variables and the number of iterations are added to $\mathbb{S}$. Otherwise another iteration is required and the evaluation is repeated. In order to manage infinite loops or those where the conditions of exiting are uncertain, an iteration limit L is given which, when reached, will cause the evaluation to cease and set all effects of the loop to unknown value ω. Setting values to ω preserves safety. We use two additional helper functions within evalloop. We define predicate unpredecessed which checks whether a statement's predecessors are all dead or the edge from them to it are unused. We also define function defs which returns the names of variables defined in a given statement.

9.1.3 Regions

The second phase of our analysis is concerned with preparing the input structure required by the third phase which is conducted using region-like structures. Thus we introduce a function to convert the CFG into a region graph. *Regions* [2, 51] are areas of code with

$$\mathbb{D} \triangleq \{s \mid \mathbb{S} = \mathbb{G}(s), \mathsf{unpredecessed}(\mathbb{S}_D, s)\}$$
$$\mathsf{block}(\mathbb{H}, [s_1, ..., s_n]) \triangleq \texttt{block}\ [R_i \mid 1 < i < n, R_i = \mathsf{regf}(\mathbb{H}, s_i) \wedge (i = 1 \vee \neg s_i \in R_{i-1})]$$
$$\mathsf{regf}(\mathbb{H}, l : b_1; ...; b_n;) \triangleq \mathsf{block}(\mathbb{H}, [l, b_1, ..., b_n])$$
$$\mathsf{regf}(\mathbb{H}, b_1; ...; b_n;) \triangleq \mathsf{block}(\mathbb{H}, [b_1, ..., b_n])$$
$$\mathsf{regf}(\mathbb{H}, s \in \mathbb{H}) \triangleq \texttt{loop}\ s, \mathsf{block}(\mathbb{H} \setminus \{s\}, [s' | s' \in \mathbb{L}(s)])$$
$$\mathsf{regf}(\mathbb{H}, s \in \bigcup_{\forall h \in \mathbb{H}} \mathbb{L}(h) \vee s \in \mathbb{D}) \triangleq \texttt{unit}$$
$$\mathsf{regf}(\mathbb{H}, s : \texttt{if}\ p\ \texttt{goto}\ l_1\ \texttt{else}\ l_2) \triangleq$$
$$\quad \text{case } \nexists e \in \mathit{Stmt}, e\ \mathbf{pdom}\ s \rightarrow$$
$$\quad\quad \texttt{condition}\ p, \mathsf{block}(\mathbb{H}, [s' \mid l_1\ \mathbf{dom}\ s']), \mathsf{block}(\mathbb{H}, [s' \mid l_2\ \mathbf{dom}\ s'])$$
$$\quad \text{case } \exists e \in \mathit{Stmt}, \nexists e' \in \mathit{Stmt}, e\ \mathbf{pdom}\ s \wedge e'\ \mathbf{pdom}\ s \wedge e\ \mathbf{psdom}\ e' \rightarrow$$
$$\quad\quad \texttt{condition}\ p, \mathsf{block}(\mathbb{H}, [s' \mid l_1\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s']),$$
$$\quad\quad\quad \mathsf{block}(\mathbb{H}, [s' \mid l_2\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s'])$$
$$\mathsf{regf}(\mathbb{H}, s : \texttt{switch}(i)\{\texttt{case}\ c_1 : l_1; ...; \texttt{case}\ c_n : l_n; \texttt{default:}\ l_0\}) \triangleq$$
$$\quad \text{case } \nexists e \in \mathit{Stmt}, e\ \mathbf{pdom}\ s \rightarrow$$
$$\quad\quad \texttt{condition}\ (i = c_1), \mathsf{block}(\mathbb{H}, [s' \mid l_1\ \mathbf{dom}\ s']), \big(, ...,$$
$$\quad\quad\quad \big(\texttt{condition}\ (i = c_n), \mathsf{block}(\mathbb{H}, [s' \mid l_n\ \mathbf{dom}\ s']), \mathsf{block}(\mathbb{H}, [s' \mid l_0\ \mathbf{dom}\ s'])\big)\big)$$
$$\quad \text{case } \exists e \in \mathit{Stmt}, \nexists e' \in \mathit{Stmt}, e\ \mathbf{pdom}\ s \wedge e'\ \mathbf{pdom}\ s \wedge e\ \mathbf{psdom}\ e' \rightarrow$$
$$\quad\quad \texttt{condition}\ (i = c_1), \mathsf{block}(\mathbb{H}, [s' \mid l_1\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s']), \big(, ...,$$
$$\quad\quad\quad \big(\texttt{condition}\ (i = c_n), \mathsf{block}(\mathbb{H}, [s' \mid l_n\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s']),$$
$$\quad\quad\quad\quad \mathsf{block}(\mathbb{H}, [s' \mid l_0\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s'])\big)\big)$$
$$\mathsf{regf}(\mathbb{H}, s : \texttt{invoke}\ i.[j(j_1, ..., j_n) : t](i_1, ..., i_n)\{b_1, ..., b_m\}) \triangleq$$
$$\quad \texttt{invoke}\ i, \mathsf{regf}(\mathbb{H}, b_1), ..., \mathsf{regf}(\mathbb{H}, b_m), s$$
$$\mathsf{regf}(\mathbb{H}, s) \triangleq \texttt{statement}\ s$$

Figure 9.6: Region-finding analysis.

a single entry point, like code blocks. We extend each region with information about its rôle in the code. We distinguish unit regions, statement regions, invocation regions, block regions, condition regions, and loop regions. We show their definitions in Fig. 9.5.

Regions are converted from Jimple CFG by the analysis defined in Fig. 9.6. The analysis is performed on the root of the CFG using regf. The function then handles each node of the CFG by recursion and returns a tree of regions. It uses the loop header set $\mathbb{H}$ and a map of loop headers to their bodies $\mathbb{L}$ from the previous analysis, and a set of dead statements $\mathbb{D}$ whose all incoming edges or predecessors are dead (according to $\mathbb{S}_D$). For convenience, we also define function block which creates a block region from a sequence of statements by applying regf to each of them in succession and aggregating them into a single region.

9.1.4 Call Count Analysis

Call count analysis is performed on the region tree in order to establish the number of times each object's methods are called. It is depicted in Fig. 9.7. The analysis begins with the application of function ccount at the root of the region tree and proceeds depth-first through the subregions. In general, method calls on objects in the tree's leafs are counted and the counts are aggregated upwards, either by adding the call counts (with addjoin) in cases of sequences or by taking the highest count (using maxjoin) in cases of alternative program paths.

Function ccount takes three arguments—the global state $\mathbb{G}$, the maximum number of executions of the parent region n, and the region of appropriate type. The function returns a map of object identifiers to the number of times that particular object's method were called. Thus, when the function comes across statement or unit regions it returns empty sets. When it reaches an invoke region it notes the object owning the method and creates a mapping of that object to the number of times the parent region is to be

$$\mathsf{ccount}(\mathbb{G}, n, \texttt{unit}\ ) \triangleq \varnothing$$
$$\mathsf{ccount}(\mathbb{G}, n, \texttt{statement}\ s) \triangleq \varnothing$$
$$\mathsf{ccount}(\mathbb{G}, n, \texttt{invoke}\ j, R_1, ..., R_m, s) \triangleq \mathbb{S} = \mathbb{G}(s),$$
$$\quad \mathsf{addjoin}(\{\mathbb{S}_V(j) \mapsto n\}, \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{G}, n, R_1), ..., \mathsf{ccount}(\mathbb{G}, n, R_m)))$$
$$\mathsf{ccount}(\mathbb{G}, n, \texttt{block}\ [R_1, ..., R_n]) \triangleq \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{S}_V, n, R_1), ..., \mathsf{ccount}(\mathbb{G}, n, R_n))$$
$$\mathsf{ccount}(\mathbb{G}, n, \texttt{condition}\ p, R_1, R_2, s) \triangleq \mathbb{S} = \mathbb{G}(s),$$
$$\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{true} \to \mathsf{ccount}(\mathbb{G}, n, R_1)$$
$$\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{false} \to \mathsf{ccount}(\mathbb{G}, n, R_2)$$
$$\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \omega \to \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{G}, n, R_1), \mathsf{ccount}(\mathbb{G}, n, R_2))$$
$$\mathsf{ccount}(\mathbb{G}, n, \texttt{loop}\ h, R) \triangleq \mathbb{S} = \mathbb{G}(h), \mathsf{ccount}(\mathbb{G}, n * \mathbb{S}_I(h), R)$$
$$\mathsf{maxjoin}(\mathbb{M}_1, ..., \mathbb{M}_n) \triangleq \{k \mapsto \max(\mathbb{M}_1(k), ..., \mathbb{M}_n(k)) \mid k \in \text{dom } \mathbb{M}_1 \cup ... \cup \text{dom } \mathbb{M}_n\}$$
$$\mathsf{addjoin}(\mathbb{M}_1, ..., \mathbb{M}_n) \triangleq \{k \mapsto \mathbb{M}_1(k) + ... + \mathbb{M}_n(k) \mid k \in \text{dom } \mathbb{M}_1 \cup ... \cup \text{dom } \mathbb{M}_n\}$$
$$\omega + c = \omega,\ \omega * c = \omega,\ \max(\omega, c) = \omega$$

Figure 9.7: Call count analysis.

executed; this mapping is then aggregated using function addjoin to the results of the evaluation of the joined bodies of the invoked method using ccount. If a block region is encountered its subregions are evaluated first and the results of these evaluations are aggregated using addjoin. When ccount encounters a conditional region the condition is checked and one of the subregions is evaluated, if the condition is true or false or both conditions are evaluated and their results are aggregated using maxjoin if the condition is unknown. Finally, with loop regions the subregion that is the loop's body is processed using ccount, but the number of executions of the parent region is multiplied by the number of loop iterations (obtained from $\mathbb{S}_I$).

Function maxjoin is used for joining the results of evaluations of two or more subregions where it is unknown which ones will execute. It takes n maps of some keys to numerical values as arguments and returns a similar map. Out of all values that share a key across the maps the maximum one is inserted into the resulting map. Function addjoin is used for aggregating the results of evaluations of a sequence of subregions that will execute one after another. It takes n maps of some keys to numerical values as arguments and returns a similar map. All values that share a key across the maps will be added together and the sum will be inserted into the resulting map under that key.

Functions at this stage of the analysis may need numerical values to be added or multiplied with the unknown value ω. If this happens, we treat it as an absorbing element, and the result of such an operation is always unknown. In a similar vein, the maximum of any set of numbers including ω is also unknown.

9.2 Implementation

We implemented our precompiler as a tool for Atomic RMI using the Soot framework. The precompiler implementation consists of three elements: Jimple creation, upper bound analysis, and code generation (as shown in Fig. 9.8). The Jimple creator converts Java source code into the Jimple intermediate language—this is provided by Soot. The upper bound analysis deduces the information about remote object calls within Jimple. It is divided into four analyses, each responsible for one pass over the code. The code generator instruments the input source code with instructions based on the information obtained by the analysis. The two components are described in more detail below.

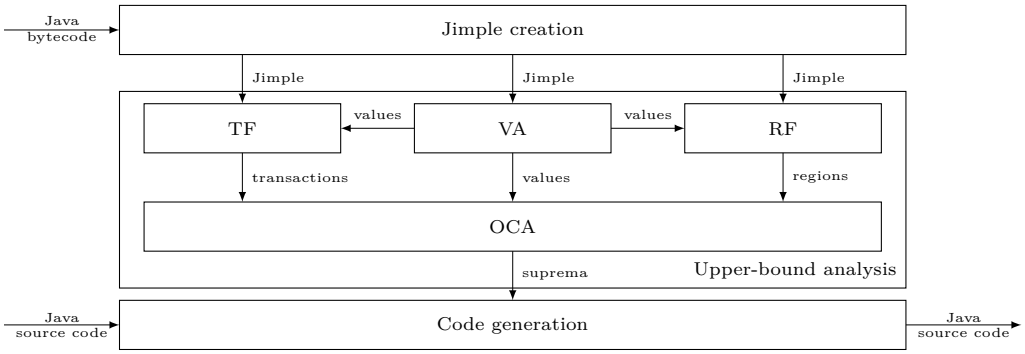


Figure 9.8: Components and information flow in the precompiler.

9.2.1 Upper Bound Analysis

The upper bound analysis consists of *value analysis* (*VA*), *region finding* (*RF*), transaction finding (*TF*), and object call analysis (*OCA*). These are forward flow analyses implemented in Soot. Each of them makes one pass over the input code in the form of a CFG from a particular starting point (the main method, for instance). The implementation of each analysis defines a transfer function applied to each node of the CFG, a join operator for joining result sets, and initial result sets (see Fig. 9.3, Fig. 9.6, and Fig. 9.7).

Value analysis is the most complex of the analyses. It is the implementation of the algorithm in Section 9.1.2 such that the transfer function and join implement eval and join. The transfer function performs whatever action is needed for a given statement type (these are recognized via the type system). The result sets represent $\mathbb{S}_V$ and $\mathbb{S}_P$, $\mathbb{S}_I$, and $\mathbb{S}_D$ are passed via separate fields (for convenience). The implementation finds loops headers and bodies using Soot's built-in loop finder. Loops are processed by running the analysis repeatedly on a pruned copy of the CFG that contains only the statements from the loop and integrating the results into the original analysis. Recurrent calls are handled by finding all applicable method bodies, starting an analysis on each, and joining the results. A stack of calls keeps track of the depth of recursion and when to bound it.

The implementation of value analysis needs to take care of additional significant mechanisms that are obvious in the formalization and therefore glossed over. These include mechanisms for evaluating expressions. Expressions' arguments' types are recognized and the semantics appropriate to them is applied (i.e. `a + b` is addition if `a` and `b` are integers or concatenation if they are strings). All combinations of basic types (at least primitives and `Object`) and operators need to be implemented. We take the approach that operators are defined by classes and perform argument-dependent operations.

Region finder converts the CFG into a region graph in accordance with the algorithm in Section 9.1.3. The algorithm performs numerous graph searches like finding domination and post-domination relations within the graph (provided by Soot) and finding if particular paths exist within the CFG (e.g. whether all paths from a conditional expression leads to the end of the body or to a common post-dominator). RF creates a region hierarchy, were each region is characterized by its type and type-specific fields.

Transaction finder is a component that tracks Atomic RMI transactions and their components: it identifies the start and possible ends of transactions, remote objects used within, and transactions' preambles. These information are collected for use by OCA and marked in Jimple using the Soot tag system.

OCA is responsible for tallying remote objects calls as in Section 9.1.4. The implementation is straightforward: it accepts the data from the preceding analyses and uses them

to traverse regions and identify those that make calls to remote objects. The number of executions of these regions is predicted and the counts are summed up with reference to particular remote objects.

The implementation must take into account the unknown values that may appear in the course of this analysis. These are implemented as a new type that allows any positive natural number or a value representing ω. The type also defines the maximum function and arithmetical operations using the unknown value (specifically addition and multiplication from Fig. 9.7).

9.2.2 Code Generation

The code generator for Atomic RMI modifies code on the lexical level using the suprema obtained from the execution of the upper bound analysis. Necessarily, in order for the code generator to modify the existing source code that source code must be available for analysis. The source is converted into tokens by the SableCC lexer [31] and then divided into lines.

The generator performs three passes over the collection of lines of tokens. In the first pass the generator locates transactions in the source code using the information provided by the transaction finding (TF) phase of code analysis. When found, all definitions in a transaction's preamble are marked for removal, with the exception of those which are followed by a comment string specifying them as manual overrides. The second pass inserts a line of code into each transaction's preamble for each identified remote object pertaining to that transaction (lines 2, 3 in Fig. 9.2a). The insert contains a variable representing the remote object and a supremum on the number of method calls to that object, and it is built using on a simple template. All the inserts are marked for prepending to the beginning of the transaction. The final pass applies all the changes marked by the previous two passes to the tokens and they can then be written to the output stream.

9.3 Discussion

Our work illustrates a static analysis for extracting the maximum number of times objects will be called in a fragment of code. Such information has a number of applications, but we concentrate on using the upper bounds as input data for Atomic RMI. We have so far found that the analysis we implemented solves this problem satisfactorily for our purposes. The tree-like region-based intermediate representation allows to find all of the method calls within the code and the use of the absorbing unknown value produces conservative results when uncertain values are involved. Both of these guarantee that the statically derived upper bounds are correct, i.e. not lower than any actual number of method calls on a particular object. Apart from being conservative, the estimated upper bounds should also be as accurate as possible—as close to the actual number of executions as possible. For typical Atomic RMI transaction code, the analysis is able to handle most scenarios adequately.

The formalization of our algorithm and adherence to it simplified the implementation of the tool. The formalization was a blueprint for the join operators and transfer function of the individual data flow analyses which it defined their *modi operandi* and allowed us to concentrate on the details of the interfaces, data structures, etc. during implementation. Another advantage is that the correctness of the created tool is verifiable, extensible, and amendable by inspection and modification of the underlying algorithm, without an initial need to delve into the actual source code.

Apart from our application of the algorithm, there are other possible applications for it. The analysis can be used to find relationships between concurrent threads: a thread accessing a shared object once, several times, or not at all may impact safety guarantees like isolation or performance in different ways. With this information prior to execution it can be treated differently, i.e., applied proper synchronization, delayed, or executed as-is without breaking guarantees. Upper bounds can also be applied in compile-time resource optimization. For instance, the amount of memory used by a given program or its influence on network traffic may be estimated from calls to particular objects if the interface is known and used to configure the environment appropriately or to optimize the analyzed program. Other uses may be found in code rewriting, automatic refactoring, etc. Apart from the work of [59] these applications seem largely unexplored.

9.4 Related Work

There is a large body of research related to analysis of programs that aims at deriving information about execution patterns statically (we sketch out some of these below). However, we do not know examples of using this information for optimizing the execution of distributed transactions in the way we do. The largest body of work to which our static analysis bears resemblance has been done with regard to the Worst Case Execution Time (WCET) problem [93]—establishing upper bounds on the time code takes to run. However, most of this work is aimed at real-time systems, not transactional concurrency control which is the main concern of our work. A number of frameworks are available for WCET analysis, like aiT [29], Bound-T [46], SWEET [35], and SymTA/P [84]. A comprehensive survey of these tools and methods was done in [94]. Whereas our approach is based on region analysis, some work in WCET use symbolic analysis [54], path analysis [37], and abstract interpretation [46, 30]. We also use the latter type of analysis for our value analysis algorithm. In WCET emphasis is placed on the problem of evaluating loops in general and bounding loop iterations in particular. This is done, among others, by the use of Presburger arithmetic [64], path analysis (using integer linear programming) [85], or a combination of methods involving abstract interpretation [27]. Our work touches on those concerns: we use loop unfolding to establish their bounds roughly similar to that of SWEET [35] but simpler. WCET tools additionally often use the Implicit Path Enumeration technique [53] or single feasible paths [103] to establish worst-case paths and perform final timing analyses. While our application presents no need for the latter, we use region-based analysis to conservatively deduce worst-case paths. WCET tools also allow for manual declaration or correction of difficult-to-deduce information (e.g., loop bounds).

Our work has significant similarities to work on lock inference. Lock inference aims to determine which memory locations or shared objects in a program must be protected by locks and where these locks should be located. Thus, our work and lock inference share the same ultimate goal of providing concurrency control via static analysis albeit by different mechanisms. The authors of [19] employ backward data flow analysis to transform a program's control flow graph into a path graph which is then used to derive locks. In [45] the authors present a method for identifying shared memory locations using type-based analysis, points-to analysis, and label flow analysis [63]. In Autolocker [57] pessimistic atomic sections are converted into lock-guarded critical sections by analyzing dependencies among annotated locks based on a computation history derived from a transformation of the code using a type system.

In [59] the authors propose a tool for the automatic inference of upper bounds on the

usage of user-specified resources. Rather than memory or execution time, these may be the number of open files, accesses to database, sent text messages, etc. This work and ours share the set of tools they use (Soot and Jimple [87]) and they both try to solve a similar problem. The tool presented by the authors performs a data flow analysis to derive data dependencies, then creates a set of equations from input-output parameter size relationships. Finally the equations are solved using a recurrence solver. Our approach differs most in that we perform region analysis to determine maximum paths and resource use where they construct and solve equations.

10 Conclusions

In order to prove the main thesis of the dissertation we surveyed the existing TM safety properties, and examined their applicability to TM with early release in Chapter 3. We then developed safety properties suitable for such TM: last-use opacity and strong last-use opacity in Chapter 5.

Then, we examined the existing pessimistic TM concurrency control algorithms, both distributed and non-distributed ones, as well as optimistic DTM and optimistic TM with early release (Chapter 4). On the basis of our analysis we selected the family of versioning algorithms as a promising basis for further research. We then extended BVA and SVA to eliminate their single point of failure by removing the dependence on a global lock, and lifted them from the commit-only model to the more general arbitrary abort model in Chapter 6 (Sections 6.1 and 6.2, respectively).

Next, we introduced OptSVA+R and OptSVA-CF+R and their variants in Chapter 6 (Sections 6.3–6.4). These are novel pessimistic TM concurrency control algorithms based on the extended versioning algorithms, but employing a number of optimizations that allow them to execute conflicting transactions with a high degree of parallelism. We then implemented these algorithms as a DTM system and showed in Chapter 8 that such DTM systems manage to outperform a state-of-the-art optimistic DTM system while maintaining a zero percent abort rate. We also showed in Chapter 7 that despite their improved generality and efficiency, the new algorithms retain strong safety guarantees. We showed this by conducting formal opacity and last-use opacity proofs which required us to introduce new proof techniques.

Finally, in Chapter 9 we introduced a precompiler to gather *a priori* knowledge about transaction executions which improves the ease with which our implementations can be used in practice.

Proof for Thesis

Below we conclude by providing a demonstration that the results presented in the dissertation bear out the thesis introduced in Chapter 1.

We introduce Atomic RMI 2, a CF DTM system implementing pessimistic DTM algorithms: OptSVA-CF+R and ROptSVA-CF+R.

a) In Section 8.2.2 we show that Atomic RMI 2 outperforms a state-of-the-art optimistic DTM system, so Atomic RMI 2 is capable of high performance.

b) We show in Theorem 10 that OptSVA-CF+R is last-use opaque, in Theorem 4 we show it is strongly progressive, and in Theorem 3 we show it is deadlock-free. Hence OptSVA-CF+R satisfies strong safety, progress, and liveness guarantees.

c) In Section 8.2.2 we show that OptSVA-CF+R does not abort in practice, so irrevocable operations execute correctly in practice. In addition, ROptSVA-CF+R completely removes the possibility of aborting from the class of reluctant transactions, so irrevocable operations always execute correctly in general, assuming all irrevocable transactions are reluctant.

d) OptSVA-CF+R supports arbitrary aborts and operates within the heterogeneous object model. In addition, it has no single point of failure to detract from scalability. Finally, the information required *a priori* can be derived via precompilation. Thus, we consider OptSVA-CF+R to apply practically.

Thus, our thesis is satisfied. □

Future Work

Even though the research presented in this dissertation achieves its aims, we see avenues for improvement that can be undertaken in future research.

Eventual Consistency

Given the limitations imposed on distributed systems that are necessary to maintain strong consistency guarantees there is a growing interest in relaxed consistency models. Such models are often sufficient for particular applications, but allow more freedom to improve scalability and availability. Eventual consistency [89] is a particularly useful approach, where the correct state spreads throughout the system over time, so that at any point any element of the system may be inconsistent, but all elements will eventually converge upon a consistent state. On the other hand relaxing properties may be unacceptable in the general case: a slightly stale shopping cart is one thing, but inconsistent bank transfer processing is quite another.

We see a promising future research direction in attempting to balance strong and eventual consistency by proposing a general-purpose transactional memory (based on the solutions presented in this dissertation) that allows eventually consistent transactions to run alongside consistent ones. Specifically, we propose to extend the versioning algorithms with a mechanism that allows certain *eventually consistent* transactions to execute quickly, without waiting for currently running transactions. When they commence, such transactions would grab the most recent consistent snapshot of all the variables they need of those snapshots that can be obtained without waiting. Once the snapshot is buffered, these transactions operate only on the buffers, to avoid waiting during reads and invalidating the global state on writes. Thus, this mode relaxes safety—the client may initially see an inconsistent view (although one generated using read-consistent data) and, since his updates are not propagated, has a different impression of the global state. Thus, the state must eventually be converged, and so, the transaction is concurrently re-executed in consistent mode to fix the client's view and apply modifications. Note that other clients only see the execution of the consistent transaction. We presented some preliminary ideas with respect to eventually consistent versioning algorithms in [101].

Benchmarks

Initially, TMs were evaluated using microbenchmarks, but these test specific features in isolation and use data structures that are too trivial to draw general conclusions about a

TM. Alternatively, there are HPC benchmark suites, but these are difficult to translate meaningfully into the transactional idiom. That is, benchmarks from SPEComp [4] or SPLASH-2 [104] are already expertly optimized to minimize synchronization, so any incorporated transactions are used rarely and have little effect on overall performance. Hence, a set of TM-specific benchmarks was needed, whose transactional characteristics and contention for shared resources were both varied and controllable. Thus, benchmarks and benchmark suites like STMBench7 [34], LeeTM [3], and STAMP [58] were developed.

As with non-distributed TMs, the variety of differently-featured distributed TMs require empirical evaluation to find how their features, the workloads, and the configuration of distributed systems influences their performance. Therefore, as with non-distributed TMs, they must be evaluated empirically. However, the existing TM benchmarks are not appropriate for distributed TMs. This is primarily the case since the structures they use are not easy to distribute. Distributing non-distributed TM benchmarks often leads to arbitrary sharding of the structure that has no purpose for the application itself (e.g., the clients still must access the entire domain). Hence distributing STMBench7, LeeTM, or *labyrinth* or *k-means* from STAMP creates applications that do not reflect realistic use cases for distributed systems. On the other hand, even if a benchmark has valid distributed variants, the conversion is often non-trivial and should not be expected to be done *ad hoc*, if it is to be uniformly applied by various research teams.

As a result systems like HyFlow, HyFlow2, Atomic RMI, and Atomic RMI 2 are all evaluated using a few microbenchmarks (usually in-house implementations) supplemented by a distributed version of the *vacation* benchmark from STAMP, which originally mimics a distributed database use case. In effect, the presentation of the evaluation sometimes leads ambiguities. In addition, a direct comparison between evaluation results from different papers is often difficult, if not impossible.

Hence, we believe that a suite of dedicated benchmarks for distributed TM systems should be created. The benchmarks in the suite should allow depth of evaluation, but the design of such a suite should emulate the breadth provided by STAMP by implementing a number of distributed applications grounded in real systems. We presented some of the preliminary work on that front in [11].

Streszczenie

Programowanie współbieżne jest powszechnie uznawane za trudne (zob. np. [21, 39, 40, 68]). Źródłem trudności jest współbieżne wykonanie programu, które może potencjalnie prowadzić do przeplotu operacji wykonywanych przez wątki lub procesy na zmiennych współdzielonych, dając w efekcie nieprawidłowe wyniki. Przykładem jest błędne odczytanie przez proces innej wartości zmiennej współdzielonej, niż ta ostatnio zapisana do tej zmiennej przez ten proces. Dlatego więc programista musi przewidzieć i wyeliminować tego typu nieprawidłowe zachowanie, synchronizując wykonanie niektórych operacji. W tym celu programista ma do dyspozycji odpowiednie niskopoziomowe konstrukcje synchronizacyjne, np. zamki (ang. *locks*), monitory (ang. *monitors*), bariery (ang. *barriers*), czy semafory (ang. *semaphores*). Jednakże korzystanie z tych mechanizmów w sposób poprawny i efektywny nie jest łatwe, gdyż wymaga wnikliwej analizy całego systemu, a błędy wynikające z niewłaściwego zastosowania synchronizacji są często trudne do wykrycia z uwagi na niedeterminizm. Błędna synchronizacja ma poważne skutki dla działania systemu, np. odczyt niespójnego stanu (ang. *inconsistent views*), zakleszczenie (ang.*deadlock*, *livelock*), hazard (ang. *race condition*), lub inwersja priorytetów (ang. *priority inversion*).

Programowanie współbieżne jest jednak nieuniknione. Wynika to ze wzrastającej popularności procesorów wielordzeniowych, gdzie współbieżne wykonanie programu jest niezbędne, aby wykorzystać potencjał wielu rdzeni procesora. Ponadto, wraz z popularyzacją architektur zorientowanych na usługi (ang. *service oriented architectures*) oraz przetwarzania w chmurze (ang. *cloud computing*), także systemy rozproszone, które są z natury współbieżne, stały się wszechobecne. Jest to widoczne do tego stopnia, że rozmaite aplikacje, począwszy od edycji dokumentów tekstowych, a kończąc na bazach danych i aplikacjach typu *Big Data*, coraz częściej delegują przetwarzanie do usług zdalnych, które wykonują określony program równolegle z programem klienta.

Skoro programiści aplikacji coraz częściej stykają się z problemami programowania współbieżnego, niezbędnym staje się dostarczenie im odpowiednich abstrakcji, które pozwalałyby wyeliminować część z tych problemów oraz ukryć szczegóły implementacji mechanizmów synchronizacji. Abstrakcje takie są wykorzystywane w innych dziedzinach programowania. Przykładowo, programiści nie piszą w praktyce własnych rozwiązań do komunikacji przez sieć, lecz korzystają z hermetycznych bibliotek (np. Netty, JGroups, Java Message Service lub Java RMI), które dostarczają tego typu funkcjonalność w formie wysokopoziomowego API, które zwalnia programistę od implementowania szczegółów zarządzania gniazdami czy serializacji danych. W podobny sposób programiści aplikacji

powinni móc tworzyć systemy współbieżne.

Pamięć Transakcyjna

Pamięć transakcyjna (ang. *transactional memory*, *TM*) [44, 71] jest zapożyczoną z systemów bazodanowych uniwersalną propozycją rozwiązania problemu synchronizacji w systemach współbieżnych poprzez zastosowanie abstrakcji transakcji (zob. np. [12, 16, 91]). W tym podejściu programista jedynie oznacza fragmenty kodu wymagające synchronizacji jako transakcje, a a system pamięci transakcyjnej jest odpowiedzialny za wykonanie poszczególnych operacji na danych współdzielonych w ramach transakcji w taki sposób, żeby zapewniona była wydajność i poprawność. Wykonanie kodu transakcji zostaje powierzone odpowiedniemu algorytmowi sterowania współbieżnością, który zapewnia, że przeplot operacji spełnia konkretne gwarancje poprawności. Gwarancje te określone są przez własności bezpieczeństwa które opisują dany algorytm, np. uszeregowalność (ang. *serializability*) [60] lub nieprzezroczystość (ang. *opacity*) [33]. Programista, z kolei, nie musi znać szczegółów działania zastosowanego algorytmu sterowania współbieżnością, a jedynie własności bezpieczeństwa, które ten algorytm spełnia. W konsekwencji, abstrakcja transakcji ukrywająca szczegóły implementacji algorytmów sterowania współbieżnością oraz spełniająca określone własności bezpieczeństwa, znacząco upraszcza programowanie, wspierając jednocześnie poprawność i wydajność działania systemów współbieżnych.

Rozproszona pamięć transakcyjna (ang. *distributed transactional memory*, *DTM*) [14, 49, 18, 68, 86, 10] przenosi ideę pamięci transakcyjnej do systemów rozproszonych. Generalizacja ta powoduje potrzebę rozwiązania dodatkowych problemów jak asynchroniczność i awarie częściowe, ale także stwarza nowe perspektywy. Cechą najbardziej odróżniającą transakcje w pamięci transakcyjnej od ich bazodanowych poprzedników jest możliwość wykonywania innych operacji niż tylko odczyt i zapis na zmiennych współdzielonych. W systemach pamięci transakcyjnej współdzielone między transakcjami mogą być obiekty, których operacje odczytu i zapisu mają bardziej złożoną semantykę, np. liczniki, czy kolejki. Mogą to być także obiekty których interfejsy są dowolnie zdefiniowane przez programistę i których implementacja ma arbitralną semantykę. W rozproszonej pamięci transakcyjnej dowolność definicji obiektów może dodatkowo służyć umiejscowieniu wykonania pewnego kodu na konkretnych (zdalnych) węzłach sieci. W szczególności wyróżnia się dwa modele. Model przepływu danych (ang. *data flow*) zakłada, że obiekt na którym wykonywana jest operacja migruje do węzła, na którym wykonywana jest transakcja, i właśnie tam wykonywany jest także kod operacji. W tym modelu efekty operacji są zawsze lokalne względem transakcji. Model przepływu sterowania (ang. *control flow*) zakłada, że obiekty są nieruchome i kod operacji wykonuje się zawsze na węźle „domowym". Oznacza to, że efekty operacji są lokalne względem obiektu, a nie transakcji. Każdy z modeli ma swoje wady i zalety, ale unikatową cechą modelu przepływu sterowania jest to, że pozwala on na „pożyczanie" przez transakcje mocy obliczeniowej od zdalnych węzłów. Pozwala to na większą elastyczność przy projektowaniu i implementacji aplikacji rozproszonych.

Optymistyczne Sterowanie Współbieżnością

Powszechnie stosowanym podejściem do synchronizacji w (rozproszonej) pamięci transakcyjnej jest podejście optymistyczne. W podejściu tym, w ujęciu ogólnym, transakcje są wykonywane jednocześnie bez względu na charakterystykę dostępów wewnątrz transakcji, a próba walidacji ich poprawności następuje później, np. w momencie zatwierdzenia (ang. *commit*) gdy wszystkie operacje transakcji zostały wykonane. Zatwierdzenie kończy się powodzeniem, gdy wykonanie transakcji przebiegło w sposób poprawny. Niepowodzenie

zatwierdzenia następuje w wypadku gdy bieżąca transakcja wykonała nieprawidłowe operacje, np. usiłując operować na tym samym obszarze pamięci (zmiennej współdzielonej, obiekcie) jednocześnie z inną transakcją w sytuacji, gdy przynajmniej jedna z nich próbuje ten obszar modyfikować. Scenariusz taki nazywany jest konfliktem (ang. *conflict*) i wymaga on wycofania (ang. *abort*) jednej z transakcji. Wycofanie transakcji oznacza, że transakcja usuwa wszelkie oznaki swojego wykonania. Następnie transakcja wycofana zostanie wykonana po raz kolejny, w nadziei, że tym razem konflikt nie wystąpi i transakcja zostanie zatwierdzona poprawnie. Podejście optymistyczne można zaimplementować na wiele sposobów, ale typowym jest buforowanie operacji lub ich wyników podczas działania transakcji, i wprowadzanie zmian do oryginalnego obiektu dopiero podczas zatwierdzenia (ang. *commit-time*), raczej niż modyfikowanie obiektów już podczas wykonywania operacji (ang. *encounter-time*). Typowe jest wykrywanie konfliktów jak najwcześniej, w celu minimalizacji marnotrawienia pracy wykonanej przez ostatecznie wycofane transakcje.

Podejście optymistyczne jest dość uniwersalnym rozwiązaniem, ale ma ono dwa mankamenty. Po pierwsze, podejście optymistyczne napotyka na problemy wynikające ze spekulacyjnego wykonywania transakcji w środowiskach z wysokim współczynnikiem współzawodnictwa (ang. *contention*)—tj. takich, gdzie wiele transakcji jednocześnie ubiega się o wykonanie operacji na tym samym obiekcie. Wysokie współzawodnictwo sprawia, że prawdopodobieństwo wystąpienia konfliktów wzrasta, co z kolei powoduje, że wzrasta częstość z jaką transakcje są wycofywane. W efekcie zwiększone jest prawdopodobieństwo, że dana transakcja będzie wielokrotnie wycofana i wielokrotnie (co najmniej częściowo) wykonana, zanim w końcu zostanie zatwierdzona. Co więcej, wykonywanych jest wiele obliczeń, których wyniki są następnie ignorowane, a po odjęciu transakcji wycofanych, konfliktujące się transakcje w praktyce wykonują się sekwencyjnie. Istnieją mechanizmy, które eliminują wyżej opisany problem problem przez zarządzanie ponownym uruchamianiem konfliktujących transakcji, tak, aby nie doprowadzać do ponownych konfliktów. W tym celu można stosować proste rozwiązania jak wykładnicze opóźnienie (ang. *exponential back-off*) [43], lub rozwiązania złożone, jak szeregowanie oparte o prawdopodobieństwo wystąpienia konfliktów [24, 105], a także inne mechanizmy sterowania współzawodnictwem (ang. *contention management*) [25, 70]. Rozwiązania te mają opóźniać wykonanie niektórych transakcji (zazwyczaj po pierwszym konflikcie), powodując obniżenie liczby współbieżnych transakcji. Rozwiązania te natomiast wymagają parametryzacji, co wymaga strojenia parametrów ręcznie lub wyprowadzania ich podczas działania systemu, a także prowadzi do konieczności reakcji na zmiany profilu obciążenia. Często także rozwiązania te wymagają centralnej koordynacji, co uniemożliwia wykorzystanie ich w systemach rozproszonych.

Kolejnym problemem podejścia optymistycznego jest obsługa operacji niewycofywalnych (ang. *irrevocable operations*). Operacje niewycofywalne są to operacje, których efekty są obserwowalne, ale nie można ich usunąć ani nie powinno się powielać. Przykładami takich operacji są operacje na zamkach, operacje wejścia/wyjścia, czy komunikacja sieciowa. Operacje te często występują w złożonych aplikacjach i są zazwyczaj trudne do zlokalizowania w kodzie transakcji, ponieważ są wykonywane w ramach procedur będących częściami używanych przez programistę bibliotek. Natomiast jeśli operacja niewycofywalna znajdzie się w kodzie transakcji wykonywanej optymistycznie, to ta transakcja może spowodować wielokrotne wykonanie operacji niewycofywalnej na skutek konfliktu. Przykładowo, spowodować to może wielokrotne wysłanie tej samej wiadomości sieciowej, łamiąc protokół komunikacji, lub wielokrotne pobranie tego samego zamka prowadząc do zakleszczenia transakcji. Rozwiązanie tego problemu w systemach optymistycznych nie jest proste. Popularnym rozwiązaniem jest spowodowanie, że transakcja wykonująca operacje niewycofywalne nie zostaje nigdy wycofana. Przykładowo, w [92] zaproponowano system, w którym transakcja zawierająca operacje niewycofywalne staje się transakcją

niewycofywalną, która zawsze wygrywa konflikty z innymi transakcjami. Jednakże tylko jedna taka transakcja może działać jednocześnie w całym systemie ze względu na paradoks w wypadku konfliktu między dwoma niewycofywalnymi transakcjami. Powoduje to jednak zmniejszenie wydajności systemu. W [9, 62] zastosowane inne rozwiązanie: system pamięci transakcyjnej utrzymuje wiele wersji tego samego obiektu, więc transakcje mogą operować na starszych wersjach jeśli operowanie na nowej wersji powodowałoby konflikt. Rozwiązanie to prowadzi do skomplikowania i spowolnienia systemu pamięci transakcyjnej. W konsekwencji, wiele istniejących systemów pamięci transakcyjnej zabrania wykonywania operacji niewycofywalnych (np. Haskell [41]) lub wymaga zdefiniowania operacji, które wykonają kompensacji ich efektów (np. [13]). Są to jednak rozwiązania niepraktyczne, zwłaszcza jeśli system rozproszonej pamięci transakcyjnej ma być zaaplikowany w złożonych systemach zorientowanych na usługi. Przykładowo, jeśli wykonanie operacji na obiekcie-usłudze powoduje efekt materialny (np. wydruk książki), to operacja niewycofywalna jest częścią semantyki usługi i nie istnieje kompensacja, która (bezstratnie) odwróci jej efekt.

Pesymistyczne Sterowanie Współbieżnością

Prostszą metodą rozwiązywania problemów wymienionych powyżej jest użycie pesymistycznego podejścia do sterowania współbieżnością. Podejście pesymistyczne ma swoje początki w transakcjach bazodanowych (np. blokowanie dwufazowe – ang. *two-phase locking* [12, 91]) i zostało przeniesione do pamięci transakcyjnej w [56, 1, 10] oraz w pracach [96, 97]. Ogólna idea pesymistycznej pamięci transakcyjnej jest taka, że transakcje nie wykonują operacji spekulacyjnie, lecz najpierw sprawdzany jest warunek poprawności wykonania danej operacji, a operacje dla których warunek nie może być natychmiast spełniony ze względu na potencjalny konflikt są opóźniane do momentu, aż konflikt staje się niemożliwy. Oznacza to, że transakcje są wycofywane bardzo rzadko lub wcale, dzięki czemu wyeliminowane zostają problemy związane wyżej wymienionymi scenariuszami z wysokim współzawodnictwem i z operacjami niewycofywalnymi.

W [56] pokazano jednakże, że podejście pesymistyczne w formie stosowanej do tej pory ma negatywny wpływ na wydajność systemów pamięci transakcyjnej, ponieważ jest uzależnione od szeregowego wykonywania transakcji, które wykonują zapisy do zmiennych. Ograniczenie to ma na celu wykluczenia konfliktów, ale powoduje ono ograniczenie równoległości wykonania.

Głównym celem przedstawionej pracy było pokazanie, że obniżenie wydajności nie jest nieodzowną częścią podejścia pesymistycznego i może być całkowicie wyeliminowane. W tym celu, rozważono zastosowanie techniki wczesnego zwalniania zasobów (ang. *early release*). Wczesne zwalnianie zasobów to technika optymalizacyjna stosowana w pamięci transakcyjnej, gdzie pary transakcji między którymi zachodzi konflikt są jednak zatwierdzane [65], jeśli tylko przeplot operacji, który prowadzi do konfliktu jest w istocie poprawny. Technika ta jest szczególnie efektywna w podejściu pesymistycznym, gdzie transakcje nie są wycofywane. Przy takim założeniu, można zezwolić transakcjom na odczytywanie ostatecznego stanu zmiennych modyfikowanych przez inną transakcję bez troski o odczytanie stanu niepoprawnego, pomimo tego, że modyfikująca transakcja nie została jeszcze zatwierdzona. Systemy korzystające z techniki wczesnego zwalniania (np. [43, 65, 28, 13, 75]) pokazują, że powoduje ona znaczącą poprawę efektywności działania pamięci transakcyjnej. Dlatego też w niniejszej pracy wykorzystano tę technikę jako rdzeń zaproponowanych optymalizacji, dążąc do stworzenia bezpiecznego i wydajnego systemu pesymistycznej pamięci transakcyjnej.

Bezpieczeństwo

W odróżnieniu od transakcji bazodanowych, pamięć transakcyjna pozwala na włączanie dowolnych operacji do kodu transakcji obok odczytów i zapisów na danych współdzielonych. Powoduje to, że pamięć transakcyjna musi wykonywać transakcje bardziej ostrożnie, niż jest to wymagane w bazach danych. Przykładowo, uszeregowalność ustanawia, że jeśli transakcje, które są zatwierdzone, wykonały się poprawnie, to całość wykonania może być uznana za poprawną. Jeśli więc transakcja bazodanowa przeczyta ze zmiennej niespójną wartość, wystarczy, że nie dopuści się tej transakcji do zatwierdzenia, i cały przeplot pozostanie poprawny w świetle uszeregowalności. Natomiast, jeśli pamięć transakcyjna pozwoli transakcji na odczytanie wartości niespójnej, może dojść do złamania jakiegoś niezmiennika i wykonania nieprzewidzianej niebezpiecznej operacji, np. podziału przez zero lub wejścia w pętlę nieskończoną. W takim wypadku, klasyczne własności bazodanowe, takie jak uszeregowalność są niewystarczające dla zapewniania poprawności pamięci transakcyjnej. Pamięć transakcyjna wymaga własności bezpieczeństwa, które będą ograniczać lub wykluczać możliwość odczytu niespójnego stanu przez transakcje niezależnie od tego, czy będą one ostatecznie zatwierdzone. W tym celu zaproponowano własność nieprzezroczystości, która, ponad wymagania uszeregowalności, wymaga także utrzymania między transakcjami porządku czasu rzeczywistego i uniemożliwia transakcjom czytanie modyfikacji wprowadzonych przez żywe (jeszcze niezatwierdzone) transakcje. Nieprzezroczystość stała się standardową własnością systemów pamięci transakcyjnej i jest *de facto* spełniana przez większość systemów prezentowanych w literaturze.

Jednak, jeśli wykluczyć czytanie modyfikacji wprowadzonych przez niezatwierdzone transakcje, niemożliwym staje się użycie techniki wczesnego zwalniania zasobów, nawet jeśli nie powoduje to niepoprawnych zachowań. Natomiast przed zaproponowaniem wydajnej pamięci transakcyjnej, niezbędnym staje się zdefiniowanie gwarancji poprawności które przez taki system muszą być spełnione i odkrycie lub zdefiniowanie własności bezpieczeństwa które będą te gwarancje określać, jednocześnie dopuszczając użycie wczesnego zwalniania zasobów. Ze względu na fakt, że nieprzezroczystość jest bardzo restrykcyjną własnością, w celach praktycznych w literaturze zaproponowano wiele innych, rozluźnionych własności bezpieczeństwa, takich jak spójność świata wirtualnego (ang. *virtual world consistency – VWC*) [48], specyfikacja pamięci transakcyjnej (ang. *transactional memory specification — TMS1* i *TMS2*) [22], nieprzezroczystość elastyczna (ang. *elastic opacity*) [28], nieprzezroczystość żywa (ang. *live opacity*) [26] i inne. W ramach tej rozprawy dokonano analizy tych własności, a także istniejących własności bazodanowych celem określenia czy pozwalają na optymalizację przez wczesne zwalnianie zasobów, jakie ograniczenia nakładają na tę optymalizację i jakich wymagają dodatkowych założeń. Na podstawie tej analizy wprowadzono nowe własności przeznaczone dla systemów pamięci transakcyjnej z wczesnym zwalnianiem zasobów, które jednocześnie dostarczają silnych gwarancji bezpieczeństwa.

Model Systemu

Pamięć transakcyjna może być używana w ramach wielu modelów systemu, które wpływają na założenia, jakie algorytm sterowania współbieżnością będzie przyjmować. Po pierwsze, pamięć transakcyjna może działać na zmiennych (obiektach-zmiennych), czyli na obiektach, których stan jest zdefiniowany przez pojedynczą wartość, która z kolei może być odczytana lub nadpisana. Model taki jest typowy dla nierozproszonej pamięci transakcyjnej (np. [21, 39, 65]), ale rozproszona pamięć transakcyjna częściej używa modelu gdzie obiekty współdzielone są bardziej złożone (np. [68, 86]). Konkretnie, wyróżniamy tutaj dwa modele obiektowe: jednorodny (ang. *homogeneus*) i niejednorodny (ang. *hetero-*

geneous). W modelu jednorodnym obiekty są jednakowe i relatywnie proste: odpowiadają strukturom takim jak liczniki czy stosy. Obiekty współdzielą jeden interfejs, który zawiera jedną operację odczytu i jedną operacje zapisu o znanej semantyce. W modelu niejednorodnym zakłada się, że każdy z obiektów definiuje własny interfejs zawierający dowolne operacje o dowolnej (i zazwyczaj nieznanej *a priori*) semantyce, działające na złożonym, hermetycznie odizolowanym stanie. Różne modele mają różne aplikacje: zmienne znajdują zastosowanie w systemach lokalnych i równoległych o dużej wydajności oraz bazach danych, podczas gdy modele obiektowe używane są w złożonych systemach chmurowych czy architekturach zorientowanych na usługi, gdzie każdy z obiektów wyrażać może nawet całe usługi.

Po drugie, systemy pamięci transakcyjnej mogą dostarczać różnych interfejsów dla samych transakcji. Wiele optymistycznych i pesymistycznych systemów jest systemami wyłącznie zatwierdzającymi (ang. *commit-only*), gdzie transakcja dąży zawsze do tego, żeby po wykonaniu swoich operacji wykonać zatwierdzenie (np. [96, 6, 56]). Alternatywnie, pamięć transakcyjna może być systemem z dowolnością wycofania (ang. *arbitrary abort*), gdzie transakcja może w dowolnym momencie działania samoistnie wycofać się, zamiast podejmować próby zatwierdzenia. Dodanie operacji wycofania do interfejsu transakcyjnego powoduje, że system staje się bardziej ekspresywny, a także dostarcza istotnej dla wydajnej implementacji odporności na awarie częściowe funkcjonalności.

Warto odnotować, że systemy pamięci transakcyjnej działające na zmiennych mogą przyjąć dużo szersze założenia odnośnie do stanu systemu, niż pamięć transakcyjna działająca na obiektach w pozostałych modelach. Powoduje to, że jeśli porównamy wydajność takich dwóch systemów stosując tylko zmienne, pamięć transakcyjna działająca na zmiennych będzie bardziej wydajna od pamięci obiektowej. Natomiast, jeśli porównamy te dwa systemy używając modelu obiektowego, pamięć transakcyjna działająca na zmiennych ma szanse działać nieprawidłowo ze względu na jej zbyt silne założenia. Podobnie, pamięć transakcyjna przystosowana do dowolnych wycofań może być użyta w modelu wyłącznie zatwierdzającym, chociaż skutkować to będzie obniżeniem wydajności w porównaniu do odpowiednika przystosowanego do pracy w modelu wyłącznie zatwierdzającym. Z drugiej strony, pamięć transakcyjna wyłącznie zatwierdzająca nie może być użyta poprawnie i wydajnie w modelu, w którym transakcje mogą być dowolnie wycofywane. Stypulujemy, że praktyczność systemu pamięci transakcyjnej jest uwarunkowana możliwością jego aplikacji w szerokiej gamie modeli systemów przy zachowaniu wydajności i poprawności. W konsekwencji rozprawa rozważy aplikację wprowadzonych algorytmów w różnych modelach, starając się osiągnąć uniwersalność. Jednocześnie przedstawiamy warianty naszych algorytmów mające na celu poprawę wydajności w konkretnych modelach.

Przyjęto też założenie, że praktyczny system rozproszonej pamięci transakcyjnej nie może opierać się na centralnych strukturach, które wprowadzałyby pojedynczy punkt awarii (ang. *single point of failure*), gdyż ograniczałoby to skalowalność (ang. *scalability*) systemu, oraz uniemożliwiałoby jego funkcjonowanie pomimo częściowych awarii.

Żywotność i Postęp

Dodatkowo, poza poprawnością systemu, praktyczny system pamięci transakcyjnej powinien także gwarantować, że poszczególne operacje wewnątrz transakcji zostaną wykonane, tj. żywotność (ang. *liveness*), oraz, że każda z transakcji ostatecznie będzie zatwierdzona, tj. postęp (ang. *progress*). Podstawową własności żywotności dla pamięci transakcyjnej jest wolność od zakleszczenia (ang. *deadlock freedom*), która oznacza, że transakcje nigdy nie doprowadzają do zakleszczeń. Zakleszczenie to sytuacja, w której między transakcjami występuje cykl oczekiwania. Silna progresywność (ang. *strong progressiveness*) [33]

to popularna własność postępu mówiąca, że konflikt nie może doprowadzić do sytuacji, w której wszystkie skonfliktowane transakcje zostają zmuszone do wycofania. System pamięci transakcyjnej który nie spełnia tych własności żywotności i postępu może doprowadzać do "klinowania się" całego systemu, a więc nie jest systemem praktycznym.

Teza

W świetle powyższych zamierzeń i wymogów, sformułowano w pracy następującą tezę:

Możliwym jest zaproponowanie pesymistycznego algorytmu sterowania współbieżnością dla rozproszonej pamięci transakcyjnej, który jednocześnie:

a) osiąga wysoką wydajność,

b) spełnia silne własności bezpieczeństwa, żywotności i postępu,

c) gwarantuje poprawne wykonanie operacji niewycofywalnych,

d) jest stosowalny w ogólnych modelach systemów.

Kontrybucja

Przedstawiona teza jest udowodniona poprzez kontrybucje wprowadzone poniżej i opisane szczegółowo w poszczególnych rozdziałach rozprawy.

I **Analiza istniejących własności i algorytmów pamięci transakcyjnych.**
Formalnie przeanalizowano istniejące własności bezpieczeństwa dla pamięci transakcyjnej oraz bazodanowe warunki spójności celem określenia, czy znajdują one zastosowanie w pamięci transakcyjnej z wczesnym zwalnianiem zasobów. W szczególności, określono, czy pozwalają one na wczesne zwalnianie zasobów, jakie klasy niespójnych odczytów są przez nie dopuszczane i jakie ograniczenia są przez nie nałożone na transakcje. Następnie, zbadano wybrane istniejące pesymistyczne i rozproszone pamięci transakcyjne, oraz pamięci transakcyjne stosujące wczesne zwalnianie zasobów, ustalając ich parametry i gwarancje bezpieczeństwa. Pozwala nam to wyciągnąć wnioski o stosowalności istniejących własności do systemów z wczesnym zwalnianiem zasobów. Ponadto, analiza pozwoliła na ustalenie, które algorytmy i techniki mogą być użyte do implementacji pesymistycznej rozproszonej pamięci transakcyjnej. Analizy przedstawione są w Rozdziałach 3 i 4, i stanowią rozszerzenie wyników zaprezentowanych w [77] i [79].

II **Nowe silne własności bezpieczeństwa dla pamięci transakcyjnej z wczesnym zwalnianiem zasobów.**
W pracy zaproponowano zostały dwie nowe własności bezpieczeństwa, *nieprzezroczystość do ostatniego użycia* i *silna nieprzezroczystość do ostatniego użycia*, które dają silne gwarancje spójności i wykluczają większość klas niespójnych odczytów stanu, jednocześnie pozwalając na wczesne zwalnianie zasobów. Własności wraz z ich charakterystyką przedstawiono oraz omówiono w Rozdziale 5. Własności te zostały wcześniej zaprezentowane w [76, 79].

III **Nowe pesymistyczne algorytmy sterowania współbieżnością dla pesymistycznej (rozproszonej) pamięci transakcyjnej.**
Ponadto w pracy opisano szereg nowych algorytmów sterowania współbieżnością dla pesymistycznej (rozproszonej) pamięci transakcyjnej. Rozpoczęto od rozszerzenia istniejących algorytmów wersjonowania [96, 97] celem wykluczenia pojedynczego punktu awarii i uogólnienia ich do modelu pozwalającego na dowolne wycofywanie transakcji, w efekcie uzyskując algorytmy *BVA+R*, *SVA+R* i *RSVA+R*.

Następnie zastosowano szereg daleko idących optymalizacji ze względu na typy wykonywanych operacji, aby uzyskać algorytmy sterowania współbieżnością OptSVA+R i OptSVA-CF+R (wraz z wariantami), które dążą do wysokiego stopnia zrównoleglenia transakcji. Pokazujemy, że owe algorytmy pozwalają na większe zrównoleglenie wykonań transakcji niż ich poprzednicy i wykazujemy ich własności. Algorytmy wprowadzone są w Rozdziale 6, a ich dowody poprawności znajdują się w Rozdziale 7. Nowe algorytmy stanowią rozszerzenie badań zaprezentowanych w [74, 75, 78, 82, 102].

IV **Dowody bezpieczeństwa i techniki dowodzenia.**
Przedstawiono zostały także techniki dowodzenia pozwalające nam na wnioskowanie o bezpieczeństwie (nieprzezroczystość i nieprzezroczystość do ostatniego użycia) algorytmów z wczesnym zwalnianiem zasobów. Następnie zastosowano wprowadzone techniki, udowadniając własności wybranych algorytmów. Zaprezentowane jest to w Rozdziale 7 i stanowi rozszerzenie wyników pokazanych w [79, 80, 102].

V **Implementacja nowych algorytmów.** Efektem pracy były też implementacje systemów rozproszonej pamięci transakcyjnej w modelu przepływu sterowania dla dwóch z zaprezentowanych algorytmów sterowania współbieżnością. Jedna z implementacji posłużyła do pokazania, że OptSVA-CF+R przewyższa efektywnością wysokiej klasy optymistyczny system rozproszonej pamięci transakcyjnej. Implementacje i ewaluacja są zaprezentowane w Rozdziale 8 i odzwierciedlają one wyniki badań w [75, 78, 82].

VI **Analiza statyczna i prekompilator.** Wprowadzono też prekompilator, który jest w stanie wygenerować informacje wymagane *a priori* przez niektóre z zaprezentowanych algorytmów pamięci transakcyjnej na podstawie analizy statycznej kodu źródłowego transakcji. Prekompilator zaprezentowany jest w Rozdziale 9 i odzwierciedla badania zaprezentowane w [72, 73].

Analiza Istniejących Własności

W celu analizy istniejących własności bezpieczeństwa, zarówno dla pamięci transakcyjnej, jak i warunków spójności dla baz danych, zdefiniowano formalnie wczesne zwalnianie zasobów jako scenariusz, gdzie wykonywane są przynajmniej dwie transakcje T_i i T_j w taki sposób, że transakcja T_i zapisuje do jakiejś zmiennej x wartość v, a następnie T_j odczytuje v z x , gdy T_i jest żywa. Mówimy wtedy, że T_i zwalnia x do T_j. Następnie zdefiniowano trzy warunki, które określają do jakiego stopnia jakaś własność $\mathfrak{P}$ pozwala na wczesne zwalnianie zasobów:

Def. 4 *(Dopuszczanie Wczesnego Zwalniania Zasobów)* Własność $\mathfrak{P}$ dopuszcza taką historię wykonania, gdzie występuje wczesne zwalnianie zasobów.

Def. 5 *(Dopuszczanie Nadpisywania)* Własność $\mathfrak{P}$ dopuszcza taką historię wykonania, gdzie występuje wczesne zwalnianie zasobów i gdzie transakcja T_i, która zapisuje wartość v do zmiennej x i zwalnia zmienną x do transakcji T_j, następnie zapisuje ponownie jakąś wartość v' do x po tym, jak T_j odczyta v z x.

Def. 6 *(Dopuszczanie Wycofywania Po Zwolnieniu Zasobów)* Własność $\mathfrak{P}$ dopuszcza taką historię wykonania, gdzie występuje wczesne zwalnianie zasobów i gdzie transakcja T_i, zwalnia jakaś zmienną do transakcji T_j, a następnie T_i ostatecznie zostaje wycofana.

Własność	Zastosowanie	Def. 4	Def. 5	Def. 6	⊆Uszeregowalne
Serializability	bazy danych, TM	✓	✓	✓	✓
CO	bazy danych	✓	✓	✓	×
Recoverability	bazy danych	✓	✓	✓	×
Cascadelessness	bazy danych	×	×	×	×
Strictness	bazy danych	×	×	×	✓
Rigorousness	bazy danych	×	×	×	✓
Opacity	TM	×	×	×	✓
Markability	TM	×	×	×	✓
TMS1	TM	×	×	×	✓
TMS2	TM	×	×	×	✓
VWC	TM	✓	×	×	✓
Live opacity	TM	✓	×	×	✓
Elastic opacity	TM	✓	×	×	×

Tablica 11.1: Podsumowanie dopuszczalności wczesnego zwalniania zasobów w istniejących własnościach.

Spośród tych definicji, Def. 4 jest warunkiem koniecznym stosowalności własności dla pamięci transakcyjnej z wczesnym zwalnianiem. Def. 5 służy wykluczeniu własności zbyt permisywnych, które pozwalają na czytanie stanu niespójnego w trakcie działania transakcji. Odczyt takiego stanu może prowadzić do niebezpiecznych sytuacji opisanych w [33]: błędów arytmetyki obiektowej, wejścia w nieskończoną pętlę, dzielenia przez zero. Def. 6 służy zapewnieniu, że nie ogranicza się możliwości wycofania transakcji, które wykonują wczesne zwalnianie zasobów. Takie ograniczenie czyni te transakcje niewycofywalnymi, co niesie ze sobą wiele problemów, np. ograniczenie możliwości wykonywania takich transakcji sekwencyjnie [92]. Co więcej wymuszenie ostatecznego zatwierdzenia w tych transakcjach doprowadza do tego, że w systemach z dowolnym wycofaniem (w szczególności w systemach potencjalnie awaryjnych) wczesne zwalnianie zasobów jest wykluczone całkowicie.

Analizujemy istniejące własności bezpieczeństwa dla pamięci transakcyjnej oraz warunki spójności używane w systemach baz danych pod względem wyznaczonych kryteriów i zaprezentowano wyniki analizy w Tablicy 11.3. Tablica ta informuje, które z wyżej wymienionych definicji są spełniane przez konkretną własność. Dodatkowo informuje, czy własność wywodzi się z systemów baz danych i czy jest szeroko używana w systemach pamięci transakcyjnej, a także, w ostatniej kolumnie, ustalono czy dana własność jest silniejsza od uszeregowalności – tzn. czy każda historia spełniona przez daną własność spełnia także własność uszeregowalności. W tabeli podano nazwy własności w języku angielskim.

Przedstawiona w pracy analiza wykazuje, że tylko niewielka liczba istniejących własności pozwala na jakiekolwiek wykorzystanie wczesnego zwalniania zasobów. Spośród pozostałych własności, te które pozwalają na wczesne zwalnianie zmiennych są albo zbyt dozwalające, pozwalając na nadpisywanie uprzednio zwolnionej zmiennej, albo zbyt restrykcyjne, wymagając od transakcji zwalniających zmienne, aby były efektywnie niewycofywalne. W konsekwencji można zauważyć brak własności praktycznie stosowalnej w systemach pamięci rozproszonej z wczesnym zwalnianiem zasobów.

Analiza Istniejących Algorytmów

Następnym krokiem była analiza istniejących algorytmów sterowania współbieżnością dla pamięci transakcyjnej i systemów pamięci transakcyjnej. Ponieważ badania nad pamięcią transakcyjną zaowocowały dużą liczbą rozwiązań, skupiono się na systemach odzwierciedlających następujące aspekty: rozproszone systemy pamięci transakcyjnej, pesymistyczne pamięci transakcyjne, oraz pamięci transakcyjne wykorzystujące wczesne zwalnianie zasobów. Dla każdej kategorii algorytmów dokonano ogólnego przeglądu istniejących algorytmów, oraz dokładnniejszej analizy wybranych przedstawicieli z każdej kategorii.

Wyniki analizy podsumowano w Tablicy 11.2. Kolumna *podejście* wskazuje czy algorytm jest pesymistyczny czy optymistyczny. Kolumna *postęp* wskazuje, czy algorytm jest blokujący (oparty na zamkach) czy też jest pozbawiony czekania. Należy odnotować, że SemanticTM nie wymaga czekania, przy założeniu istnienia odpowiedniego modułu szeregującego transakcje przed wykonaniem. Kolumna *modyfikacje* wskazuje, czy algorytm wprowadza modyfikacje do obiektów podczas wykonania operacji na obiekcie czy modyfikacje są opóźnione do momentu zatwierdzenia. Kolumna *wycofania* mówi, kiedy transakcje w danym algorytmie wykonują wymuszone wycofanie. W kolumnie *a priori* wskazane są dane wymagane do poprawnego wykonania wykonania transakcji jeszcze przed uruchomieniem. Kolumna *obiekty* mówi w jakim modelu ze względu na definicje obiektów działa algorytm. Warto zauważyć, że algorytmy działające w modelu obiektowym niejednorodnym mogą być użyte również w modelu jednorodnym, a algorytmy działające w modelu jednorodnym mogą być użyte także w modelu ze zmiennymi. Z drugiej strony algorytmy oznaczone jako działające z dowolnymi obiektami są używane w modelu zmiennych, ale mogą być trywialnie zgeneralizowane do dowolnego modelu obiektowego. Kolumna *zakleszczenie* informuje czy algorytm dopuszcza wystąpienie zakleszczenia. Kolumna *bezpieczeństwo* wskazuje, jakie własności bezpieczeństwa są spełnione przez algorytm (podane w języku angielskim). Kolumna *zwalnianie zasobów* mówi czy algorytm wykorzystuje technikę wczesnego zwalniania zasobów. Niektóre algorytmy dopuszczają wczesne zwalnianie przez transakcje zmiennych tylko-do-odczytu. W końcu, kolumna *operacje niewycofywalne* wskazuje jak algorytm radzi sobie z operacjami niewycofywalnymi: czy są one wykonywane zawsze poprawnie, czy mogą wystąpić w wycofanej transakcji lub czy mogą być wykonane wielokrotnie. Rozróżnić należy tutaj wycofanie w sensie ogólnym i wycofanie na życzenie programisty—jeśli programista nakaże transakcji wycofać się (poprzez wywołanie operacji wycofania), wycofanie operacji niewycofywalnej jest zamierzone przez programistę, więc jest zachowaniem poprawnym.

Własności Pamięci Transakcyjnej z Wczesnym Zwalnianiem

Warto zauważyć, że pomimo istnienia własności bezpieczeństwa, które pozwalają na wczesne zwalnianie zmiennych, takich jak spójność świata wirtualnego (ang. *virtual world consistency*, *VWC*), nieprzezroczystość elastyczna (ang. *elastic opacity*) [28], czy nieprzezroczystość żywa (ang. *live opacity*), to w praktyce algorytmy pamięci transakcyjnej które używają wczesnego zwalniania nie zaspokajają żadnej z nich, a jedynie relatywnie słabą własność uszeregowalności (ang. *serializability*) lub uszeregowalności konfliktowej (ang. *conflict serializability*). Zauważamy, że silniejsze z tych własności nie mogą być użyte ponieważ wymagają one, żeby transakcje, które zwalniają wcześnie zmienne nie mogły się wycofać. Z drugiej strony, zachowania przedstawionych algorytmów bardzo różnią się od siebie. Np. algorytmy z rodziny blokowania dwufazowego (ang. *two-phase locking*, *2PL*) i algorytm wersjonowania w oparciu o suprema (ang. *Supremum Versio-*

Algorytm	**Podejście**	**Postęp**	**Modyfikacje**	**Wycofania**	**A priori**	**Obiekty**	**Zakleszczenie**	**Bezpieczeństwo**	**Zwalnianie zasobów**	**Operacje niewycofywalne**
B2PL	pesymistyczne	blokujący	p. wykonania	przy zakleszczeniu	∅	dowolne	tak	*serializable*	tak	wycofywane
C2PL	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*RSet*, *WSet*	dowolne	nie	*serializable*	tak	poprawne
S2PL	pesymistyczne	blokujący	p. wykonania	przy zakleszczeniu	∅	dowolne	tak	*strict*	odczyty	wycofywane
R2PL	pesymistyczne	blokujący	p. wykonania	przy zakleszczeniu	∅	dowolne	tak	*rigorous*	nie	wycofywane
CS2PL	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*RSet*, *WSet*	dowolne	nie	*opaque*	odczyty	poprawne
CR2PL	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*RSet*, *WSet*	dowolne	nie	*opaque*	nie	poprawne
CAS2PL	pesymistyczne	blokujący	p. wykonania	dowolne	*RSet*, *WSet*	dowolne	nie	*opaque*	odczyty	wycofywalne przez użytkownika
CAR2PL	pesymistyczne	blokujący	p. wykonania	dowolne	*RSet*, *WSet*	dowolne	nie	*opaque*	nie	wycofywalne przez użytkownika
BVA	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*ASet*	niejednorodne	nie	*opaque*	nie	poprawne
SVA	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*ASet*, suprema	niejednorodne	nie	*serializable*	tak	poprawne
TL2/DTL2	optymistyczne	blokujący	p. zatwierdzenia	przy konflikcie	∅	zmienne	nie	*opaque*	nie	wycofywane
TFA	optymistyczne	blokujący	p. zatwierdzenia	przy konflikcie	∅	jednorodne	nie	*opaque*	nie	wycofywane
MS-PTM	pesymistyczne	blokujący	p. zatwierdzenia	bez wycofywania	∅	zmienne	nie	*opaque*	nie	poprawne
PLE	pesymistyczne	blokujący	p. wykonania	bez wycofywania	∅	zmienne	nie	*opaque*	nie	poprawne
SemanticTM	pesymistyczne	pozbawiony czekania*	p. wykonania	bez wycofywania	*ASet*, zależności	zmienne	nie	*opaque*	nie	powtarzane
DATM	optymistyczne	blokujący	p. zatwierdzenia	przy nadpisaniu, zakleszczeniu i kaskadzie	∅	zmienne	tak	*conflict serializable*	tak	wycofywane

Tablica 11.2: Podsumowanie algorytmów sterowania współbieżnością dla pamięci transakcyjnej.

ning Algorithm, *SVA*) nie pozwalają na nadpisywanie zmiennej po jej zwolnieniu, podczas gdy DATM na to pozwala. Ponieważ te różnice nie są określone przez własności, wnioskujemy, że brakuje odpowiednich własności bezpieczeństwa dla pamięci transakcyjnej wykorzystującej wczesne zwalnianie zasobów.

Rozproszona Pamięć Transakcyjna

Spośród przeanalizowanych algorytmów wyszczególniamy algorytmy, które mogą być użyte do implementacji rozproszonej pamięci transakcyjnej: algorytmy blokowania dwufazowego, algorytmy wersjonowania (ang. *versioning algorithms*): BVA i SVA, oraz DTL2 i TFA. Algorytmy te były projektowane ze szczególnym uwzględnieniem systemów rozproszonych lub są w nich wykorzystywane. Spośród tych systemów BVA i SVA używają globalnego zamka celem przyznania każdej uruchamianej transakcji numeru wersji. Jest to problematyczne ze względu na skalowalność systemu, ponieważ wymaga, żeby każdy klient kontaktował się z jednym konkretnym węzłem sieci. Natomiast, jak pokazano poniżej, globalny zamek można z tych algorytmów wyeliminować na rzecz bardziej złożonego schematu zamków opartych na wiedzy *a priori* wykorzystywanej przez te algorytmy. Ponadto, spośród tych algorytmów, TFA jest zaprojektowany do pracy w modelu przepływu danych, podczas gdy blokowanie dwufazowe, algorytmy wersjonowania i DTL2 działają w modelu przepływu sterowania.

Algorytmy MS-PTM, PLE i DATM są mniej zdatne do implementacji w środowisku rozproszonym, ponieważ używają globalnych zamków, które są problematyczne ze względu na skalowalność. Ponadto, MS-PTM, PLE i DATM implementują mechanizm bezruchu (ang. *quiescence*), który opóźnia zatwierdzanie transakcji do momentu kiedy wszystkie poprzednie transakcje zakończyły proces modyfikacji zmiennych. Dzięki temu, wszystkie transakcje mogą zakończyć się bezkonfliktowo, ale mechanizm ten wymaga komunikacji pomiędzy procesami obsługującymi transakcje. Komunikacja taka jest niepraktyczna w systemie rozproszonym, gdzie klienci mogą być geograficznie rozproszeni, oddzieleni zaporami ogniowymi (ang. *firewalls*) lub, w przypadku urządzeń mobilnych, mogą nie posiadać odpowiedniej mocy obliczeniowej do obsługi wykrywania bezruchu. W konsekwencji przed wprowadzeniem tych algorytmów do środowiska rozproszonego należy najpierw wprowadzić metody pozwalające transakcjom na wypychanie informacji niezbędnych do działania mechanizmu bezruchu oraz rozproszyć globalne zamki. W przeciwieństwie do algorytmów wersjonowania, wymagane tutaj rozwiązania są nietrywialne.

Trudno jest wyobrazić sobie zastosowanie SemanticTM w środowisku rozproszonym ze względu na wymaganie tego systemu co do porządku wykonywania operacji. SemanticTM wymaga, żeby operacje na zmiennych były umieszczone w kolejkach odpowiednich dla tych zmiennych w taki sposób, że dla dowolnych dwóch transakcji wszystkie operacje pierwszej z nich są umieszczone we wszystkich kolejkach przed operacjami drugiej transakcji lub *vice versa*. Wymaganie to jest trudne do zaspokojenia w systemie rozproszonym, gdzie jego egzekwowanie wymagałoby zastosowania mechanizmu szeregującego równie złożonego jak sama pamięć transakcyjna. Z tego powodu należy uznać SemanticTM za algorytm mało praktyczny w kontekście systemów rozproszonych.

Warto zauważyć, że spośród wymienionych systemów nadających się do implementacji w systemach rozproszonych, tylko blokowanie dwufazowe i algorytmy wersjonowania wspierają operacje niewycofywalne.

Nowe Własności

Zarówno wykonany przegląd własności, jak i przegląd algorytmów, sugerują, że brak jest własności dobrze opisujących praktyczne algorytmy sterowania współbieżnością w pamięci transakcyjnej. W konsekwencji, wprowadzono dwie nowe własności, które wypełniają ową niszę.

Nieprzezroczystość do Ostatniego Użycia

Pierwszą zaprezentowaną własnością jest własność nieprzezroczystości do ostatniego użycia (ang. *last-use opacity*). Własność ta jest oparta o definicję nieprzezroczystości, więc zachowuje ona silne gwarancje bezpieczeństwa. Wyjątkiem jest wymaganie, żeby transakcje zawsze czytały wartości zmiennych które były zapisane przez transakcje uprzednio zatwierdzone. Nieprzezroczystość do ostatniego użycia zezwala na czytanie danych zapisanych przez transakcje niezatwierdzone pod warunkiem, że dane te były zapisane przez ostateczne operacje zapisu (ang. *closing writes*). Ostateczna operacja zapisu to taka operacja, po której nie wystąpi żadna inna operacja zapisu na tej samej zmiennej i będzie to prawdą dla wszystkich potencjalnych wykonań tej transakcji.

Własność nieprzezroczystości do ostatniego użycia mówi, że transakcje, które są zatwierdzone mogą czytać tylko wartości zapisane przez inne transakcje, które są również zatwierdzone. Ponadto, transakcje niezatwierdzone (żywe lub wycofane) mogą czytać wartości zapisane przez inne transakcje, jeśli transakcje te są zatwierdzone lub jeśli wartości te były zapisane przez ostateczną operację zapisu jakiejś niezatwierdzonej transakcji.

Określona w ten sposób własność bezpieczeństwa może być używana do opisu systemów z wczesnym zwalnianiem zasobów, jednocześnie wykluczając nadpisywanie i pozwalając na wycofywanie transakcji, które zwolniły zasoby wcześnie.

Gwarancje

Nieprzezroczystość do ostatniego użycia zapewnia następujące silne gwarancje względem poprawności wykonania kodu transakcyjnego:

Uszeregowalność (ang. *Serializability*) Jeśli transakcja zostanie zatwierdzona, to dowolna wartość przez nią odczytana może być wyjaśniona przez operacje poprzedzających lub współbieżnych zatwierdzonych transakcji. Transakcje czytające stan niespójny nie zostaną zatwierdzone.

Porządek Czasu Rzeczywistego (ang. *Real-time Order*) Kolejne transakcje nie będą zamieniane kolejnością celem zaspokojenia uszeregowalności, wiec poprawny przeplot będzie odpowiadać zewnętrznemu względem systemu zegarowi.

Odzyskiwalność (ang. *Recoverability*) Jeśli transakcja odczyta wartość zapisaną przez drugą transakcję, to pierwsza z nich zostanie zatwierdzona dopiero po tym, jak druga z nich zostanie zatwierdzona.

Wykluczenie Nadpisywania (ang. *Precluding Overwriting*) Jeśli transakcja odczytuje wartość zapisaną do jakiejś zmiennej przez drugą transakcję, to ta druga transakcją nie wykona ponownie zapisu do tejże zmiennej.

Wycofywanie po Wczesnym Zwolnieniu (ang. *Aborting Early Release*) Transakcja, która wykonała wczesne zwolnienie zasobu, może następnie zostać wycofana.

Wyłączny Dostęp (ang. *Exclusive Access*) Transakcja ma wyłączny dostęp do danej zmiennej, od chwili wykonania pierwszej operacji na tej zmiennej i co najmniej do momentu ostatecznego zapisu na niej.

między pierwszą operacją, którą wykona na tej zmiennej, i minimum do ostatecznej modyfikacji którą wykonuje na tej zmiennej.

Niespójny Odczyt Systemu

Nieprzezroczystość do ostatniego użycia nie wyklucza sytuacji, gdzie transakcja odczytuje wartość zapisaną przez inną transakcje, która to następnie zostanie wycofana. Rezultatem takiej sytuacji jest odczyt niespójnego stanu systemu przez pierwszą z transakcji, co może mieć różne implikacje dla poprawności działania programu w zależności od modelu systemu.

W modelu systemu z transakcjami wyłącznie zatwierdzającymi transakcje nie mogą samoczynnie zostać wycofane – mogą jednak zostać wycofane siłowo, np. w konsekwencji konfliktu. Ponieważ nieprzezroczystość do ostatniego użycia wyklucza odczyt danych zapisanych przez transakcje przed ostatecznym zapisem do danej zmiennej, transakcja wycofana nie zapisywałaby do tej zmiennej jakiejkolwiek innej wartości. W konsekwencji, wartość zapisana przez tę transakcje byłaby identyczna niezależnie, czy transakcja ostatecznie wykonałaby zatwierdzenie czy wycofanie. Dlatego też można uważać stan zaobserwowany przez inne transakcje za bezpieczny. Innymi słowy, jeśli wartość odczytana z wycofanej transakcji spowodowałaby błąd, wystąpiłby on niezależnie od tego, czy ta transakcja została wycofana czy zatwierdzona. W takim wypadku, programista ma gwarancję, że odczyt technicznie niespójnego stanu nie wprowadzi niepoprawnego zachowania do systemu. Warto zauważyć, że taki model systemu jest powszechnie używany (np. [28, 5, 6]).

Alternatywnym jest model z transakcjami dowolnie wycofywanymi. W takim modelu transakcja może wykonać operację wycofania dowolnie, także w ramach swojej „logiki biznesowej". W takim wypadku możliwym jest, że transakcja będzie używała operacji wycofania właśnie do naprawieniu stanu systemu po zapisaniu do zmiennych niespójnych, niebezpiecznych wartości. Przykładowo, transakcja T_i może zapisać do jakiejś zmiennej x wartość v, po czym, jeśli v łamie predefiniowany warunek niezmienny, wykonać wycofanie. Natomiast, jeśli zapisanie do x wartości v będzie ostatecznym zapisem, x może zostać odczytane przez inną transakcję T_j, jeszcze zanim T_i wykona wycofanie. W takim wypadku T_j może podjąć niebezpieczne działanie na podstawie wartości v. W konsekwencji, używanie modelu dowolnego wycofania może powodować niebezpieczne zachowania. Zachowań tych można uniknąć, np. nigdy nie wprowadzając do zmiennych wartości łamiących niezmienniki, lub przesuwając potencjalne wykonanie wycofania przed operację ostateczną. Jeśli rozwiązania te są niewystarczające, poniżej przedstawiono silniejszą wersję własności nieprzezroczystości do ostatniego użycia, która wyklucza niespójny odczyt stanu systemu.

Wprowadzano także w pracy alternatywny wariant powyższego modelu, który nazwano modelem z ograniczonym wycofaniem (ang. *restricted abort model*). W modelu tym zakładamy, że transakcje mogą wykonać operację wycofania dowolnie, ale operacja ta jest umieszczana w kodzie transakcji automatycznie przez niezależne od transakcji zdarzenia, np. awarie lub przerwania, a nie są one częścią logiki biznesowej transakcji. Jeśli programista nie może związać logiki transakcji z operacją wycofania, własność daje takie same gwarancje względem niespójnego stanu jak w modelu z transakcjami dążącymi do zatwierdzenia.

Silna Nieprzezroczystość do Ostatniego Użycia

Własność silnej nieprzezroczystości do ostatniego użycia (ang. *strong last-use opacity*) jest odmianą własności nieprzezroczystości do ostatniego użycia, która zachowuje się podobnie, ale wykorzystuje inną definicję operacji ostatecznych. Silnie ostateczna operacja

Własność	Zast.	Def. 4	Def. 5	Def. 6	⊆Uszeregowalne
Last-use opacity	TM	✓	×	✓	✓
Strong last-use opacity	TM	✓	×	✓	✓

Tablica 11.3: Podsumowanie dopuszczalności wczesnego zwalniania zasobów we wprowadzonych własnościach.

zapisu to taka operacja, po której nie wystąpi żadna inna operacja zapisu na tej samej zmiennej ani operacja wycofania i będzie to prawda dla wszystkich potencjalnych wykonań tej transakcji. Silna nieprzezroczystość do ostatniego użycia mówi, że transakcje, które są zatwierdzone, mogą czytać wartości zapisane przez inne transakcje, które są również zatwierdzone. Transakcje niezatwierdzone natomiast mogą czytać wartości zapisane przez inne transakcje, o ile transakcje te są zatwierdzone, lub jeśli odczytywane wartości były zapisane przez silnie ostateczną operację zapisu transakcji niezatwierdzonej.

Zdefiniowana w ten sposób własność daje te same własności co nieprzezroczystość do ostatniego użycia, a ponadto wyklucza negatywne konsekwencje niespójnego stanu we wszystkich modelach. Wiążę się to jednak z wykluczeniem potencjalnie poprawnych wykonań. Oznacza to także, że w systemach, gdzie operacja wycofania może wystąpić w dowolnym momencie (np. ze względu na awarie), wczesne zwalnianie zasobów jest całkowicie wykluczone.

Nowe Algorytmy

W pracy wprowadzono nowe algorytmy pesymistycznego sterowania współbieżnością dla pamięci transakcyjnej, które kierujemy w szczególności do systemów rozproszonej pamięci transakcyjnej, w których występować mogą operacje niewycofywalne. Nowe algorytmy opierają się o rodzinę algorytmów wersjonowania, a w szczególności algorytm SVA. Algorytmy wersjonowania są pesymistyczne i pozbawione wycofań, więc nie powodują błędnych wykonań operacji niewycofywalnych. Dodatkowo SVA wykorzystuje mechanizm wczesnego zwalniania zmiennych, który może być użyty do wykonania wchodzących w konflikt transakcji częściowo równolegle, co pozwala na krótsze przeploty, niż np. algorytmy blokowania dwufazowego.

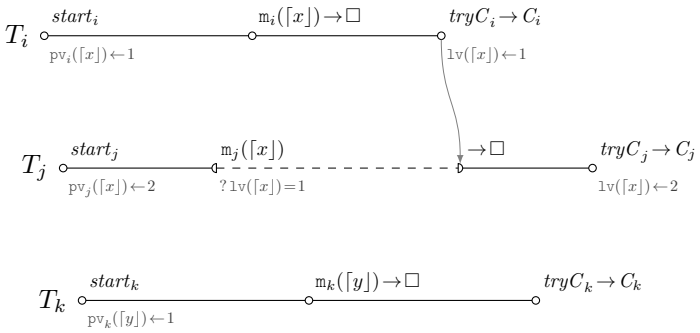
W poniższych opisach algorytmów użyto notacji x, y, z kiedy odnosimy się do zmiennych, natomiast obiekty (zarówno w modelu obiektów jednorodnych, jak i niejednorodnych) oznaczono dla odróżnienia jako $\lceil x \rfloor, \lceil y \rfloor, \lceil z \rfloor$.

SVA

Algorytm SVA [96, 97] jest podstawowym algorytmem wersjonowania, na którym oparto algorytmy zaproponowane w rozprawie. Algorytmy wersjonowania używają liczników wersji w celu ustalenia, czy dana transakcja może w bieżącym momencie wykonać operację na konkretnym obiekcie współdzielonym czy owa operacja musi być opóźniona celem uniknięcia konfliktu.

Mechanizm Wersjonowania

Intuicyjnie, liczniki te działają przez analogię do zarządzania kolejką w banku: klienci którzy przychodzą do banku pobierają numerek z automatu i czekają z podejściem do



Rys. 11.1: Mechanizm wersjonowania w SVA.

okienka aż ich numerek zostanie wywołany. W analogii tej klient jest transakcją, okienko obiektem współdzielonym, a numerek pobrany przez klienta to wersja transakcji dla tego okienka, która jest porównywana z wywoływanym numerkiem—wersją obiektu.

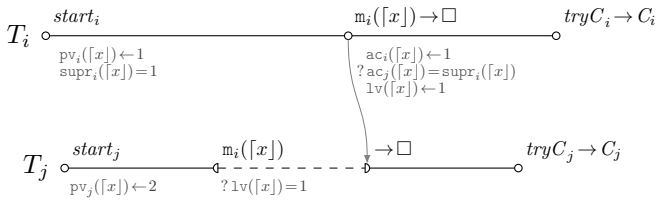
Konkretnie, za każdym razem, kiedy transakcja T_i jest uruchamiana, pobiera ona *prywatną wersję* $\mathtt{pv}_i(\lceil x \rfloor)$ dla każdego obiektu współdzielonego $\lceil x \rfloor$, na którym transakcja będzie wykonywać operacje (zbiór tych obiektów jest znany z góry). Wartości wersji prywatnych dla obiektu $\lceil x \rfloor$ nadawane kolejnym transakcjom są kolejnymi liczbami naturalnymi. Dodatkowo, wartości te nadawane są atomowo w taki sposób, że mając dwie transakcje T_i, T_j, jeśli dla jakiegoś obiektu $\lceil x \rfloor$, $\mathtt{pv}_i(\lceil x \rfloor) < \mathtt{pv}_j(\lceil x \rfloor)$, to dla każdego innego obiektu $\lceil y \rfloor$, na którym obie transakcje będą wykonywać operacje, $\mathtt{pv}_i(\lceil y \rfloor) < \mathtt{pv}_j(\lceil y \rfloor)$. W celu zapewnienia tego warunku SVA używa globalnego zamka, który szereguje operacje rozpoczęcia wykonywane przez wszystkie transakcje.

Po pobraniu wersji, SVA używa wersji prywatnych do podjęcia decyzji, czy transakcja T_i może wykonać operacje na obiekcie $\lceil x \rfloor$, porównując $\mathtt{pv}_i(\lceil x \rfloor)$ z *wersją lokalną* obiektu $\mathtt{lv}(\lceil x \rfloor)$. Wersja lokalna obiektu jest równa wartości wersji prywatnej transakcji, która jako ostatnia tego używała obiektu i już zakończyła jego używanie (np. już została zatwierdzona). W takim wypadku, biorąc pod uwagę, że transakcje mają kolejne wartości wersji prywatnej, transakcja T_i może wykonywać operacje na $\lceil x \rfloor$ wtedy, gdy jej wartość wersji prywatnej dla $\lceil x \rfloor$ jest kolejna względem wartości wersji lokalnej $\lceil x \rfloor$, tj. $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$. Warunek ten nazwano *warunkiem dostępu*.

Kiedy transakcja zakończyła wykonywanie wszystkich operacji, wykonuje ona operację zatwierdzenia, kiedy to dla wszystkich obiektów, dla których pobrała wersje prywatne, zapisuje swoją wartość wersji prywatnej dla tego obiektu do licznika wersji lokalnej tego obiektu. W kontekście algorytmów wersjonowania nazywa się to *zwolnieniem obiektu* $\lceil x \rfloor$.

Na Rys. 11.1 pokazano przykład działania mechanizmu wersji. Tutaj T_i i T_j próbują wykonać operacje na obiekcie $\lceil x \rfloor$ w tym samym czasie. Transakcja T_i startuje wcześniej, więc $\mathtt{pv}_i(\lceil x \rfloor) = 1$, natomiast T_j stratuje jako druga, więc $\mathtt{pv}_j(\lceil x \rfloor) = 2$. Ponieważ początkowo $\mathtt{lv}(\lceil x \rfloor) = 0$, to T_j nie jest w stanie wykonać operacji na $\lceil x \rfloor$, więc czeka. Z kolei dla T_i warunek $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$ jest prawdziwy, więc T_i wykonuje operację (metodę) $\mathtt{m}$ na $\lceil x \rfloor$ bez czekania (zwracana wartość nie jest istotna, więc jest oznaczona $\square$). Kiedy T_i zostanie zatwierdzona, ustawia $\mathtt{lv}(\lceil x \rfloor)$ na $\mathtt{pv}_i(\lceil x \rfloor)$ czyli 1, co powoduje, że T_j spełni teraz warunek $\mathtt{pv}_j(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$ i przejdzie do wykonania operacji na $\lceil x \rfloor$. W międzyczasie T_k może wykonać operację na $\lceil y \rfloor$ zupełnie równolegle.

Mechanizm wersjonowania zapewnia transakcjom wyłączny dostęp do obiektów, jednocześnie powodując, że transakcje o rozłącznym zbiorach obiektów, na których wykonują operacje, nie blokują się nawzajem.



Rys. 11.2: Przykład wczesnego zwalniania zmiennych w SVA.

Wczesne Zwalnianie Zasobów

Dodatkowo, SVA używa mechanizmu wczesnego zwalniania zasobów opartego o *suprema*. Supremum dla obiektu $\lceil x \rfloor$ (oznaczone $\mathtt{supr}_i(\lceil x \rfloor)$) w transakcji T_i to liczba informująca jaka jest maksymalna liczba wykonań operacji na $\lceil x \rfloor$ przez transakcję T_i. Jeśli supremum jest zdefiniowane *a priori* w transakcji T_i, to transakcja liczy wywołania operacji na obiekcie $\lceil x \rfloor$ za pomocą licznika $\mathtt{ac}_i(\lceil x \rfloor)$, i jeśli po wywołaniu operacji na $\lceil x \rfloor$ liczba wywołań jest równa supremum, to obiekt $\lceil x \rfloor$ zostaje zwolniony przez zapisanie wartości wersji prywatnej transakcji do wersji lokalnej obiektu. Dzięki temu inna transakcja może zacząć wykonywać operacje na tym obiekcie jeszcze zanim T_i zostanie zatwierdzona. Z drugiej strony, ponieważ T_i osiągnęła supremum dla $\lceil x \rfloor$, to nie wykona kolejnych operacji na $\lceil x \rfloor$.

Przykład przeplotu wygenerowanego przez SVA z użyciem wczesnego zwalniania zmiennych jest pokazany na Rys. 11.2. Tutaj transakcje T_i i T_j wykonują operacje na $\lceil x \rfloor$. Tak jak na Rys. 4.12, ponieważ wersja prywatna T_i dla $\lceil x \rfloor$ jest niższa niż w przypadku T_j, ta pierwsza wykonuje swoje operacje na $\lceil x \rfloor$ jako pierwsza, a T_j czeka aż $\lceil x \rfloor$ będzie zwolniony. Tutaj natomiast T_i zna swoje supremum dla $\lceil x \rfloor$, t.j. $\mathtt{supr}_i(\lceil x \rfloor) = 1$. Więc wykonawszy swoją operację na $\lceil x \rfloor$, T_i zwiększa $\mathtt{ac}_i(\lceil x \rfloor)$, co oznacza, że supremum zostało osiągnięte, tj. $\mathtt{ac}_i(\lceil x \rfloor) = \mathtt{supr}_i(\lceil x \rfloor)$. W takim wypadku T_i zwalnia $\lceil x \rfloor$ od razu, zamiast czekać do momentu zatwierdzenia. W rezultacie, T_j może wykonywać operacje na $\lceil x \rfloor$ wcześniej. Transakcja T_j może nawet być zatwierdzona przed T_i.

Mechanizm wczesnego zwalniania zasobów przez suprema pozwala transakcjom korzystającym z tych samych obiektów na wykonywanie się z większym współczynnikiem zrównoleglenia, niż w wypadku samego mechanizmu wersjonowania, jednocześnie zapewniając, że odczytany z niezatwierdzonych transakcji stan zawsze będzie prawidłowy. Pozwala to na efektywne i poprawne przeplatanie transakcji.

Własności

W ramach pracy zademonstrowano gwarancje bezpieczeństwa SVA. Konkretnie, wprowadzono dekompozycję – technikę dowodzenia opartą na rafinacji obserwacyjnej (ang. *observational refinement*) która pozwala pokazać, że dany przeplot jest nierozróżnialny od poprawnego przeplotu nieprzezroczystego pod względem efektów wykonania operacji. W ten sposób demonstrujemy, że mimo tego, że SVA nie jest nieprzezroczystym algorytmem, to każdy przeplot wygenerowany przez SVA jest nierozróżnialny od poprawnego przeplotu nieprzezroczystego (Twierdzenie 7). Ponadto, SVA nie powoduje zakleszczeń oraz zapewnia silną progresywność.

Rozproszone Pobieranie Wersji

Celem zapewnienia atomowości pobierania wersji przy starcie transakcji algorytm SVA używa globalnego zamka, który każda transakcja pobiera na początku swojej procedury

inicjalizacji i zwalnia na jej końcu. Zamek ten rodzi problem, jeśli algorytm ma być zaimplementowany w środowisku rozproszonym, ponieważ globalny zamek stanowi przeszkodę dla skalowalności—niezależnie od tego, ile nowych węzłów dodamy do systemu, zawsze wszyscy klienci muszą kontaktować się z jednym węzłem, który jest odpowiedzialny za globalny zamek. Dodatkowo, jeśli węzeł, na którym znajduje się zamek, ulegnie awarii, to taka awaria (częściowa) paraliżuje cały system.

Aby wyeliminować problemy związane z globalnym zamkiem, zaprezentowano w pracy wariant SVA, który używa rozproszonych zamków w procedurze rozpoczęcia transakcji. Wariant ten wymaga, żeby z każdym obiektem współdzielonym $\lceil x \rfloor$ związany był zamek $\texttt{lk}(\lceil x \rfloor)$ (zlokalizowany na tym samym węźle co $\lceil x \rfloor$). Wtedy każda transakcja, zamiast pobierać globalny zamek przed pobraniem wersji, będzie pobierać zbiór zamków związanych z obiektami, na których transakcja planuje wykonywać operacje. Oznacza to, że transakcje, które nie współdzielą obiektów, mogą wykonywać procedurę rozpoczęcia równolegle, a także, że nie ma pojedynczego zamka, który musi obsługiwać wszystkie transakcje.

Dodatkowo, celem uniknięcia zakleszczeń wymagamy, żeby zamki były zawsze pobierane w według globalnie ustalonego porządku. Jeśli ten porządek jest zachowany, nie mogą wystąpić cykle oczekiwań, a więc zakleszczenia są wykluczone. Jest to prostsze rozwiązanie niż np. istniejące rozwiązanie w konserwatywnych algorytmach blokowania dwufazowego, gdzie unikanie zakleszczeń zaimplementowane jest przez cykliczne odpytywanie o stan zamków.

SVA+R

SVA jest algorytmem działającym w modelu transakcji wyłącznie zatwierdzających (a nawet wykonania pozbawione są całkowicie wycofań). Natomiast, jeśli algorytm pamięci transakcyjnej ma być praktyczny w dowolnym systemie, powinien on wspierać operację wycofania, a więc działać w modelu dowolnych wycofań. Wprowadzono więc nowy algorytm wersjonowania SVA+R (ang. *SVA with rollback*), który rozszerza SVA o operację wycofania. Wymaga to wprowadzenia dodatkowych mechanizmów opisanych poniżej.

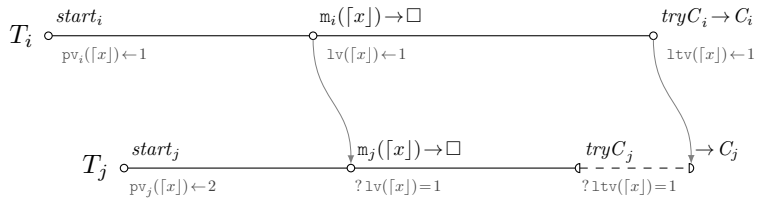
Odtwarzanie Obiektów

Po pierwsze, wprowadzono bufor $\texttt{st}_i(\lceil x \rfloor)$, który transakcja T_i utrzymuje dla każdego obiektu $\lceil x \rfloor$, na którym wykonuje operacje. Obiekt $\lceil x \rfloor$ jest kopiowany do bufora $\texttt{st}_i(\lceil x \rfloor)$ w momencie kiedy transakcja po raz pierwszy spełni warunek dostępu do obiektu $\lceil x \rfloor$, czyli tuż przed wykonaniem pierwszej operacji na $\lceil x \rfloor$. Bufor jest następnie używany wewnątrz samej procedury wycofania transakcji: transakcja przed zwolnieniem obiektu $\lceil x \rfloor$ przywróci go do wcześniejszej postaci, kopiując zawartość $\texttt{st}_i(\lceil x \rfloor)$ z powrotem do $\lceil x \rfloor$.

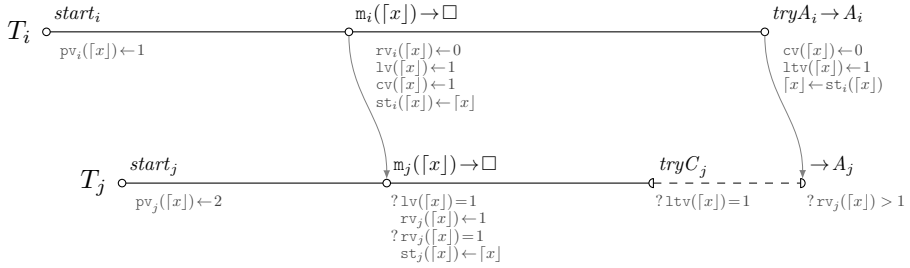
Porządek Zatwierdzania

Druga modyfikacja ma na celu zapobieganie sytuacjom, gdzie transakcja T_i zwolni obiekt $\lceil x \rfloor$, pozwalając innej transakcji T_j na odczytanie jego stanu, a następnie T_j wykona zatwierdzenie, zanim T_i zostanie zatwierdzona. Jest to problematyczne, ponieważ istnieje możliwość, ze T_i ostatecznie zostanie wycofana, co powoduje, że T_j została zatwierdzona wykonawszy operacje na już-niespójnym stanie.

W tym celu SVA+R porządkuje wykonania operacji zakończenia transakcji: wycofań i zatwierdzeń, w taki sam sposób jak dostępy do obiektów. W związku z tym wprowadzono *lokalną końcową wersję* obiektu (ang. *local terminal version*) oznaczoną $\texttt{ltv}(\lceil x \rfloor)$ dla obiektu $\lceil x \rfloor$. Wersja ta działa podobnie jak $\texttt{lv}(\lceil x \rfloor)$, ale transakcje zapisują tam



Rys. 11.3: Wymuszenie porządku zatwierdzania transakcji w SVA+R.



Rys. 11.4: Wykrycie niespójnego stanu systemu w SVA+R.

swoją wersję prywatną tylko wtedy, gdy zakończyły wykonywać zatwierdzenie lub wycofanie. Dodatkowo, każda transakcja T_i przed wykonaniem zatwierdzenia lub wycofania czeka, aż warunek $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{ltv}(\lceil x \rfloor)$ stanie się prawdziwy. W konsekwencji, jeśli transakcja T_j będzie operować na obiekcie zmodyfikowanym przez T_i, to zatwierdzenie T_j będzie opóźnione tak, żeby wystąpiło po zatwierdzeniu lub wycofaniu T_i. Przykład takiej sytuacji jest pokazany na Rys. 11.3.

Kaskadowe Wycofania

Trzecia modyfikacja to rozwiązanie sytuacji gdzie transakcja T_i zwolni obiekt $\lceil x \rfloor$, pozwalając innej transakcji T_j na odczytanie jego stanu, a następnie T_i zostanie wycofana. Powoduje to, że T_j operuje na technicznie niespójnym stanie, więc musi także zostać wycofana.

SVA+R rozpoznaje tą sytuację i zmusza T_j do wykonania wycofania. Wykrywanie niespójnego stanu jest osiągnięte za pomocą dwóch liczników: wersji obecnej (ang. *current version*) oznaczonej $\mathtt{cv}(\lceil x \rfloor)$ dla obiektu $\lceil x \rfloor$ i wersji naprawczej (ang. *recovery version* oznaczonej $\mathtt{rv}_i(\lceil x \rfloor)$ dla obiektu $\lceil x \rfloor$ i transakcji T_i. Wersja obecna wyznacza najnowszą spójną wersję obiektu, podczas gdy wersja naprawcza oznacza ostatnią spójną wersję tego obiektu, która została zaobserwowana przez daną transakcje. Transakcje przypisują coraz większe wartości wersji obecnej obiektu w momencie gdy transakcje te są zatwierdzane lub zwalniają obiekt wcześnie (na podstawie swojej wersji prywatnej). Natomiast, jeśli transakcja jest wycofywana, to wartość wersji obecnej obiektu jest cofana do wartości mniejszej (na podstawie wersji naprawczej transakcji). Z kolei wartość wersji naprawczej transakcji jest pobierana z wersji obecnej, kiedy transakcja po raz pierwszy spełni warunek dostępu do obiektu $\lceil x \rfloor$. Można więc zauważyć, że jeśli transakcja zaobserwowała jakąś wartość wersji obecnej obiektu, która została zapisana w liczniku wersji naprawczej tej transakcji, a następnie jakaś wcześniejsza transakcja została wycofana i zmniejszyła wartość wersji obecnej, to $\mathtt{rv}_i(\lceil x \rfloor) > \mathtt{cv}(\lceil x \rfloor)$. W konsekwencji transakcje SVA+R sprawdzają warunek $\mathtt{rv}_i(\lceil x \rfloor) > \mathtt{cv}(\lceil x \rfloor)$ przed wykonaniem dowolnej operacji oraz zatwierdzenia dla każdej zmiennej, dla której pobrana została wersja naprawcza. Jeśli warunek jest prawdziwy, to wiadomo, że transakcja działa na niespójnym stanie, więc będzie zmuszona do wykonania wycofania zamiast wykonania zamierzonej operacji.

Pokazano przykład takiego scenariusza na Rys. 11.4. Transakcje T_i i T_j wykonują operacje na obiekcie $\lceil x \rfloor$ i mają przyznane wersje prywatne dla $\lceil x \rfloor$ równe odpowiednio 1 i 2. W konsekwencji, T_i wykonuje operację na $\lceil x \rfloor$ jako pierwsza. W tym momencie T_i ustawia $\mathtt{rv}_i(\lceil x \rfloor)$ na obecną wartość $\mathtt{cv}(\lceil x \rfloor) = 0$ (wartość początkowa z założenia) i kopiuje obiekt do bufora. Następnie T_i wykonuje operację na $\lceil x \rfloor$ i zwalnia $\lceil x \rfloor$ (na podstawie supremum), przez co T_i także podnosi $\mathtt{cv}(\lceil x \rfloor)$ do wartości swojej wersji prywatnej dla $\lceil x \rfloor$, czyli 1. Następnie T_j spełnia warunek dostępu do $\lceil x \rfloor$ po raz pierwszy, więc kopiuje $\lceil x \rfloor$ do bufora i ustawia $\mathtt{rv}_j(\lceil x \rfloor)$ na obecną wartość $\mathtt{cv}(\lceil x \rfloor) = 1$. Skoro $\mathtt{rv}_i(\lceil x \rfloor) = \mathtt{cv}(\lceil x \rfloor)$, to zezwala się na wykonanie operacji. Następnie transakcja T_j próbuje wykonać zatwierdzenie, ale nie może tego zrobić, ponieważ T_i nie została zatwierdzona, dlatego T_j czeka. W międzyczasie transakcja T_i zostanie arbitralnie wycofana. Powoduje to, że transakcja zapisuje swoją wersję naprawczą z powrotem do wersji obecnej obiektu $\mathtt{cv}(\lceil x \rfloor) = 0$ i obiekt zostaje przywrócony z bufora. Wówczas, jeśli transakcja T_j wykona jakąkolwiek operacje lub spróbuje się zatwierdzić, to prawdziwy będzie warunek $\mathtt{rv}_j(\lceil x \rfloor) > \mathtt{cv}(\lceil x \rfloor)$, co spowoduje, że transakcja T_j zostanie forsownie wycofana.

Własności

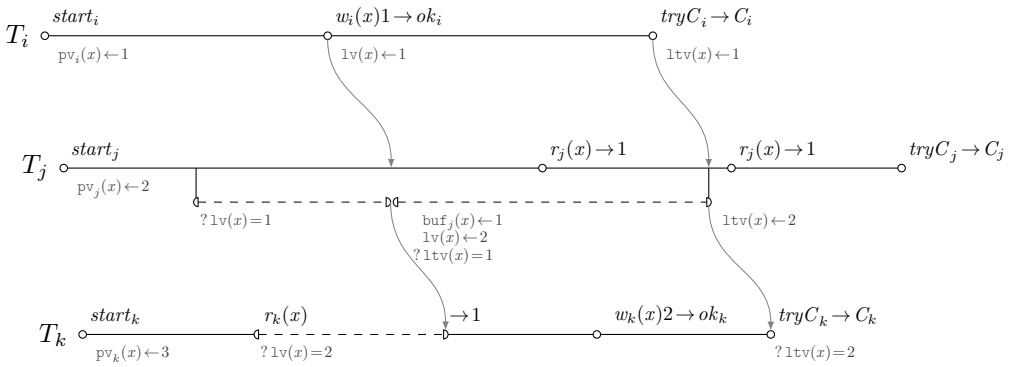
SVA+R nie powoduje zakleszczeń, oraz zapewnia silną progresywność, tak jak SVA. Jednak w przeciwieństwie do SVA, nie można pokazać, że SVA+R będzie generował nieodróżnialne przeploty od przeplotów nieprzezroczystych. Jest tak, ponieważ może się zdarzyć, że transakcja odczyta stan innej transakcji, która to zostanie ostatecznie wycofana. Natomiast, jak pokazuje Twierdzenie 8, SVA+R spełnia własność nieprzezroczystości do ostatniego użycia.

RSVA+R

Ponieważ wprowadzenie mechanizmu dowolnego wycofywania transakcji do algorytmu powoduje, że SVA+R pozwala na kaskadowe wycofania i niespójne widoki, to występuje problem w kontekście operacji niewycofywalnych. To znaczy, może dojść do sytuacji, gdzie transakcja odczyta wartość obiektu zwolnionego wcześniej przez wcześniejszą transakcję i wycofanie wcześniejszej z transakcji spowoduje wycofanie obu. Jeśli którakolwiek transakcja zawierała operacje niewycofywalne, to są one w takim przypadku obsłużone niepoprawnie. W szczególności, transakcja zmuszona do wycofania na skutek wycofania innej, zwalniającej obiekty wcześnie transakcji nie jest w stanie takiego wycofania przewidzieć.

Celem zażegnania tego problemu wprowadzono w pracy wariant algorytmu SVA+R nazwany RSVA+R, który pozwala na zdefiniowane klasy transakcji *niechętnych* (ang. *reluctant*), które nigdy nie są wycofywane. Jest tak dlatego, że forsowne wycofanie transakcji w SVA+R wymaga, żeby transakcja operowała na zmiennej zwolnionej wcześnie przez inną transakcję która zostanie ostatecznie wycofana. Można tę sytuację wykluczyć, jeśli transakcja nie przyjmie zmiennej zwolnionej wcześniej, lecz poczeka, aż wcześniejsza transakcja wykona zatwierdzenie lub się wycofa. Transakcję, która nie przyjmuje zmiennych zwolnionych wcześniej nazywamy właśnie transakcją niechętną, a implementacja tej transakcji zmienia warunek dostępu do zmiennych z $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$ na warunek $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{ltv}(\lceil x \rfloor)$. Jeśli transakcja niechętna nigdy nie jest zmuszona do wycofania, to jest ona podobna do transakcji niewycofywalnej, ale może ona jednocześnie wykonać operację wycofania samodzielnie.

RSVA+R umożliwia doprowadzenie do poprawnego wykonania transakcji z operacjami niewycofywalnymi. Rozwiązanie to wprowadza *trade-off* między poprawnością wykonania operacji niewycofywalnych i wydajnością systemu. Jeśli zbiór transakcji niechętnych jest duży, to system pamięci transakcyjnej ma mało szans na zrównoleglenie



Rys. 11.5: Optymalizacja tylko-do-odczytu w OptSVA+R.

wykonań skonfliktowanych transakcji. W konsekwencji transakcje skonfliktowane niechętne będą często wykonywane sekwencyjnie. Warto zauważyć, że w przeciwieństwie do rozwiązań z [92], które wymagają, żeby transakcje niewycofywalne były wykonywane pojedynczo, RSVA+R mimo wszystko pozwala na wykonanie niechętnych nieskonfliktowanych transakcji równolegle. Z drugiej strony, jeśli zbiór transakcji niechętnych jest niewielki, to algorytm RSVA+R zapewnia jednocześnie w pełni poprawne wykonanie operacji niewycofywalnych oraz wysoki stopień zrównoleglenia transakcji.

RSVA+R spełnia te same własności co SVA+R.

OptSVA+R

Algorytmy wersjonowania zaprezentowane do tej pory działają w oparciu o model obiektów niejednorodnych oraz przy założeniu, że semantyka operacji wykonywanych na obiektach jest zastrzeżona lub ulega zmianom dynamicznie w trakcie działania systemu. W takich systemach traktowanie wszystkich operacji konserwatywnie jako potencjalny odczyt i modyfikacja jest praktycznym uniwersalnym rozwiązaniem.

Z drugiej strony, w systemach takich jak rozproszone magazyny danych czy nierozproszone systemy transakcyjne częściej wykorzystuje się prostsze obiekty jednorodne lub zmienne. W takich systemach algorytmy wersjonowania są dużo mniej wydajne niż alternatywne pamięci transakcyjne, ponieważ nie wprowadzają one optymalizacji generowanych przeplotów na podstawie semantyki operacji. Przykładowo, dwie operacje odczytu zmiennej we współbieżnych transakcjach mogą być zawsze wykonane równolegle względem siebie, podczas gdy warianty SVA+R zawsze wykonają je sekwencyjnie.

W konsekwencji zaprezentowano algorytm OptSVA+R, który rozszerza algorytm SVA+R i sprowadza go do modelu zmiennych, wykorzystując wiedzę odnośnie semantyki operacji i uproszczoną definicję stanu obiektów w celu wprowadzenia optymalizacji mających na celu maksymalne zrównoleglenie skonfliktowanych transakcji. OptSVA+R wprowadza do SVA+R buforowanie operacji których efekty nie są widoczne na zewnątrz transakcji celem opóźnienia sprawdzania warunku dostępu do zmiennych. Dodatkowo OptSVA+R wprowadza nowatorski mechanizm przekazywania wykonania niektórych operacji do osobnych wątków, jeśli operacje te opóźniłyby wykonanie transakcji, a ich wyniki nie są niezbędne do kontynuowania obliczeń.

Zmienne Tylko-do-odczytu

Pierwszą optymalizacją możliwą dzięki rozróżnieniu operacji odczytu od zapisu jest wykorzystywane w większości istniejących pamięci transakcyjnych zrównoleglenie wykonań

transakcji tylko-do-odczytu. OptSVA+R idzie o krok dalej, pozwalając także na częściowe zrównoleglenie wykonań operacji odczytu na zmiennych tylko-do-odczytu nawet w transakcjach, które wykonują zapisy na innych zmiennych.

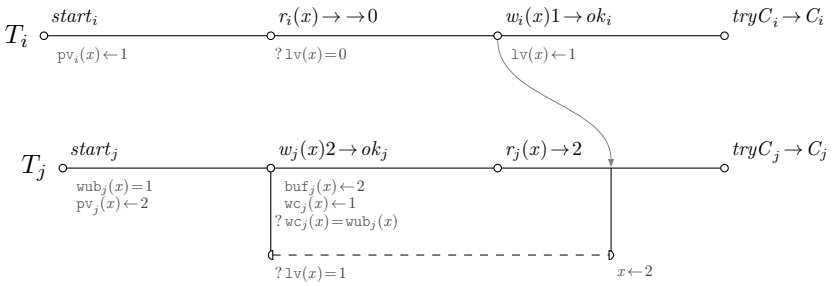
W OptSVA+R suprema rozbite są na dwie wartości: maksymalna liczba odczytów zmiennej, które transakcja T_i wykona na zmiennej x, czyli $\mathtt{rub}_i(x)$, oraz analogiczna maksymalna liczba zapisów $\mathtt{wub}_i(x)$. Jeśli transakcja T_i zastanie sytuację, gdzie dla zmiennej x $\mathtt{rub}_i(x) > 0$ i $\mathtt{wub}_i(x) = 0$, to taka zmienna jest zmienną tylko-do-odczytu. W przypadku takich zmiennych transakcja OptSVA+R zapisze spójną wartość zmiennej x do bufora oznaczonego $\mathtt{buf}_i(x)$ i wykona wszystkie odczyty nie bezpośrednio ze zmiennej, a właśnie z bufora $\mathtt{buf}_i(x)$. Podczas gdy zapisanie zmiennej do bufora wymaga, żeby transakcja zsynchronizowała się z innymi transakcjami (gdyż inna transakcja może w tym samym czasie wykonywać modyfikacje na zmiennej x), transakcja T_i musi przed wykonaniem buforowania spełnić warunek dostępu. Natomiast, od razu po zbuforowaniu zmiennej transakcja może już ową zmienną zwolnić, a także, biorąc pod uwagę, że zmienna nie ulegnie modyfikacji, wykonać procedurę zatwierdzenia transakcji dla tej zmiennej.

W celu zwolnienia zmiennej jak najwcześniej buforowanie można wykonać nawet przed pierwszym odczytem zmiennej przez transakcje, więc w OptSVA+R buforowanie zmiennych tylko-do-odczytu uruchamiane jest już przy starcie transakcji. Natomiast ze względu na fakt, że buforowanie wymaga, żeby transakcja czekała na warunek dostępu, buforowanie oddelegowane jest do osobnego wątku. Pozwala to transakcji nie spowalniać wykonywania innych operacji ze względu na buforowanie zmiennych. Z drugiej strony, operacja odczytu na zmiennej tylko-do-odczytu może się odbyć tylko po tym jak zmienna zostanie zbuforowana, więc operacje na takich zmiennych czekają, aż wątek zakończy procedurę buforowania. Wykonanie buforowania w osobnym wątku pozwala transakcji na znalezienie najlepszego możliwego momentu w czasie kiedy ta procedura może być przeprowadzona, zwalniając zmienną najwcześniej jak to tylko możliwe, a jednocześnie blokując transakcje na warunku dostępu tylko, jeśli jest to absolutnie niezbędne.

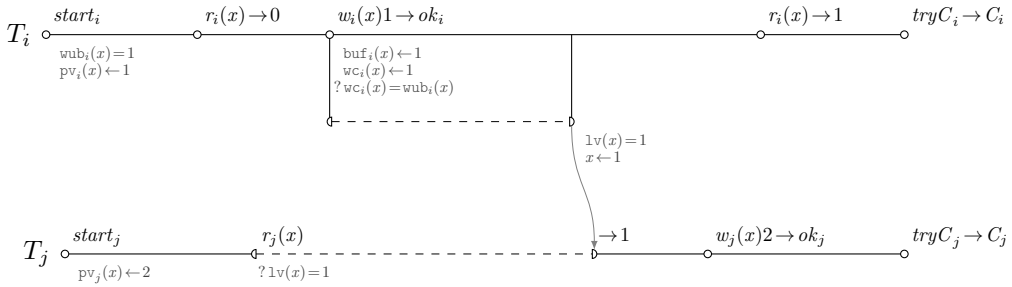
Przykład optymalizacji zmiennej tylko-do-odczytu jest zilustrowany na Rys. 11.5. W momencie startu transakcja T_j uruchamia osobny wątek celem zbuforowania zmiennej x. Operacje wykonywane w osobnym wątku są zaprezentowane poniżej linii głównego wątku transakcji. Wątek wykonuje dwie kolejne operacje: buforowanie i zatwierdzanie pojedynczej zmiennej. Buforowanie zmiennej może odbyć się dopiero, gdy spełniony jest warunek dostępu do zmiennej x, więc wątek czeka do momentu, aż T_i zwolni zmienną x. Od razu po zbuforowaniu zmiennej, wątek zwalnia zmienną x i przechodzi do wykonania procedury zatwierdzenia. Pozwala to transakcji T_k na wykonanie operacji na x jak tylko T_j ją zbuforuje, lecz nie wymaga czekania aż T_j wykona na niej wszystkie swoje operacje. Dodatkowo, T_k może także zakończyć się wcześniej, ponieważ transakcja T_j wykona procedurę zatwierdzenia dla x jeszcze przed zakończeniem samej transakcji. Warto zauważyć, że w wypadku wycofania samej T_j wartość zmiennej x nie musi być zmieniona, więc nie ma potrzeby zmuszać T_k do wycofania się, natomiast w przypadku gdyby T_i została wycofana, zarówno T_i jak i T_j będą wycofane.

Synchronizacja przy Pierwszym Odczycie

Jeśli pierwszą (lub jedyną) operacją na jakiejś zmiennej jest zapis, to zapis ten można wykonać na buforze, bez odniesienia do aktualnego stanu zmiennej. Wykonanie operacji na buforze nie wymaga synchronizacji z innymi transakcjami, więc pierwsza operacja zapisu na zmiennej może się odbyć bez sprawdzania warunku dostępu. Dodatkowo, ponieważ wszystkie kolejne operacje zapisu lub odczytu na tej zmiennej mogą być wykonane na buforze, transakcja może odsunąć sprawdzanie warunku dostępu aż do ostatniego zapisu. Ostatecznie warunek dostępu musi być spełniony celem zapisania wartości zmiennej z bufora do zmiennej właściwej, co może zostać wykonane w dowolnym momencie między



Rys. 11.6: Opóźniona synchronizacja przy pierwszym zapisie w OptSVA+R.



Rys. 11.7: Wczesne zwalnianie przy ostatnim zapisie w OptSVA+R.

ostatnim zapisem a zatwierdzeniem transakcji. Ponieważ moment jest dowolny, to zadanie czekania na warunek dostępu i uaktualnienia zmiennej z bufora jest oddelegowane do osobnego wątku.

Przykład takiego przeplotu jest pokazany na Rys. 11.6, gdzie transakcja T_i ma dostęp do zmiennej x, ale T_j jednocześnie wykonuje operację zapisu na x, stosując bufor, a po wykonaniu tej operacji uruchamia wątek, który czeka na warunek dostępu (aż T_i zwolni x) i, gdy ten jest spełniony, zapisuje wartość z bufora do zmiennej.

Optymalizacja ta pozwala transakcji na wykonanie dodatkowych operacji na danym obiekcie (używając bufora), zanim spełniony zostanie warunek dostępu dla tego obiektu. W konsekwencji oznacza to, że konfliktujące ze sobą transakcje wykonują się bardziej równolegle niż było to możliwe w przypadku SVA+R.

Wczesne Zwalnianie przy Ostatnim Zapisie

Ponieważ OptSVA+R rozróżnia odczyty od zapisów, można zastosować kolejną optymalizację. Jeśli wykonywać wszystkie zapisy na lokalnym buforze i wprowadzać modyfikacje do zmiennych dopiero przy ostatnim zapisie, to odczyty, które następują po zapisach, mogą także korzystać z bufora. W konsekwencji, jeśli zmienna została zbuforowana i wszystkie zapisy zostały wykonane, zmienna może być zwolniona, a wszystkie kolejne odczyty mogą korzystać z wartości zmiennej zapisanej w buforze. Powoduje to, że wczesne zwalnianie zmiennych jest wykonywane wcześniej, niż w innych algorytmach wersjonowania.

Scenariusz taki jest zilustrowany na Rys. 11.7. T_i po wykonaniu wszystkich swoich zapisów na zmiennej x (tzn. jednego) zwalnia zmienną i wykonuje kolejne odczyty na buforze. Pozwala to transakcji T_j na wykonywanie operacji na zmiennej x mimo tego, że T_i wykonuje dalsze odczyty na tej zmiennej (korzystając z bufora). Powoduje to, że przeplot transakcji jest bardziej zrównoleglony, a więc wykonanie jest bardziej efektywne.

Zaprezentowana optymalizacja pozwala transakcjom na zwalnianie obiektów wcze-

śniej, niż w przypadku SVA+R, ponieważ transakcje OptSVA+R nie muszą czekać na wykonanie wszystkich operacji odczytu na zmiennej, zanim zmienna zostanie zwolniona. Powoduje to zwiększenie stopnia współbieżności między transakcjami operującymi na tych samych zmiennych, co przekłada się na efektywność algorytmu. Co więcej, wykorzystanie wszystkich trzech optymalizacji powoduje, że transakcje wymagają wyłącznego dostępu do zmiennych w bardzo krótkich interwałach:

a) zmienne, które są tylko czytane, są trzymane na wyłączność tylko w momencie buforowania,

b) zmienne, do których się tylko zapisuje, są trzymane na wyłączność tylko w momencie przepisywania stanu z bufora do pamięci po ostatniej operacji zapisu,

c) zmienne, z których zarówno się czyta jak i do których się pisze, ale gdzie pierwszą operacją jest zapis, są również trzymane na wyłączność tylko w momencie przepisywana stanu z bufora do pamięci po ostatniej operacji zapisu,

d) pozostałe zmienne są trzymane na wyłączność tylko między pierwszym odczytem a ostatnim zapisem.

Własności

W przypadku OptSVA+R formalnie porównano przeploty wygenerowane przez ten algorytm z przeplotami generowanymi przez SVA+R i pokazujemy, że przeploty wygenerowane przez OptSVA+R są zawsze nie dłuższe, a zazwyczaj krótsze, niż te wytworzone przez SVA+R. W konsekwencji OptSVA+R zapewnia większą wydajność niż jego poprzednik.

Jednocześnie OptSVA+R daje te same gwarancje bezpieczeństwa co SVA+R: spełnia nieprzezroczystość do ostatniego użycia. Natomiast ze względu na sposób, w jaki OptSVA+R oddziela wykonanie operacji od efektu, jaki dana operacja ma na zmienną, staje się trudnym udowodnienie tej własności wprost. W konsekwencji wprowadzono nową technikę dowodzenia nazwaną harmonią śladów (ang. *trace harmony*). Ślad definiujemy jako historię wykonania danego programu, w której ujęte są wykonania operacji transakcyjnych oraz wykonania instrukcji niskopoziomowych, takich jak dostępy do pamięci. Jeśli operacje i instrukcje w śladach spełniają zestaw cząstkowych wymagań określonych w Definicjach 29–52, to przeplot pokazany w danym śladzie jest nieprzezroczysty do ostatniego użycia, co pokazano w Twierdzeniu 9. Następnie w Lemacie 70 i Wniosku 22 pokazujemy, że każdy ślad wygenerowany przez OptSVA+R jest harmoniczny, więc OptSVA+R jest nieprzezroczysty do ostatniego użycia.

Dodatkowo OptSVA+R jest także wolny od zakleszczeń i silnie progresywny.

Warianty

Przez analogię do SVA+R i RSVA+R wprowadzono w pracy ROptSVA+R, wariant OptSVA+R, który pozwala na zdefiniowanie klasy transakcji niechętnych, które nigdy nie są zmuszane do wycofania kosztem wykonywania operacji na zmiennych zwolnionych wcześniej. Ponadto wprowadzono OptSVA, wariant OptSVA+R, który usuwa z algorytmu możliwość wykonania ręcznej operacji wycofania, co prowadzi do całkowitego wyeliminowania wycofań w algorytmie. Algorytm ten jest prostszy od OptSVA+R, ale nadaje się do użycia jedynie w modelu systemu z transakcjami dążącymi do zatwierdzenia.

OptSVA-CF+R

Ograniczenie OptSVA+R do modelu obiektów-zmiennych pozwala na wprowadzenie dużej liczby optymalizacji względem SVA+R. Natomiast przyjęcie tego modelu powoduje,

że OptSVA+R nie jest stosowalny w wielu typach aplikacji. W szczególności, jeśli OptSVA+R miałby zostać wykorzystany do tworzenia rozproszonych systemów pamięci transakcyjnych działających w modelu przepływu sterowania, które z kolei byłyby aplikowane w złożonych systemach rozproszonych, założenia które OptSVA+R czyni względem stanu zmiennych i operacji które te zmienne wspierają są zbyt silne. W konsekwencji wprowadzono kolejny algorytm, OptSVA-CF+R, który ma zaaplikować optymalizacje wprowadzone w OptSVA+R do obiektów jednorodnych i niejednorodnych zarazem stanowiąc kompromis pomiędzy wydajnością a ogólnością stosowanego modelu.

Obiekty Niejednorodne

Jeśli przyjąć, że OptSVA-CF+R nie posiada żadnej wiedzy na temat obiektów na których operuje, żadna z wprowadzonych w OptSVA+R optymalizacji nie jest stosowalna. Nie można jednak oczekiwać od uniwersalnego rozwiązania, żeby znana i brana pod uwagę była semantyka każdej operacji każdego obiektu w modelu niejednorodnym. W konsekwencji wprowadzono kompromis, który wymaga kategoryzacji każdej operacji każdego obiektu w jednej z trzech klas:

a) operacja *odczytu* to operacja, która wykonuje dowolny kod (także z efektami ubocznymi), który może zwracać dowolną wartość, ale który nie modyfikuje stanu obiektu,

b) operacja *zapisu* to operacja, która wykonuje dowolny kod, który może modyfikować stan obiektu, ale który nie czyta stanu obiektu ani nie zwraca wartości,

c) operacja *aktualizacji* to dowolna operacja która wykonuje dowolny kod i może zarówno odczytywać, jak i modyfikować stan obiektu, oraz zwracać wartość.

Klasyfikacja ta pozwala określić semantykę operacji w stopniu pozwalającym na zaaplikowanie optymalizacji. Rozróżnienie operacji „czystego" zapisu od operacji aktualizacji pozwala nam w szczególności na aplikowanie optymalizacji związanych z buforowaniem operacji zapisu. W wypadku gdy semantyka jakiejś operacji jest nieznana, może ona zawsze być sklasyfikowana jako operacja aktualizacji bez groźby niepoprawnego wykonania.

Buforowanie Obiektów

Ponieważ operacje zapisu na zmiennych mają prostą semantykę, która wiąże się z nadpisaniem całego stanu zmiennej, tj. jej wartości, możliwe jest wykonywanie operacji zapisu na „pustych" buforach. Nie jest to możliwe jednak w przypadku obiektów, których stan jest złożony: po wykonaniu operacji zapisu nie ma pewności, że następna operacja odczytu będzie korzystać z tego samego pola, które zostało zapisane przez operację zapisu.

Celem rozwiązania tego problemu wprowadzono kolejny typ bufora: dziennik (ang. *log buffer*). Dziennik ma taki sam interfejs jak obiekt, z którym jest związany, ale operacja zlecona do wykonania nie zostaje wykonana *de facto*, a jedynie dodana do listy operacji do wykonania. Następnie taki dziennik może być zaaplikowany do obiektu, z którym jest związany, co spowoduje wykonanie wszystkich zleconych operacji. Dziennik może być użyty do odsuwania wykonań operacji zapisu i nie wymaga on uprzedniej inicjalizacji, tak jak jest to konieczne w przypadku standardowych buforów.

Asynchroniczne Buforowanie

OptSVA-CF+R obsługuje obiekty tylko-do-odczytu przez analogię do OptSVA+R. Podobnie jest w przypadku buforowania obiektów, na których wykonywany jest zapis, chociaż w tym wypadku procedura jest bardziej złożona i konserwatywna ze względu na użycie dwóch typów buforów. W wypadku, gdy pierwsza operacja na danym typie obiektu jest zapisem, zapis ten jest wykonywany bez synchronizacji na dzienniku dla

tego obiektu. Jeśli po zapisie występują kolejne zapisy, to one także są kierowane do dziennika obiektu. Natomiast, jeśli po zapisach występuje aktualizacja lub odczyt, to należy dziennik zaaplikować do obiektu, żeby uzyskać zaktualizowany stan. Oznacza to, że OptSVA-CF+R w takim wypadku musi dokonać synchronizacji i czekać na warunku dostępu do danej zmiennej, co nie jest konieczne w przypadku OptSVA+R. Dodatkowo, w przeciwieństwie do OptSVA+R, gdzie zapisy zawsze wykonywane są przy użyciu bufora, transakcje OptSVA-CF+R, które wykonują zapisy, wykonują je bezpośrednio na obiekcie właściwym, jeśli warunek dostępu został uprzednio spełniony podczas odczytu lub aktualizacji. W końcu, OptSVA-CF+R, podobnie jak OptSVA+R, wykonuje kopię obiektu do bufora przy ostatnim zapisie lub aktualizacji (ostatniej potencjalnej modyfikacji dowolnego typu). W wypadku, gdy na obiekcie wykonywane były tylko zapisy, buforowanie wymaga także, aby dziennik był zaaplikowany do obiektu przed skopiowaniem go do bufora.

Różnice między optymalizacjami zaaplikowanymi w OptSVA+R i OptSVA-CF+R są ilustruje Rys. 11.8. Pokazano tam wykonanie tego samego programu przy użyciu OptSVA+R (Rys. 11.8a) i OptSVA-CF+R (Fig. 11.8b). W obu przykładach $\lceil x \rfloor$ jest komórką z referencją; prostym obiektem który, zawiera pojedyncze pole stanowiące jego stan i interfejs analogiczny do zmiennej. W obu zaprezentowanych przeplotach transakcja T_i uruchamia się jako pierwsza, ale wykonuje operację zapisującą 2 do zmiennej x (obiektu $\lceil x \rfloor$) dopiero po długim opóźnieniu. W międzyczasie transakcja T_j wykonuje własny zapis do x ($\lceil x \rfloor$), zapisując 1. Ponieważ jest to początkowy zapis, to OptSVA+R wykonuje go na buforze $\texttt{buf}_j(x)$, a OptSVA-CF+R na dzienniku $\texttt{log}_j(\lceil x \rfloor)$, więc żaden z algorytmów nie powoduje, że T_j czeka na T_i. Następnie T_j wykonuje operację odczytu na x ($\lceil x \rfloor$). W OptSVA+R ta operacja jest wykonana na buforze $\texttt{buf}_j(x)$, co nie wymaga synchronizacji, więc operacja wykonuje się bez czekania. Natomiast w OptSVA-CF+R operacja odczytu nie może być wykonana na dzienniku, ponieważ dziennik nie zna stanu obiektu. Niezbędnym więc jest w OptSVA-CF+R, żeby T_j w tym momencie czekała, aż T_i nie zwolni obiektu $\lceil x \rfloor$. Dopiero wtedy T_j może zaaplikować $\texttt{log}_j(\lceil x \rfloor)$ do $\lceil x \rfloor$ i wykonać odczyt. W efekcie T_j wykonuje się dłużej w OptSVA-CF+R niż w OptSVA+R.

Przykład pokazuje więc, że generalizacja modelu niesie ze sobą potencjalny spadek efektywności wykonania. Jest to nieuniknione, biorąc pod uwagę złożoność obsługiwanych obiektów. Z drugiej strony, OptSVA-CF+R wciąż cechuje się bardzo wysokim stopniem zrównoleglenia transakcji skonfliktowanych i, jak pokazano poniżej, osiąga wysoką wydajność w praktyce.

Warianty

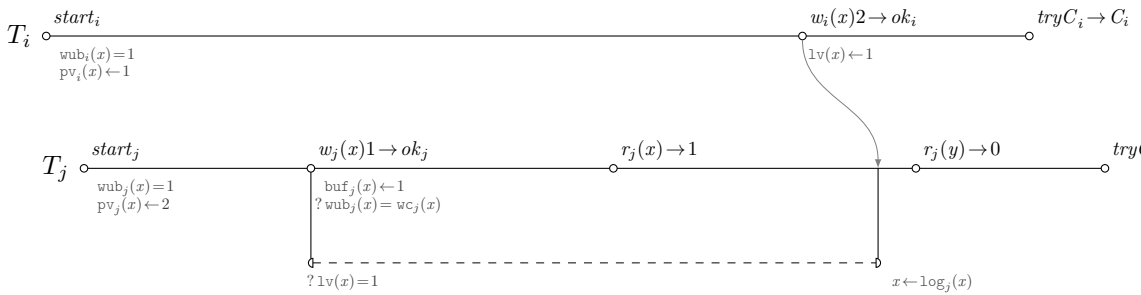
Tak samo, jak w przypadku OptSVA+R i SVA+R, OptSVA-CF+R ma wariant z transakcjami niechętnymi (ROptSVA-CF+R) oraz wariant działający w modelu z transakcjami dążącymi do zatwierdzenia (OptSVA-CF).

Własności

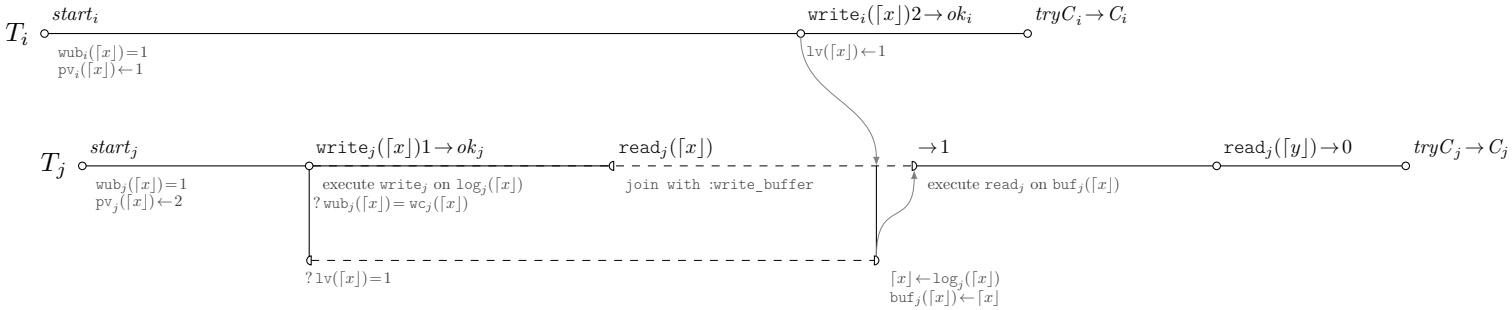
Pokazujemy, że OptSVA-CF+R jest nieprzezroczysty do ostatniego użycia przez analogię do OptSVA+R, oraz że jest on wolny od zakleszczeń i silnie progresywny.

Podsumowanie

Wprowadzone algorytmy są podsumowane w Tablicy 11.4. Wszystkie algorytmy są pesymistyczne, blokujące i pozbawione zakleszczeń. Algorytmy z transakcjami niechętnymi zapewniają poprawne wykonanie transakcji niechętnych ze względu na operacje niewycofywalne, a algorytmy bez wycofań zapewniają poprawne wykonanie wszystkich transakcji



(a) OptSVA+R.



(b) OptSVA-CF+R.

Rys. 11.8: Obsługa buforowania w OptSVA+R vs OptSVA-CF+R.

Algorytm	**Modyfikacje**	**Wycofania**	**A priori**	**Obiekty**	**Bezpieczeństwo**	**Zwalnianie zasobów**	**Operacje niewycofywalne**
SVA	p. wykonania	bez wycofania	*ASet*, `supr`	niejednorodne	opaque-equivalent	tak	$T_i \in \mathbb{T}$
SVA+R	p. wykonania	dowolne, przy kaskadzie	*ASet*, `supr`	niejednorodne	*last-use opaque*	tak	$\varnothing$
RSVA+R	p. wykonania	dowolne, przy kaskadzie	*ASet*, `supr`, $\mathbb{R}$	niejednorodne	*last-use opaque*	tak	$T_i \in \mathbb{R}$
OptSVA	p. zatwierdzeniu	bez wycofania	*ASet*, `wub`, `rub`,	zmienne	*last-use opaque**	tak	$T_i \in \mathbb{T}$
OptSVA+R	p. zatwierdzeniu	dowolne, przy kaskadzie	*ASet*, `wub`, `rub`,	zmienne	*last-use opaque*	tak	$\varnothing$
ROptSVA+R	p. zatwierdzeniu	dowolne, przy kaskadzie	*ASet*, `wub`, `rub`, $\mathbb{R}$	zmienne	*last-use opaque*	tak	$T_i \in \mathbb{R}$
OptSVA-CF	p. zatwierdzeniu	bez wycofania	*ASet*, `wub`, `rub`, klasy operacji	dowolne	*last-use opaque**	tak	$T_i \in \mathbb{T}$
OptSVA-CF+R	p. zatwierdzeniu	dowolne, przy kaskadzie	*ASet*, `wub`, `rub`, klasy operacji	dowolne	*last-use opaque*	tak	$\varnothing$
ROptSVA-CF+R	p. zatwierdzeniu	dowolne, przy kaskadzie	*ASet*, `wub`, `rub`, $\mathbb{R}$, klasy operacji	dowolne	*last-use opaque*	tak	$T_i \in \mathbb{R}$

Tablica 11.4: Podsumowanie wprowadzonych algorytmów wersjonowania.

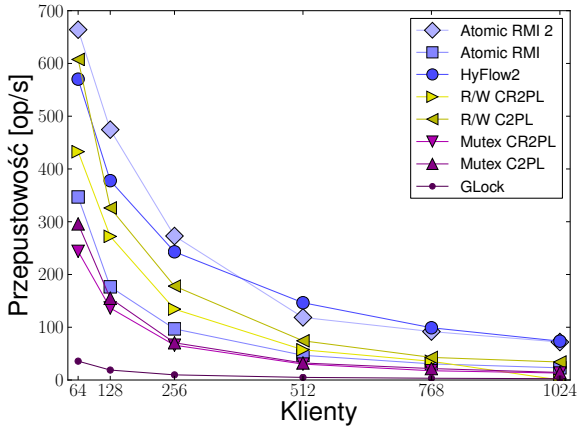
ze względu na operacje niewycofywalne. Dodatkowo, algorytmy bez wycofania, poza spełnianiem nieprzezroczystości do ostatniego użycia, generują przeploty nierozróżnialne od przeplotów nieprzezroczystych pod względem efektów.

Implementacje

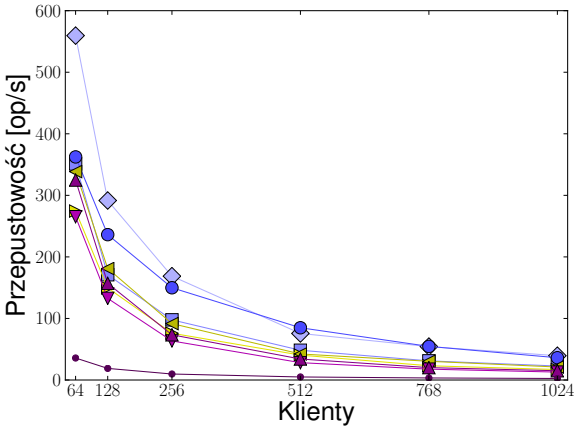
Zaimplementowano dwa z zaproponowanych algorytmów: SVA+R i OptSVA-CF+R (wraz z wariantami pozwalającymi na definicje transakcji niechętnych) jako systemy rozproszonej pamięci transakcyjnej oparte na Java RMI, nazwane odpowiednio Atomic RMI i Atomic RMI 2. Implementacje dostarczają mechanizmów niezbędnych do praktycznego funkcjonowania systemu w środowisku rozproszonym, takich jak obsługa awarii częściowych, serializacja obiektów, implementacja buforów, oraz instrumentacja kodu w sposób ukrywający algorytmy sterowania współbieżnością przed programistą. Architektura zaimplementowanych systemów rozproszonej pamięci transakcyjnej rozszerza architekturę Java RMI o obiekty pełnomocników (ang. *proxy*), które implementują algorytmy sterowania współbieżnością przechwytując komunikację między klientami-transakcjami a obiektami współdzielonymi.

Implementacje zostały przetestowane pod względem wydajności przy użyciu programu wzorcowego (ang. *benchmarks*) EigenBench [47] przystosowanego do ewaluacji rozproszonej pamięci transakcyjnej. Ewaluacja porównuje Atomic RMI i Atomic RMI 2 z wysokiej klasy optymistycznym systemem rozproszonej pamięci transakcyjnej, HyFlow2 [86]. Dodatkowo porównano zaimplementowane w ramach pracy systemy z implementacjami algorytmów blokowania dwufazowego opartymi na zamkach z rozróżnieniem operacji odczytu i zapisu (R/W) lub traktującymi operacje jednakowo (Mutex), a także z implementacją zamka globalnego. Implementacje przetestowano na 16-węzłowym klastrze obliczeniowym połączonym siecią o prędkości 1Gb. Każdy węzeł posiada dwa czterordzeniowe procesory quad-core Intel Xeon L3260 taktowane 2.83 GHz z 4 GB pamięci RAM. Na każdym węźle działa system operacyjny OpenSUSE 13.1 (jądro 3.11.10, architektura x86_64). Użyto języka Groovy w wersji 2.3.8 oraz 64-bitowej Java HotSpot(TM) JVM w wersji 1.8 (build 1.8.0_25-b17).

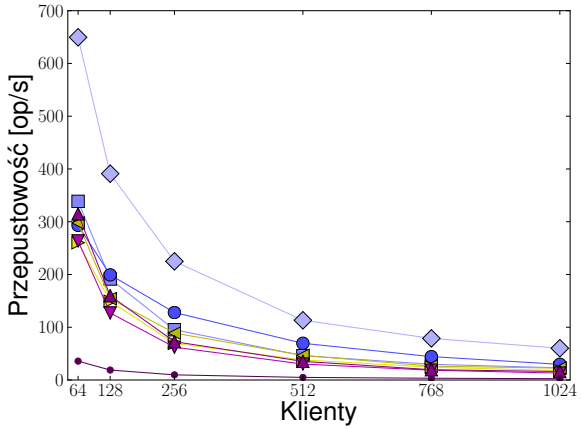
Wyniki ewaluacji pokazane są na Rys. 11.9–11.10. Miarą wydajności jest przepustowość mierzona w liczbie operacji wykonanych na sekundę. Rys. 11.9 pokazuje, że wraz ze wzrostem liczby klientów (a więc i ze wzrostem współzawodnictwa o zasoby) spada przepustowość wszystkich systemów. Spadek wydajności jest szczególnie stromy do liczby 256 klientów i stabilizuje się dla 1024 klientów. Wszystkie algorytmy przewyższają wydajnością wykonanie sekwencyjne przy użyciu zamka globalnego. W scenariuszu, gdzie stosunek odczytów do zapisów wynosi 90%, HyFlow2 i Atomic RMI 2 wykonują się z wydajnością znacznie przewyższającą wydajność pozostałych systemów o 9–267%, lecz porównywalną względem siebie. W pozostałych dwóch scenariuszach wszystkie implementacje tracą na efektywności, z wyjątkiem Atomic RMI 2, który działa 9–359% lepiej od pozostałych implementacji (w tym HyFlow2). Różnica wydajności jest wynikiem optymalizacji operacji zapisu w Atomic RMI 2, która pozwala na skracanie przeplotów transakcji, gdy występują długie sekwencje operacji zapisu. Pozostałe implementacje nie optymalizują zapisów w takim stopniu. W szczególności HyFlow2 i 2PL opierają się głównie o zrównoleglenie odczytów. Degradacja wydajności Atomic RMI 2 jest wyjaśniona potrzebą zarządzania wątkami w celu obsługi asynchronii lokalnej. Powoduje to, że każdy węzeł musi obsłużyć więcej działających jednocześnie wątków niż w innych implementacjach. Spośród pozostałych implementacji, warianty C2PL działają zawsze lepiej, niż odpowiadające im warianty CS2PL, natomiast warianty R/W działają lepiej niż Mutex.



(a) 90% odczytów, 10% zapisów.

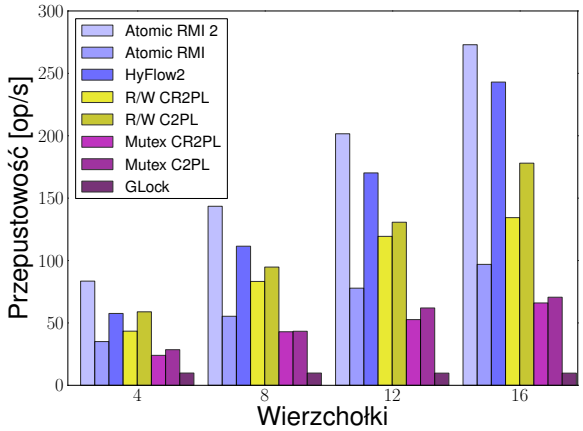


(b) 50% odczytów, 50% zapisów.

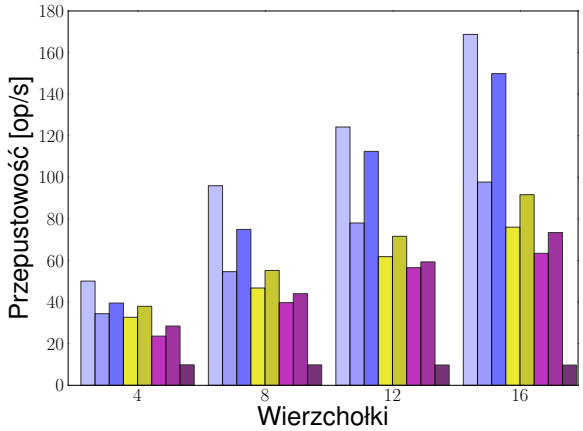


(a) 10% odczytów, 90% zapisów.

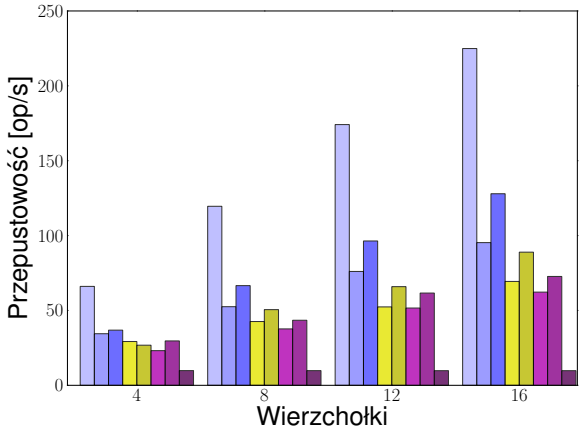
Rys. 11.9: Przepustowość (ang. *throughput*) vs liczba klientów.



(b) 90% odczytów, 10% zapisów, 10 tablic.



(c) 50% odczytów, 50% zapisów, 10 tablic.



(d) 10% odczytów, 90% zapisów, 10 tablic.

Rys. 11.10: Przepustowość (ang. *throughput*) vs liczba węzłów.

Atomic RMI działa porównywalnie pod względem wydajności do C2PL i zdecydowanie gorzej niż Atomic RMI 2.

Rys. 11.10 pokazuje zmianę w przepustowości wraz z dodawaniem nowych węzłów do systemu (ale przy stałej przepustowości). Wraz z poszerzaniem systemu o dodatkową moc obliczeniową, przepustowość systemu wzrasta, dzięki powiększającej się liczbie potencjalnie wykonywanych równolegle operacji. Porównanie pokazuje, że Atomic RMI 2 zdecydowanie przewyższa wydajnością pozostałe implementacje, w tym także Atomic RMI i HyFlow2. Różnica ta uwydatnia się znów w wypadku gdy scenariusz jest zdominowany przez operacje zapisu, które są lepiej zoptymalizowane w Atomic RMI 2 niż w innych implementacjach. Ze względu na stosunkowo dużą liczbę obiektów współdzielonych w systemie (10 tablic na każdym z węzłów), sposób generowania transakcji powoduje, że Atomic RMI 2 jest także w stanie często zwalniać obiekty wcześnie, prowadząc do większego współczynnika zrównoleglenia transakcji. Spodziewać się można, że gdyby liczba obiektów była ograniczona, różnica między HyFlow2 i Atomic RMI 2 wyrównałaby się w przypadku scenariuszy analogicznych do Rys. 11.9b i 11.9c.

Istotnym jest także, że w ewaluacji liczba transakcji wycofanych przez Atomic RMI i Atomic RMI 2 wynosiła 0, podczas gdy HyFlow2 musiał wycofać i ponowić 60–89% transakcji (w zależności od scenariusza). Oznacza to, że w praktyce Atomic RMI 2 zachowuje się bezpiecznie względem operacji niewycofywalnych, podczas gdy w HyFlow2 mogą one być wykonane wielokrotnie.

Statyczna Analiza i Prekompilator

Dodatkowym elementem prezentowanym w pracy, poza algorytmami i ich własnościami, są dwa narzędzia mające na celu poprawienie praktyczności i wydajności zaprezentowanych systemów: prekompilator i moduł szeregowania transakcji.

Ze względu na fakt, że algorytmy wersjonowanie wymagają znajomości *a priori* obiektów używanych przez poszczególne transakcje, zadaniem prekompilatora jest analiza statyczna kodu każdej z transakcji i wydobycie tej informacji w sposób automatyczny. Dodatkowo, prekompilator bada kod, poszukując poszczególnych wywołań operacji na obiektach współdzielonych i oblicza przybliżone suprema dla każdego z obiektów wewnątrz transakcji. Prekompilator uwalnia programistę od potrzeby przygotowywania tej informacji ręcznie.

Podsumowanie

Aby udowodnić główną tezę niniejszej pracy zostały przeanalizowane istniejące własności bezpieczeństwa oraz ich przydatność w kontekście pamięci transakcyjnej z wczesnym zwalnianiem zmiennych (Rozdział 3). Następnie wprowadzone zostały własności bezpieczeństwa, które mają praktyczne zastosowanie dla tego typu pamięci transakcyjnych: nieprzezroczystość do ostatniego użycia i silna nieprzezroczystość do ostatniego użycia (Rozdział 5).

W dalszej kolejności opisano istniejące pesymistyczne algorytmy sterowania współbieżnością dla pamięci transakcyjnej, zarówno te rozproszone jak i nierozproszone, oraz optymistyczne algorytmy sterowania współbieżnością w rozproszonej pamięci transakcyjnej, a także algorytmy używające wczesnego zwalniania zasobów (Rozdział 4). Na

podstawie tej analizy wybrano rodzinę algorytmów wersjonowania jako podstawę do dalszych badań. Następnie zmodyfikowano algorytm SVA, eliminując zależność od globalnego zamka, oraz pozwalając na swobodne wycofywanie transakcji (Rozdział 6, Sekcje 6.1 i 6.2).

Dalej zaprezentowano algorytmy OptSVA+R i OptSVA-CF+R oraz ich warianty. Są to nowatorskie algorytmy pesymistycznego sterowania współbieżnością rozszerzające istniejące algorytmy wersjonowania i stosujące optymalizacje, które pozwalają na wykonanie skonfliktowanych transakcji z większym stopniem zrównoleglenia niż było to możliwe do tej pory (Rozdział 6, Sekcje 6.3–6.4). Opracowane pesymistyczne algorytmy sterowania współbieżnością zostały użyte do implementacji dwóch systemów rozproszonej pamięci transakcyjnej. W pracy pokazano, że system oparty na OptSVA-CF+R i ROptSVA-CF+R jest w stanie uzyskać lepszą wydajność, niż wysokiej klasy implementacja optymistycznej pamięci transakcyjnej, jednocześnie nie powodując konieczności wycofania transakcji (Rozdział 8).

Pokazano także, że choć zastosowane algorytmy reprezentują bardziej ogólne i wydajne podejście, zachowują one silne gwarancje bezpieczeństwa. Zostało to w pracy zademonstrowane przeprowadzając formalne dowody poprawności tych algorytmów, co wymagało wprowadzenia nowych technik dowodzenia poprawności algorytmów sterowania współbieżnością (Rozdział 7).

Ponadto, zaproponowano dodatkowy praktyczny moduł dla opracowanych systemów pamięci transakcyjnej, tj. prekompilator, który automatycznie generuje wiedzę *a priori* używaną do wczesnego zwalniania obiektów w algorytmach wersjonowania przez statyczną analizę kodu programu (Rozdział 9).

Dowód Tezy

Jako dowód postawionej w pracy tezy:

Przedstawiono Atomic RMI 2, system rozproszonej pamięci transakcyjnej w modelu przepływu sterowania, implementujący pesymistyczne algorytmy OptSVA-CF+R i ROptSVA-CF+R.

a) W Sekcji 8.2.2 pokazano, że Atomic RMI 2 przewyższa wydajnością wysokiej klasy optymistyczny system rozproszonej pamięci transakcyjnej, a więc Atomic RMI 2 jest systemem wydajnym.

b) Twierdzeniem 10 pokazano, że OptSVA-CF+R jest nieprzezroczysty do ostatniego użycia, Twierdzeniem 4 pokazano, że jest on silnie progresywny, a Twierdzeniem 3 pokazano, że jest on pozbawiony zakleszczeń. W konsekwencji OptSVA-CF+R spełnia silne gwarancje bezpieczeństwa, postępu i żywotności.

c) W Sekcji 8.2.2 pokazano, że OptSVA-CF+R w praktyce nie doprowadza do wycofań transakcji, a więc operacje niewycofywalne wykonywane są (w praktyce) poprawnie. Ponadto, ROptSVA-CF+R całkowicie wyklucza możliwość wycofywania transakcji niechętnych, a więc operacje niewycofywalne zawsze będą wykonywane poprawnie w przypadku ogólnym, zakładając, że będą wykonywane w ramach transakcji niechętnych.

d) OptSVA-CF+R wspiera dowolne wycofywanie transakcji, i operuje na niejednorodnym modelu obiektowym. Dodatkowo, nie ma pojedynczego punktu awarii. Ponadto, informacje niezbędne do działania tego algorytmu mogą być wygenerowane *a priori* przez prekompilator. W konsekwencji OptSVA-CF+R można uznać za algorytm mający praktyczne zastosowanie.

Reasumując, teza jest spełniona. □

Bibliography

[1] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *Proceedings of DISC'12: the 26th International Symposium on Distributed Computing*, Oct. 2012.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 2nd edition, Aug. 2006.

[3] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Lee-TM: A non-trivial benchmark for transactional memory. In *Proceedings of ICA3PP'08: the 8th International Conference on Algorithms and Architectures for Parallel Processing*, June 2008.

[4] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of WOMPAT'01: the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, July 2001.

[5] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *Proceedings of PODC'13: the 32nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2013.

[6] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. Safety of live transactions in transactional memory: TMS is necessary and sufficient. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, 2014.

[7] H. Attiya and S. Hans. Transactions are back—but how different they are? In *Proceedings of TRANSACT'14: the 7th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2014.

[8] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *Proceedings of ICDCS'13: the 33rd International Conference on Distributed Computing Systems*, July 2013.

[9] H. Attiya and E. Hillel. Single-version STMs can be multi-version permissive. In *Proceedings of ICDCN'11: the 12th International Conference on Distributed Computing and Networking*, Jan. 2011.

[10] H. Avni, S. Dolev, P. Fatourou, and E. Kosmas. Abort free semantic TM by depedency aware scheduling of transactional instructions. In *Proceedings of NETYS'14: the International Conference on Networked Systems*, May 2014.

[11] J. Baranowski, P. Kobyliński, K. Siek, and P. T. Wojciechowski. Helenos: A realistic benchmark for distributed transactional memory. *Journal of Systems and Software*, Mar. 2016. `arXiv:1603.07899 [cs.DC]` (revision).

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[13] A. Bieniusa, A. Middelkoop, and P. Thiemann. Brief announcement: Actions in the twilight—concurrent irrevocable transactions and inconsistency repair. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.

[14] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of PPoPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[15] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9), Sept. 1991.

[16] W. Cellary, E. Gelenbe, and T. Morzy. *Concurrency control in distributed database systems.* North-Holland, 1988.

[17] J. C. Corbett and et al. Spanner: Google's globally-distributed database. In *Proceedings of OSDI'12: the 10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012.

[18] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proceedings of PRDC'13: the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, Nov. 2009.

[19] D. Cunningham, K. Gudka, and S. Eisenbach. Keep off the grass: Locking the right path for atomicity. In *Proceedings of CC'08: the 17th International Conference on Compiler Construction, part of Part of ETAPS'08: the Joint European Conferences on Theory and Practice of Software*, Mar. 2008.

[20] D. Dice, A. Matveev, and N. Shavit. Implicit privatization using private transactions. In *Proceedings of TRANSACT'10: the 5th ACM SIGPLAN Workshop on Transactional Computing*, Apr. 2010.

[21] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of DISC'06: the 20th International Symposium on Distributed Computing*, Sept. 2006.

[22] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5), Sept. 2013.

[23] S. Dolev, P. Fatourou, and E. Kosmas. Abort free semantic TM by depedency aware scheduling of transactional instructions. In *Proceedings of TRANSACT'13: the 8th ACM SIGPLAN Workshop on Transactional Computing*, Mar. 2013.

[24] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of PODC'08: the 28th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 2008.

[25] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing vs curing: Avoiding conflicts in transactional memories. In *Proceedings of PODC'09: the 28th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 2009.

[26] D. Dziuma, P. Fatourou, and E. Kanellou. Consistency for transactional memory computing. *Bulletin of the EATCS*, 113, 2014.

[27] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proceedings of WCET '07: the 7th International Workshop on Worst-Case Execution Time Analysis*, July 2007.

[28] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of DISC'09: the 23rd International Symposium on Distributed Computing*, Sept. 2009.

[29] C. Ferdinand and R. Heckmann. AiT: Worst-case execution time prediction by static program analysis. In *Proceedings of WCC '04: the 18th International Federation for Information Processing World Computer Congress*, Aug. 2004.

[30] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT'01: the 1st International Workshop on Embedded Software*, Oct. 2001.

[31] É. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *Proceedings of TOOLS '98: the 26th International Conference on Technology of Object-Oriented Languages and Systems*, Aug. 1998.

[32] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *Proceedings of PPoPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008.

[33] R. Guerraoui and M. Kapałka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.

[34] R. Guerraoui, M. Kapałka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of EuroSys'07: the 2nd ACM SIGOPS European Conference on Somputer Systems*, June 2007.

[35] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *Proceedings of WORDS'05: the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Sept. 2005.

[36] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1), Jan. 1988.

[37] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of RTAS'08: the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2008.

[38] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3), 2005.

[39] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of OOPSLA'03: the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[40] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2nd edition, 2010.

[41] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of PPoPP'05: the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[42] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proceedings of ESOP'86: the 1st European Symposium on Programming*, 1986.

[43] M. Herlihy, V. Luchangco, M. Moir, and I. W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC'03: the 22nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[44] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of ISCA'93: the 20th International Symposium on Computer Architecture*, pages 289–300, May 1993.

[45] M. Hicks, J. S. Foster, and P. Prattikakis. Lock inference for atomic sections. In *Proceedings of TRANSACT'06: the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[46] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proceedings of EUSIPCO 2000: the 10th European Signal Processing Conference*, Sept. 2000.

[47] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proceedings of IISWC'10: the IEEE International Symposium on Workload Characterization*, 2010.

[48] D. Imbs, J. R. de Mendivil, and M. Raynal. On the consistency conditions or transactional memories. Technical Report 1917, IRISA, Dec. 2008.

[49] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *Proceedings of ICPP'08: the 37th IEEE International Conference on Parallel Processing*, Sept. 2008.

[50] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2), Mar. 1977.

[51] Y.-F. Lee, B. G. Ryder, and M. E. Fiuczynski. Region analysis: A parallel elimination method for data flow analysis. *IEEE Transactions on Software Engineering*, 21, Nov. 1995.

[52] M. Lesani and J. Palsberg. Decomposing opacity. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, 2014.

[53] Y.-T. S. Li and S. Malik. *Performance analysis of real-time embedded software*. Springer, Nov. 1998.

[54] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3), 1999.

[55] N. Lynch. Distributed algorithms. 1996.

[56] A. Matveev and N. Shavit. Towards a fully pessimistic STM model. In *Proceedings of TRANSACT '12: the 7th ACM SIGPLAN Workshop on Transactional Computing*, Aug. 2012.

[57] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *Proceedings of POPL'06: the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2006.

[58] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of IISWC'08: the IEEE International Symposium on Workload Characterization*, Sept. 2008.

[59] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. User-definable resource usage bounds analysis for Java bytecode. *Electronic Notes in Theoretical Computer Science*, 253(5), Dec. 2009.

[60] C. H. Papadimitrou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.

[61] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of OSDI '10: 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[62] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.

[63] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via CFL reachability. In *Proceedings of SAS'06: the 13th International Symposium on Static Analysis*, Aug. 2006.

[64] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8, 1992.

[65] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *Proceedings of PPoPP'09: the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2009.

[66] H. E. Ramadan, I. Roy, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of MICRO'08: the 41st annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2008.

[67] M. M. Saad and R. B. Transactional forwarding: Supporting highly-concurrent STM in asynchronous distributed systems. In *Proceedings of SBAC-PAD'12: the 24th IEEE International Symposium on Computer Architecture and High Performance Computing*, Oct. 2012.

[68] M. M. Saad and B. Ravindran. HyFlow: A high performance distributed transactional memory framework. In *Proceedings of HPDC '11: the 20th International Symposium on High Performance Distributed Computing*, June 2011.

[69] M. M. Saad and B. Ravindran. Transactional forwarding algorithm. Technical report, Department of Electrical and Computer Engineering, Virginia Tech., Jan. 2011.

[70] I. W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of PODC'05: the 24th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2005.

[71] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of PODC'95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1995.

[72] K. Siek and P. T. Wojciechowski. Brief announcement: Statically computing upper bounds on object calls for pessimistic concurrency control. In *Proceedings of EC^2'10: the Workshop on Exploiting Concurrency Efficiently and Correctly*, July 2010.

[73] K. Siek and P. T. Wojciechowski. A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java. In *Proceedings of FMICS'12: the 17th International Workshop on Formal Methods for Industrial Critical Systems*, number 7437 in Lecture Notes in Computer Science, Aug. 2012.

[74] K. Siek and P. T. Wojciechowski. Brief announcement: Towards a fully-articulated pessimistic distributed transactional memory. In *Proceedings of SPAA'13: the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, July 2013.

[75] K. Siek and P. T. Wojciechowski. Atomic RMI: a distributed transactional memory framework. In *Proceedings of HLPP'14: the 7th International Symposium on High-level Parallel Programming and Applications*, July 2014.

[76] K. Siek and P. T. Wojciechowski. Brief announcement: Relaxing opacity in pessimistic transactional memory. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, Oct. 2014.

[77] K. Siek and P. T. Wojciechowski. Zen and the art of concurrency control: An exploration of tm safety property space with early release in mind. In *Proceedings of WTTM'14: the 6th Workshop on the Theory of Transactional Memory*, July 2014.

[78] K. Siek and P. T. Wojciechowski. Atomic RMI: A distributed transactional memory framework. *International Journal of Parallel Programming*, 44(3), June 2015.

[79] K. Siek and P. T. Wojciechowski. Last-use opacity: A strong safety property for transactional memory with early release support. *ACM Transactions on Programming Languages and Systems*, June 2015. `arXiv:1506.06275 [cs.DC]` (submitted).

[80] K. Siek and P. T. Wojciechowski. Proving opacity of transactional memory with early release. foundations of computing and decision sciences. *Foundations of Computing and Decision Sciences*, 40(4), Dec. 2015.

[81] K. Siek and P. T. Wojciechowski. Transactions scheduled while you wait. *Journal of Grid Computing*, Oct. 2015. (submitted).

[82] K. Siek and P. T. Wojciechowski. Atomic RMI 2: Highly parallel pessimistic distributed transactional memory. *Transactions on Parallel and Distributed Systems*, Apr. 2016. `arXiv:1606.03928 [cs.DC]` (submitted).

[83] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Proceedings of WTW'06: the Workshop on Transactional Memory Workloads*, June 2006.

[84] J. Staschulat, J. Braam, R. Ernst, T. Rambow, R. Schlor, and R. Busch. Cost-efficient worst-case execution time analysis in industrial practice. In *Proceedings of ISoLA'06: the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Nov. 2006.

[85] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18:157–179, May 2000.

[86] A. Turcu, B. Ravindran, and R. Palmieri. HyFlow2: A high performance distributed transactional memory framework in scala. In *Proceedings of PPPJ'13: the 10th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, Sept. 2013.

[87] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot—a Java optimization framework. In *Proceedings of CASCON'99: the Conference of the Centre for Advanced Studies on Collaborative Research*, Nov. 1999.

[88] R. Vallée-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical Report 1998-4, McGill University, July 1998.

[89] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1), Jan. 2009.

[90] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2), Apr. 1989.

[91] G. Weikum and G. Vossen. *Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers, 2002.

[92] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of SPAA'08: the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.

[93] R. Wilhelm. Determining bounds on execution times. In *Handbook on Embedded Systems*, chapter 14. CRC Press, 2006.

[94] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), Apr. 2008.

[95] A. Wojciechowski and K. Siek. Barcode scanning from mobile-phone camera photos delivered via MMS: Case study. In *Advances in Conceptual Modeling—Challenges and Opportunities*, volume 5232 of *Lecture Notes in Computer Science*, Oct. 2008.

[96] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proceedings of PPDP'05: the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.

[97] P. T. Wojciechowski. *Language design for atomicity, declarative synchronization, and dynamic update in communicating systems.* Publishing House of Poznań University of Technology, 2007.

[98] P. T. Wojciechowski. Extending atomic tasks to distributed atomic tasks. In *Proceedings of EC^2'10: the Workshop on Exploiting Concurrency Efficiently and Correctly*, July 2008.

[99] P. T. Wojciechowski, T. Kobus, and M. Kokociński. Model-driven comparison of state-machine–based and deferred-update replication schemes. In *Proceedings of SRDS'12: the 31st IEEE International Symposium on Reliable Distributed Systems*, Oct. 2012.

[100] P. T. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS'04: the 18th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2004.

[101] P. T. Wojciechowski and K. Siek. Having your cake and eating it too: Combining strong and eventual consistency. In *Proceedings of PaPEC'14: the 1st Workshop on the Principles and Practice of Eventual Consistency*, Apr. 2014.

[102] P. T. Wojciechowski and K. Siek. The optimal pessimistic transactional memory algorithm, May 2016. `arXiv:1605.010361 [cs.DC]` (in submission).

[103] F. Wolf, R. Ernst, and W. Ye. Path clustering in software timing analysis. *IEEE Transactions on Very Large Scale Integrated Systems*, 9, Dec. 2001.

[104] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of ISCA'95: the 22nd Annual International Symposium on Computer Architecture*, May 1995.

[105] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of SPACC'08: the 20thACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.

A
Proofs

Property Strength

Last-use Opacity

Below we compare last-use opacity to other properties and consistency conditions to determine their relative strength.

Opacity

Opacity is strictly stronger than last-use opacity.

Lemma 71. *For any history S and transaction $T_i \in S$, if $Vis(S, T_i)$ is legal, then $LVis(S, T_i)$ is legal.*

Proof. By definition of $Vis(S, T_i)$, if operation $op \in Vis(S, T_i)$, then $op \in Vis(S, T_i)$ only if $op \in H|T_j$ and either $i = j$ or $T_j \prec_S T_i$ and T_j is committed. By definition of $LVis(S, T_i)$, given transactions T_i, T_j and operation $op \in S|T_j$, if $i = j$ or $T_j \prec_S T_i$ and T_j is committed, then $S|T_j \subseteq LVis(S, T_i)$. Therefore $LVis(S, T_i) \equiv Vis(S, T_i)$. Since $Vis(S, T_i)$ and $LVis(S, T_i)$ preserve the order of operations in S, then $LVis(S, T_i) = Vis(S, T_i)$. Hence, if $Vis(S, T_i)$ is legal, then $LVis(S, T_i)$ is legal. □

Lemma 72. *Any final-state last-use opaque history H is final-state last-use opaque.*

Proof. From Def. 12, for any final-state opaque history H, there is a sequential history $S \equiv Compl(H)$ s.t. S preserves the real time order of H and every transaction T_i in S is legal in S. Thus, for every transaction T_i in S $Vis(S, T_i)$ is legal. From the definition of completion, any T_i is either committed or aborted in $Compl(H)$ and therefore likewise completed or aborted in S. If T_i is committed in S, then it is legal in S, so $Vis(S, T_i)$ is legal, and therefore T_i is last-use legal in S. If T_i is aborted in S, then it is legal in S, so $Vis(S, T_i)$ is legal, and therefore, from Lemma 71, $LVis(S, T_i)$ is also legal, so T_i is last-use legal in S. Given that all transactions in S are last-use legal in S, then, from Def. 23, H is final-state last-use opaque. □

Lemma 73. *Any opaque history H is last-use opaque.*

Proof. If H is opaque, then, from Def. 13, any prefix P of H is final-state opaque. Since any prefix P of H is final-state opaque, then, from Lemma 72, any P is also final-state last-use opaque. Then, by Def. 24 H is last-use opaque. □

Serializability

Last-use opacity is strictly stronger than serializability.

Lemma 74. *Any last-use opaque history H is serializable.*

Proof. From Lemma 23. □

Virtual World Consistency

VWC is incomparable to last-use opacity.

Lemma 75. *There exists a last-use opaque history H that is not virtual world consistent.*

Proof. Since last-use opacity supports aborting early release (Lemma 27), then by Def. 4 and by Def. 6 there exists some last-use opaque history where some transaction reads from a live transaction and aborts. Since, by Lemma 16 VWC, does not support aborting releasing transactions, then, by the same definitions, such a history is not VWC. Hence a history with a transaction releasing early may be last-use opaque but not VWC. □

Lemma 76. *There exists a virtual world consistent history H that is not last-use opaque.*

Proof. Since each transaction in a VWC history can be explained by a different causal past from other transactions, it is possible that in a correct VWC history transactions do not agree on the order of operations in the sequential witness history. However, in order for H to to be last-use opaque the legality of transactions needs to be established using a single sequential history with a single order of operations. Thus, it is possible for a VWC history not to be last-use opaque. □

Transactional Memory Specification

TMS1 is incomparable to last-use opacity.

Lemma 77. *There exists a last-use opaque history H that is not TMS1.*

Proof. Since last-use opacity supports aborting early release (Lemma 27), then by Def. 6 it supports early release, so by Def. 4 there exists some last-use opaque history where some transaction reads from a live transaction and aborts. Since, by Lemma 13 TMS1, does not support early release, then, by the same definitions, histories containing early release are not TMS1. Hence a history with a transaction releasing early may be last-use opaque but not TMS1. □

Lemma 78. *There exists a TMS1 history H that is not last-use opaque.*

Proof. Let history H be the history presented in Fig. A.1. In [22] (Fig. 6 therein) the authors show that the history satisfies TMS1. The same history is not last-use opaque. Note that if $Vis(S, T_i)$ is to be legal, in any S equivalent to H, $T_i \prec_S T_j$, because T_i reads 0 from x and T_j writes 2 to x (and commits). In addition, $T_j \prec_S T_l$, because T_l reads 2 from x and $T_k \prec_S T_l$, because T_l reads z from T_k. Then, by extension $T_i \prec_S T_j \prec_S T_l$. However, note that in any S it must be that $T_l \prec_S T_i$, because T_l reads y from T_i, which is a contradiction. Thus, H is not last-use opaque. □

TMS2 is strictly stronger than last-use opacity.

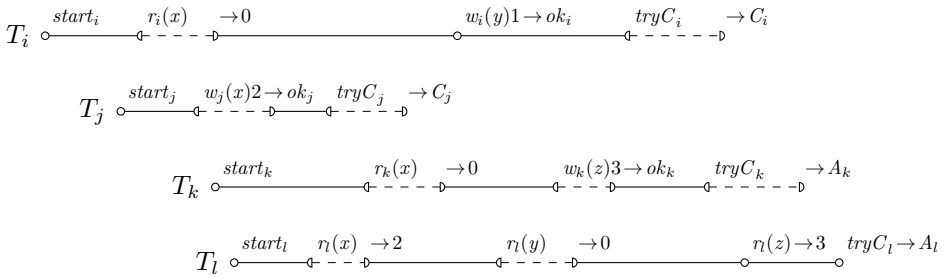


Figure A.1: TMS1 history example [22].

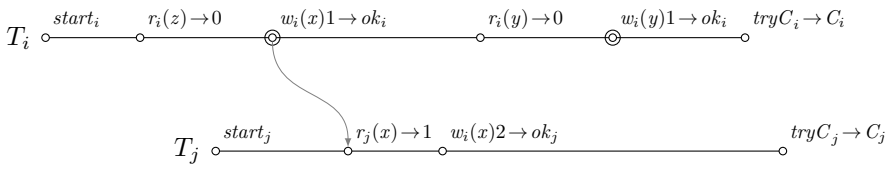


Figure A.2: Last-use opaque history that does not satisfy elastic opacity.

Proposition 1. *All TMS2 histories are last-use opaque.*

Proof. The authors of [22] believe (but do not demonstrate) that all opaque histories satisfy TMS2. If this is the case, then, since all opaque histories are last-use opaque (Lemma 88), then it is true that all last-use opaque histories satisfy TMS2. Thus, we believe the proposition is true, pending a demonstration that all opaque histories satisfy TMS2. □

Elastic Opacity

Last-use opacity and elastic opacity are incomparable.

Lemma 79. *There exists an elastic opaque history H that is not last-use opaque.*

Proof. Since elastic opaque histories may not be serializable [28], and since, as all last-use opaque histories trivially require serializability then some elastic opaque histories are not last-use opaque. □

Lemma 80. *There exists a last-use opaque history H that is not elastic opaque.*

Proof. Let history H be the history presented in Fig. A.2. It should be straightforward to see that H is last-use opaque for an equivalent sequential history $S = H|T_i \cdot H|T_j$. Operations on z are always justified in any sequential equivalent history since they are all within T_i and their effects are not visible in T_j. The read operation on y is expected to read 0 since it is not preceded in S by any write, and it does read 0. Thus operations on y and z will not break legality of either T_i or T_j. With that in mind, the history can be shown to be last-use opaque by analogy to Lemma 82.

On the other hand, let T_i be an elastic transaction. The only possible well-formed cut of $H|T_i$ is $C_i = \{[r(z)0, w(x)1, r(y)0,\ w(y)1]\}$. (In particular, the following cut is not well-formed, since $w(x)1$ and $w(y)0$ are in two different subhistories of the cut: $C_i' = \{[r(z)0, w(x)1], [r(y)0,\ w(y)1]\}$). Let $f_C(H)$ be a cutting function that applies cut C. Then, since the cut contains only one subhistory, it should be straightforward to see that $f_C(H) = H$. Then, we note that H contains an operation in $H|T_j$ that reads

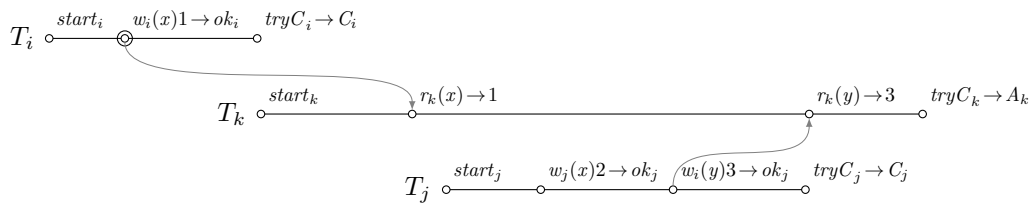


Figure A.3: Cascadeless history that does not satisfy last-use opacity.

the value of x from $H|T_i$ and T_i is live. That means that in the prefix P of H s.t. $H = P \cdot [tryC_i \to C_i, tryC_j \to C_j]$ both transactions will be aborted in any completion of P, so for any sequential equivalent history S $Vis(S, T_i)$ will not contain $S|T_j$, since either T_j is aborted in any S. Therefore $Vis(S, T_i)$ will not justify reading 1 from x and will not be legal, causing P not to be final state opaque (Def. 12), which in turn means that H is not opaque (Def. 13). □

Recoverability

Last-use opacity is strictly stronger than recoverability.

Lemma 81. *Any last-use opaque history H is recoverable.*

Proof. From Lemma 25. □

Cascadelessness

Cascadelessness is incomparable to last-use opacity.

Lemma 82. *There exists a last-use opaque history H that is not cascadeless.*

Proof. Let H be the history in Fig. 5.2. Since T_i reads from T_j in H_1 and $r_j(x) \to v \prec_{H_1} tryC_i \to C_i$ the history is not cascadeless, since it contradicts Def. 10. Let $C = Compl(H)$ s.t. $H = C$, and let $\hat{S}_H$ be a sequential history s.t. $\hat{S}_H = C|T_i \cdot C|T_j$. Then $Vis(\hat{S}_H, T_i) = \hat{S}_H|T_i = [w_i(x)1 \to ok_i]$ and $LVis(\hat{S}_H, T_j) = \hat{S}_H|T_i \cdot \hat{S}_H|T_j = [w_i(x)1 \to ok_i, r_j(x) \to 1]$. Trivially, $Vis(\hat{S}_H, T_i)$ and $LVis(\hat{S}_H, T_j)$ are both legal, so T_i is committed and legal, and T_j is last-use legal. Thus H is final-state last-use opaque. By analogy, all prefixes of H are also final-state last-use opaque, so H is last-use opaque. □

Lemma 83. *There exists a cascadeless history H that is not last-use opaque.*

Proof. The history in Fig. A.3 is shown to be cascadeless (ACA) in [7]. However, note, that $Compl(H) = H$, and given any sequential $S \equiv Compl(H)$ T_k T_k must follow both T_i and T_k in S because T_k reads form both transactions. Since $T_i \prec_H T_j$ and $T_i \prec_H T_k$, then T_i must precede both other transactions in S. Hence, $S = H|T_i \cdot H|T_j \cdot H|T_k$, so $Vis(S, T_k) = S$ and therefore $Vis(S, T_k)$ is illegal because $r_k(x) \to 1$ is preceded in $Vis(S, T_k)|x$ by $r_k(x) \to 2$. □

Strictness

Strictness and last-use opacity are also incomparable.

Lemma 84. *There exists a last-use opaque history H that is not strict.*

Proof. Since any strict history is also ACA [7], and since Lemma 82 shows that not all last-use opaque histories are ACA, then not all last-use opaque histories are strict. □

Lemma 85. *There exists a strict history H that is not last-use opaque.*

Proof. The history in Fig. A.3 is shown to be strict in [7]. However, as we show in Lemma 83, this history is not last-use opaque. □

Rigorousness

Rigorousness is strictly stronger than last-use opacity.

Lemma 86. *Any rigorous history H is last-use opaque.*

Proof. Since [7] demonstrates that rigorous histories are opaque, and since we show in Lemma 88 that opaque histories are also last-use opaque, then all rigorous histories are last-use opaque. □

Live Opacity

Live opacity is stronger than last-use opacity.

Lemma 87. *Any live opaque history H is last-use opaque.*

Proof. Since H is live opaque there exists a sequential history S that justifies serializability of H and an extension S' of S where if transaction T_i is not in S then it is replaced in S' by T_i^{gr} containing only non-local reads. S' is legal and preserves the real-time order of H (accounting for replaced transactions). In addition, from Lemma 19, no transaction in H reads from a live transaction (in any prefix of H). Therefore, since S' is legal, any read operation $op_i = r_i(x) \to v$ in H that is preceded $w_j(x)v \to u$ in H, T_j is committed in S and is included in S' in full.

Let S'' be a sequential history constructed by replacing the operations removed to create S' where if $T_i \in H$ and $T_i \notin S$ then T_i is aborted in S''. S'' preserves the real time order of H and $S'' \equiv H$. Note that, since S' is legal, if some write op^w is in S'' and not in S', then there is no non-local read operation op^r reading the value written by op^w. Hence any operation reading the value written by op^w is local, and since all local reads in transactions that are replaced in S' read legal values (by Def. 18), then all reads reading from any op_w read legal values in S''. Since S' is legal, then all reads reading from transactions that are in S read legal values in S'. Since $S'' \equiv H$, then these read and write operations also read legal values in S''. Because of this, and since no transaction reads from another live transaction, $Vis(S'', T_i)$ will be legal for any transaction in S''. In addition, $LVis(S'', T_i)$ will be legal for any aborted transaction in S''. Therefore any live opaque H will be final state last-use opaque. Since any prefix of H is also live opaque, then any prefix will also be final-state last-use opaque, hence H is last-use opaque. □

Markability

Lemma 88. *Any markable history H is last-use opaque.*

Proof. Trivially from Lemma 88. □

Commitment Order Preservation

CO and last-use opacity are incomparable.

Lemma 89. *There exists a last-use opaque history H that is not CO.*

Proof. Let H be the history in Fig. 5.9. Since both $H|T_i$ and $H|T_j$ each single write operation on x, then for any equivalent sequential history $\hat{S}_H$, all of $Vis(\hat{S}_H, T_i)$, $Vis(\hat{S}_H, T_j)$, $LVis(\hat{S}_H, T_i)$, and $LVis(\hat{S}_H, T_j)$ are always legal. Hence, H is final-state last-use opaque. Since the conclusion follows for any prefix of H, then any prefix is final-state last-use opaque, and H is last-use opaque. However, since T_i and T_j conflict, and since T_i executes its write on x before T_j, but T_j commits before T_i, then H is not CO. □

Lemma 90. *There exists a CO history H that is not last-use opaque.*

Proof. We show in Lemma 5 that CO supports overwriting, so by Def. 5, there exists some H that contains overwriting and satisfies CO. We show in Lemma 26 that last-use opacity precludes overwriting, so such H is not last-use opaque. □

SVA+R Last-use Opacity

Since the values used within writes are under the control of the program (rather than SVA+R) we simply assume that they are within the domain of the variables (Assumption 5).

Definition 54 (Operation Execution Conditional)**.** *Given predicate P and operation op $P \rightharpoonup op$ denotes that P is true only if op executes.*

Definition 55 (Operation Execution Converse)**.** *Given predicate P and operation op $P \leftharpoonup op$ denotes that op executes only if P is true.*

Let there be any $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$, $op_i \in H|T_i$.

Definition 56. *op_i is closing access on x in T_i, denoted $op_i = \ddot{o}p_i^x$ if both:*

a) op_i is closing read on x in T_i or op_i is closing write on x in T_i, and

b) $\nexists op_i' \in H$ s.t. $op_i \prec_H op_i'$ and op_i' is closing read on x in T_i or op_i' is closing write on x in T_i.

Let there be any $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$, $op_i \in H|T_i$, $op_i = r_i(x) \rightarrow v$ or $w_i(x)v \rightarrow ok_i$.

Lemma 91 (Access Condition)**.** $\texttt{lv}(x) = \texttt{pv}_i(x) - 1 \leftharpoondown op_i$.

Proof. Condition at line 11 dominates access at line 16. □

Lemma 92 (Abort Condition)**.** $\texttt{ltv}(x) = \texttt{pv}_i(x) - 1 \leftharpoondown res_i\big[A_i\big]$.

Proof. Access condition at line 35 dominates :`dismiss` at line 36 in procedure abort for each variable. Hence, all variables must pass line 35 before abort concludes. □

Lemma 93 (Commit Condition)**.** $\texttt{ltv}(x) = \texttt{pv}_i(x) - 1 \leftharpoondown res_i\big[C_i\big]$.

Proof. By analogy to Lemma 92. □

Lemma 94 (Early Release)**.** *If $op_i = \ddot{o}p_i^x$ then $\texttt{lv}(x) = \texttt{pv}_i(x) \rightharpoonup op_i$.*

Proof. $\texttt{lv}(x)$ can be set by T_i at line 59 and at line 47. The former is set during the last access on some x in T_j (line 18 dominates line 59). The latter is set during commit, which means that if any closing access was present, it was executed prior to commit. □

Let $r_i = res_i\big[A_i\big]$ or $res_i\big[C_i\big]$.

Lemma 95 (Release). *If* $\nexists op'_i \in H|T_i$ *s.t.* $op'_i = \ddot{o}p^x_i$ *and* $x \in \mathtt{ASet}_i$ *then* $\mathtt{lv}(x) = \mathtt{pv}_i(x) \rightharpoonup r_i$.

Proof. If op'_i is not closing access then line 18 will not be passed, so only assignment of $\mathtt{lv}(x)$ is in line 47 which execute only during commit or abort. □

Lemma 96 (Terminal Release). *If* $x \in \mathtt{ASet}_i$ *then* $\mathtt{lv}(x) = \mathtt{pv}_i(x) \rightharpoonup r_i$.

Proof. $\mathtt{ltv}(x)$ can be set only in line 39 or line 30, which are part of abort and commit, respectively. □

Let there be any H, $T_i \in H$, $T_j \in H$, $op_i \in H|T_i$, $op_i = r_i(x) \to u$, $op_j \in H|T_j$, $op_j = w_j(x)v \to ok_j$.

Lemma 97 (No Buffering). *If* $op_j \lessdot_{H|x} op_i$ *and not* $op_j \prec_H res_j\big[A_j\big] \prec_H op_i$ *then* $u = v$.

Lemma 98 (Revert On Abort). *If* $op_j \lessdot_{H|x} op_i$ *and* $op_j \prec_H res_j\big[A_j\big] \prec_H op_i$ *then* $u \neq v$.

Proof. If abort is executed then the :recover procedure is executed for all $x \in \mathtt{ASet}_i$. Thus, line 54 restores x to value v' which is acquired before the any operation on x is executed by T_i, hence $v' \neq v$, so $u \neq v$. □

Let $H|start$ be a subhistory of H that for each $T_j \in H$ contains only the operation $start_j$.

Lemma 99 (Consecutive Versions). *If* $x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$ *and* $inv_i\big[start_i\big] \lessdot_{H|start} inv_i\big[start_j\big]$ *then* $\mathtt{pv}_i(x) - 1 = \mathtt{pv}_j(x)$.

Proof. If T_i returns at line 3 for x then no T_j s.t. $x \in \mathtt{ASet}_j$ returns at line 3 until T_i executes line 7 for x. Hence, T_i alone increments $\mathtt{gv}(x)$ at line 5 and sets $\mathtt{pv}_i(x)$ to the new value of $\mathtt{gv}(x)$. If $start_i \lessdot_{H|start} start_j$ then T_i will return at line 3 and T_j will return next. No other transaction will return at line 3 between T_i and T_j. □

Lemma 100 (Unique Versions). *If* $x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$ *then* $\mathtt{pv}_i(x) \neq \mathtt{pv}_j(x)$.

Proof. From Lemma 99. □

Lemma 101 (Monotonic Versions). *If* $x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$ *and* $inv_i\big[start_i\big] \prec_{H|start} inv_i\big[start_j\big]$ *then* $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$.

Proof. From Lemma 99, Lemma 100. □

Definition 57 (Version Order). *Let* $\prec_x$ *be an order s.t.* $T_i \prec_x T_j$ *iff* $\mathtt{pv}_i(x) < \mathtt{pv}_j(x)$.

Lemma 102 (Forced Abort Condition). $\mathtt{rv}_i(x) < \mathtt{cv}(x) \rightharpoonup res_i\big[A_i\big]$.

Proof. Condition at line 14 dominates abort at line 15. Condition at line 27 dominates abort at line 28. □

Let there be any $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$, $op_i \in H|T_i$, $op_i = r_i(x) \to v$ or $w_i(x)v \to ok_i$.

Lemma 103. $\mathtt{cv}(x) < \mathtt{rv}_i(x) \leftharpoondown op_i$.

Proof. Condition at line 14 dominates abort at line 15. □

Lemma 104 (Current Version Early Release). *If* $op_i = \ddot{o}p^x_i$ *then* $\mathtt{cv}(x) = \mathtt{rv}_i(x) \rightharpoonup op_i$.

Proof. By analogy to Lemma 94. □

Lemma 105 (Current Version Release)**.** *If* $\nexists op_i \in H|T_i$ *s.t.* $op_i = \ddot{o}p_i^x$ *and* $x \in \mathtt{ASet}_i$ *then* $\mathtt{cv}(x) = \mathtt{rv}_i(x) \rightharpoonup r_i$.

Proof. By analogy to Lemma 95. □

Lemma 106. $\mathtt{cv}(x) = \mathtt{rv}_i(x) \leftharpoondown res_i\big[A_i\big]$.

Proof. From Lemma 95 and Lemma 105. □

Let there be any $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$, $T_i \in H$, $T_j \in H$, $op_i \in H|T_i$, $op_j \in H|T_j$, $op_i = r_i(x) \rightarrow v$ or $w_i(x)v \rightarrow ok_i$, $op_j = r_j(x) \rightarrow v$ or $w_j(x)v \rightarrow ok_j$.

Lemma 107 (Access Order)**.** $\mathtt{pv}_i(x) < \mathtt{pv}_j(x) \Leftrightarrow op_i \prec_H op_j$.

Proof. From Lemma 91 and Lemma 101. □

Let there be any H, $T_i \in H$, $T_j \in H$, $op_i \in H|T_i$, $op_i = r_i(x) \rightarrow u$, $op_j \in H|T_j$, $op_j = w_j(x)v \rightarrow ok_j$. Let there be any $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$.

Lemma 108 (Access Prefix)**.** *If* $\mathtt{lv}(x) = \mathtt{pv}_j(x)$ *then* $\forall T_k \in H$ *s.t.* $\mathtt{pv}_x(k) < \mathtt{pv}_i(x)$ *either* $res_k\big[C_k\big] \in H|T_k$, $res_k\big[A_k\big] \in H|T_k$, *or* $\ddot{o}p_k^x \in H|T_k$.

Proof.

$$\forall T_l, T_k \in H \text{ s.t. } \mathtt{pv}_x(l) = \mathtt{pv}_x(k) - 1 : \tag{A.1}$$
$$\text{Lemma 91} \Longrightarrow \mathtt{lv}(x) = \mathtt{pv}_x(k) - 1 \leftharpoondown op_k \tag{A.2}$$
$$(A.1) \wedge (A.2) \Longrightarrow \mathtt{lv}(x) = \mathtt{pv}_x(l) \leftharpoondown op_k \tag{A.3}$$
$$\text{Lemma 94} \Longrightarrow \mathtt{lv}(x) = \mathtt{pv}_x(l) \rightharpoonup \ddot{o}p_l^x \tag{A.4}$$
$$\text{Lemma 95} \Longrightarrow \mathtt{lv}(x) = \mathtt{pv}_x(l) \rightharpoonup r \text{ where } r = res_i\big[A_i\big] \text{ or } r = res_i\big[C_i\big] \tag{A.5}$$
$$(A.4) \wedge (A.5) \Longrightarrow T_l \text{ is committed, aborted or decided on } x \tag{A.6}$$

Trivially extends for any T_l, T_k s.t. $\mathtt{pv}_x(l) < \mathtt{pv}_x(k)$. □

Lemma 109. *If* $\mathtt{ltv}(x) = \mathtt{pv}_j(x)$ *then* $\forall T_k \in H$ *s.t.* $\mathtt{pv}_x(k) < \mathtt{pv}_i(x)$ *either* $res_k\big[C_k\big] \in H|T_k$, *or* $res_k\big[A_k\big] \in H|T_k$.

Proof.

$$\forall T_l, T_k \in H \text{ s.t. } \mathtt{pv}_x(l) = \mathtt{pv}_x(k) - 1 : \tag{A.7}$$
$$\text{Lemma 93} \Longrightarrow \mathtt{ltv}(x) = \mathtt{pv}_x(k) - 1 \leftharpoondown res_k\big[C_i\big] \tag{A.8}$$
$$(A.7) \wedge (A.8) \Longrightarrow \mathtt{ltv}(x) = \mathtt{pv}_x(l) \leftharpoondown op_k \tag{A.9}$$
$$\text{Lemma 96} \Longrightarrow \mathtt{ltv}(x) = \mathtt{pv}_x(l) \rightharpoonup r \text{ where } r = res_i\big[A_i\big] \text{ or } r = res_i\big[C_i\big] \tag{A.10}$$
$$(A.10) \Longrightarrow T_l \text{ is committed or aborted} \tag{A.11}$$

Trivially extends for any T_l, T_k s.t. $\mathtt{pv}_x(l) < \mathtt{pv}_x(k)$. □

Let there be any H, $T_i \in H$, $T_j \in H$, $op_i \in H|T_i$, $op_i = r_i(x) \rightarrow u$, $op_j \in H|T_j$, $op_j = w_j(x)v \rightarrow ok_j$.

Lemma 110 (Forced Abort)**.** *If* $x \in \mathtt{ASet}_i \cap \mathtt{ASet}_j$ *and* $res_j\big[A_j\big] \in H|T_j$ *and* $op_i \prec_H res_j\big[A_j\big]$ *then* $res_i\big[A_i\big] \in H|T_i$.

Proof.

$$res_j\big[A_j\big] \in H|T_j \wedge \textit{Lemma } 106 \Longrightarrow \mathtt{cv}(x) = pvjx \leftharpoondown res_j\big[A_j\big] \tag{A.12}$$
$$\textit{Lemma } 101 \Longrightarrow \mathtt{pv}_j(x) < \mathtt{pv}_i(x) \tag{A.13}$$
$$(A.12) \wedge (A.13) \Longrightarrow \mathtt{cv}(x) < pvix \leftharpoondown res_j\big[A_j\big] \tag{A.14}$$
$$\textit{Lemma } 103 \Longrightarrow \mathtt{cv}(x) = \mathtt{rv}_i(x) \leftharpoondown op_i \Longrightarrow \mathtt{rv}_i(x) = \mathtt{pv}_j(x) \tag{A.15}$$
$$(A.15) \wedge (A.13) \Longrightarrow \mathtt{rv}_i(x) > \mathtt{pv}_x(i) \tag{A.16}$$
$$(A.16) \wedge (A.14) \Longrightarrow \mathtt{rv}_i(x) < \mathtt{cv}(x) \tag{A.17}$$
$$(A.17) \Longrightarrow \mathtt{rv}_i(x) < \mathtt{cv}(x) \rightharpoonup res_i\big[A_i\big] \Longrightarrow res_i\big[A_i\big] \in H|T_i \tag{A.18}$$

□

Let there be any $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$, $T_i \in H$, $T_j \in H$, $op_i \in H|T_i$, $op_j \in H|T_j$, $op_i = r_i(x) \rightarrow v$ or $w_i(x)v \rightarrow ok_i$, $op_j = r_j(x) \rightarrow v$ or $w_j(x)v \rightarrow ok_j$.

Definition 58 (Completion Construction)**.** *$H_C = Compl(H)$ s.t. $\forall T_k \in H$, $res_k\big[C_k\big] \notin H|T_k \Leftrightarrow res_k\big[A_k\big] \in H_C|T_k$*

Definition 59 (Sequential History Construction)**.** *$\hat{S}_H$ is a sequential history s.t. $\hat{S}_H \equiv H_C$ and $T_i \prec_{H_C} T_j \Longrightarrow T_i \prec_{\hat{S}_H} T_j$ and $T_i \prec_x T_j \Longrightarrow T_i \prec_{\hat{S}_H} T_j$.*

Let there be any H, $T_i \in H$, $T_j \in H$, $op_i \in H|T_i$, $op_i = r_i(x) \rightarrow u$, $op_j \in H|T_j$, $op_j = w_j(x)v \rightarrow ok_j$.

Lemma 111. *If T_i reads x from T_j then T_j is committed in H or T_j is decided on x in H.*

Proof.

$$T_i \text{ reads } x \text{ from } T_j \Longrightarrow op_i = r_i(x) \rightarrow v \wedge op_j = w_j(x)v \rightarrow ok_i \wedge op_j \prec_H op_i \tag{A.19}$$
$$\textit{Lemma } 91 \Longrightarrow \mathtt{lv}(x) = \mathtt{pv\_}(x)1 \leftharpoondown op_i \tag{A.20}$$
$$\textit{Lemma } 107 \wedge op_j \prec_H op_i \Longrightarrow \mathtt{pv}_j(x) < \mathtt{pv}_i(x) \tag{A.21}$$
$$(A.21) \wedge \textit{Lemma } 108 \Longrightarrow T_j \text{ is committed, aborted, or decided on } x \tag{A.22}$$

Let us assume that T_j is aborted:

$$op_i \prec res_j\big[A_j\big] : \textit{Lemma } 99 \Longrightarrow v \neq v \Longrightarrow \text{ contradiction} \tag{A.23}$$
$$res_j\big[A_j\big] \prec_H op_i : \textit{Lemma } 94 \Longrightarrow \mathtt{lv}(x) = \mathtt{pv}_{\_}(x)op_i \wedge op_i = \ddot{o}p_i^x \tag{A.24}$$

Thus, T_i is committed or decided on x. □

Corollary 23. *If P is any prefix of H, then if T_i reads x from T_j in P then T_j is committed in P or T_j is decided on x in P.*

Lemma 112. *If T_i reads x from T_j and T_j is committed in H then T_j is committed in H.*

Proof.

$$T_i \text{ reads } x \text{ from } T_j \implies op_i = r_i(x) \to v \land op_j = w_j(x)v \to ok_i \land op_j \prec_H op_i \tag{A.25}$$

$$\text{Lemma 93} \implies \texttt{ltv}(x) = \texttt{pv}_x(k) - 1 \leftharpoonup res_i\big[ok_i\big] \tag{A.26}$$

$$\text{Lemma 107} \land op_j \prec_H op_i \implies \texttt{pv}_j(x) < \texttt{pv}_i(x) \tag{A.27}$$

$$\text{Lemma 109} \land (A.26) \land (A.27) \implies r \in H|T_j \text{ where } r = res_j\big[A_j\big] \text{ or } r = res_j\big[C_j\big] \tag{A.28}$$

$$\text{Lemma 110} \implies \text{if } res_j\big[A_j\big] \in H|T_j \text{ then } res_j\big[A_j\big] \in H|T_j \implies \text{contradiction} \tag{A.29}$$

$$(A.29) \implies res_i\big[A_i\big] \in H|T_i \tag{A.30}$$

□

Let there be any $\mathbb{P}, \Pi, H \models \mathcal{E}(\mathbb{P}, \Pi)$, $T_i \in H$, $op_i = r_i(x) \to v$, $op_i \in H|T_i$.

Lemma 113. *If* $res_i\big[C_i\big] \in \hat{S}_H|T_i$ *then* $\exists op_j = w_j(x)v \to ok_j \in Vis(\hat{S}_H, T_i)$.

Proof. If $i = j$ then trivially $op_j \in Vis(\hat{S}_H, T_i)$. Otherwise:

$$i \neq j \land \text{Lemma 97} \implies \exists T_j \land op_j \in H_C|T_j \tag{A.31}$$

$$(A.31) \land \text{Lemma 112} \land res_i\big[C_i\big] \in H_C|T_i \implies \exists res_j\big[C_j\big] \in H_C|T_j \tag{A.32}$$

$$\text{Def. 59} \land (A.32) \implies res_j\big[C_j\big] \in \hat{S}_H|T_j \land T_j \prec_{\hat{S}_H} T_i \tag{A.33}$$

$$(A.33) \implies \hat{S}_H|T_j \subseteq Vis(\hat{S}_H, T_i) \implies op_j \in Vis(\hat{S}_H, T_i) \tag{A.34}$$

□

Lemma 114. $\exists op_j = w_j(x)v \to ok_j \in LVis(\hat{S}_H, T_i)$.

Proof. If $i = j$ then trivially $op_j \in LVis(\hat{S}_H, T_i)$. Otherwise:

$$i \neq j \land \text{Lemma 97} \implies \exists T_j \land op_j \in H_C|T_j \tag{A.35}$$

$$(A.35) \land \text{Lemma 111} \land res_i\big[C_i\big] \in H_C|T_i \implies \text{either } \exists res_j\big[C_j\big] \in H_C|T_j \text{ or } \exists \ddot{o}p_j^x \in H_C|T_j \tag{A.36}$$

$$\text{Def. 59} \land (A.36) \implies res_j\big[C_j\big] \in \hat{S}_H|T_j \land T_j \prec_{\hat{S}_H} T_i \tag{A.37}$$

$$(A.37) \implies \hat{S}_H|T_j \subseteq LVis(\hat{S}_H, T_i) \implies op_j \in LVis(\hat{S}_H, T_i) \tag{A.38}$$

$$(A.36) \implies \ddot{o}p_j^x \in \hat{S}_H|T_j \land T_j \prec_{\hat{S}_H} T_i \tag{A.39}$$

$$(A.39) \land \hat{S}_H\hat{|}T_j \subseteq LVis(\hat{S}_H, T_i) \implies op_j \in LVis(\hat{S}_H, T_i) \tag{A.40}$$

□

Lemma 115. *Given* $\hat{S}_H$ *and any two transactions* $T_i, T_j \in \hat{S}_H$ *s.t. there is an operation execution* $w_j(x)v \to ok_j \in \hat{S}_H|T_j$ *and* $r_i(x) \to v \in \hat{S}_H|T_i$ *then there is no operation* $w_k(x)u \to ok_k$ *(executed by some* $T_k \in \hat{S}_H$*) in* $Vis(\hat{S}_H, T_i)$ *s.t.* $w_k(x)u \to ok_k$ *precedes* $r_i(x) \to v$ *in* $Vis(\hat{S}_H, T_i)$ *and follows* $w_j(x)v \to ok_j$ *in* $Vis(\hat{S}_H, T_i)$.

Proof. For the sake of contradiction, assume that op_k exists as specified.

If $k = i$, then $op_k \prec_{H|T_i} op_i$, which contradicts Lemma 97 (assuming unique writes).

If $k = j$, then from Lemma 111 T_j is either committed or decided on x in $\hat{S}_H$. If T_i commits, then op_i reading v contradicts Lemma 97. If T_i does not commit in P, then this contradicts Lemma 112.

Otherwise, $\exists T_k \in H$ s.t. $op_k \in H|T_k$ from Lemma 111 T_j is either committed or decided on x in $\hat{S}_H$ and from Lemma 112 T_k is committed in H. Since T_k commits, this contradicts Lemma 97. □

Lemma 116. *Given $\hat{S}_H$ and any two transaction $T_i, T_j \in \hat{S}_H$ s.t. there is an operation execution $w_j(x)v \to ok_j \in \hat{S}_H|T_j$ and $r_i(x) \to v \in \hat{S}_H|T_i$ then there is no operation $w_k(x)u \to ok_k$ (executed by some $T_k \in \hat{S}_H$) in $LVis(\hat{S}_H, T_i)$ s.t. $w_k(x)u \to ok_k$ precedes $r_i(x) \to v$ in $Vis(\hat{S}_H, T_i)$ and follows $w_j(x)v \to ok_j$ in $Vis(\hat{S}_H, T_i)$.*

Proof. By analogy to Lemma 115. □

Proof for Lemma 38. Given $\hat{S}_H$, let $T_i \in \hat{S}_H$ be any transaction that is committed in $\hat{S}_H$. In that case, from Lemma 113 and Lemma 115, every read operation execution $r_i(x) \to v$ in $Vis(\hat{S}_H, T_i)$ is preceded in $Vis(\hat{S}_H, T_i)$ by a write operation execution $w_j(x)v \to ok_j$ (for some T_j). In addition, from Assumption 5, every write operation execution $w_i(x)v \to ok_i$ in $Vis(\hat{S}_H, T_i)$ trivially writes $v \in D$. Therefore, for every variable x, $Vis(\hat{S}_H, T_i)|x \in Seq(x)$, so $Vis(\hat{S}_H, T_i)$ is legal. Consequently T_i in $\hat{S}_H$ is legal in $\hat{S}_H$.

Given the same $\hat{S}_H$, let $T_i \in \hat{S}_H$ be any transaction that is not committed in $\hat{S}_H$ (so it is aborted in $\hat{S}_H$). From Lemma 114 and Lemma 116, every read operation execution $r_i(x) \to v$ in $LVis(\hat{S}_H, T_i)$ is preceded in $LVis(\hat{S}_H, T_i)$ by a write operation execution $w_j(x)v \to ok_j$ (for some T_j). In addition, from Assumption 5, every write operation execution $w_i(x)v \to ok_i$ in $LVis(\hat{S}_H, T_i)$ trivially writes $v \in D$. Therefore, for every variable x, $LVis(\hat{S}_H, T_i)|x \in Seq(x)$, so $LVis(\hat{S}_H, T_i)$ is legal. Thus, T_i in $\hat{S}_H$ is last-use legal in $\hat{S}_H$.

Since all committed transactions in $\hat{S}_H$ are legal in $\hat{S}_H$ and since all aborted transactions in $\hat{S}_H$ are last-use legal in $\hat{S}_H$, then, by Def. 23 H is final-state last use opaque. □

Last-use Opacity from Trace Harmony

Composition Rules

Given trace $\mathscr{T}$ and a history $H = Hist(\mathscr{T})$, let $\hat{C} = Compl(H)$ be a completion of H s.t. for every $T_i \in H$, if T_i is live or commit-pending in H, then T_i is aborted in H_C. Let $\hat{T}_i$ such a transaction in $\hat{C}$ that corresponds to a completion of T_i in $\hat{C}$.

Definition 60 (Equivalent Sequential History Construction)**.** *Let $\hat{S}_H$ be a sequential history s.t. $\hat{S}_H \equiv H_C$ and, given two transactions $T_i, T_j \in \hat{C}$:*

1. *if $T_i \prec_{\mathscr{T}} T_j$, then $T_i \prec_{\hat{S}_H} T_j$,*
2. *otherwise, if $T_i \dot{\prec}_{\mathscr{T}} T_j$ for any variable x, then $T_i \prec_{\hat{S}_H} T_j$,*
3. *otherwise, if $\exists op_j = w_j(x)\square \to ok_j \in \mathscr{T}|T_j$ and $\exists e^i = g_i(x)\square \in \mathscr{T}|T_i$ or $e^i = \circ s_i(x)\square \in \mathscr{T}|T_i$, then $T_i \prec_{\hat{S}_H} T_j$.*

Definition 61 (Last-use Visible History Construction)**.** *Given transactions T_i and T_j in $\mathscr{T}$:*

1. *if T_j is committed in $\mathscr{T}$, then $\hat{T}_j$ is included in $LVis(\hat{S}_H, T_i)$ as a whole, otherwise*
2. *if T_j is aborted in $\mathscr{T}$ and $T_j \prec_{\mathscr{T}} T_i$, $\hat{T}_j$ is not included in $LVis(\hat{S}_H, T_i)$ at all, otherwise*
3. *if there exists $\xi(\mathscr{T}, T_j, T_i)$, then $\hat{S}_H \overset{\circ}{|} \hat{T}_j$ is included in $LVis(\hat{S}_H, T_i)$, otherwise*
4. *T_j is not included in $LVis(\hat{S}_H, T_i)$ at all.*

Auxiliary Lemmas

Lemma 117. *Let there be a consonant, isolation-ordered, trace $\mathscr{T}$ in obbligato and $H = Hist(\mathscr{T})$ from which $\hat{S}_H$ is generated, and $T_i, T_j \in \mathscr{T}$. Given any non-local $op_i = r_i(x) \to v \in \mathscr{T}|T_i$ s.t. $\exists e_i = g_i(x)v \in \mathscr{T}|T_i$ and $op_i \leftharpoonup e_i$ and given any non-local $op_j = w_j(x)v \to ok_j \in \mathscr{T}|T_j$ s.t. $\exists e_j = \in \circ s_i(x)v \in \mathscr{T}|T_j$ and $op_j \rightharpoonup e_j$, and $e_j \lessdot_{\mathscr{T}} e_i$, then $\nexists T_k \in \mathscr{T}$ s.t. T_k $op_k = w_j(x)v' \to ok_j$ s.t. $op_i \prec_{\hat{S}_H} op_k \prec_{\hat{S}_H} op_j$ and T_k is either committed or decided on x in trace $\mathscr{T}$.*

Proof. Assume for the sake of contradiction that such T_k exists in $\mathscr{T}$. Since both op_i and op_j are non-local, then $i \neq j \neq k$.

If T_k is committed, then, from the definition of commit write obbligato, $\exists e_k = \circ s_k(x)v' \in \mathscr{T}|T_k$ if $inv_k\big[w_k(x)v'\big]$ is the invocation event of op_k then $inv_k\big[w_k(x)v'\big] \prec_{\mathscr{T}} e_k \prec_{\mathscr{T}} res_k\big[C_k\big]$.

If T_k is decided on x in $\mathscr{T}$, then, from the definition of closing write obbligato, $\exists e_k = \circ s_k(x)v' \in \mathscr{T}|T_k$ s.t. if $inv_k\big[w_k(x)v'\big]$ is the invocation event of op_k then $inv_k\big[w_k(x)v'\big] \prec_{\mathscr{T}} e_k \prec_{\mathscr{T}} e_i$.

Thus, in either of the above cases, $e_j \prec_{\mathscr{T}} e_k$ and either $e_k \prec e_i$ or $e_i \prec e_k$. If then $e_k \prec e_i$, it is not true that $e_j \lessdot_{\mathscr{T}} e_i$, which is a contradiction. Alternatively, if $e_i \prec e_k$, then, since $\mathscr{T}$ is isolation-ordered, $T_i \dot{\prec}^x_{\mathscr{T}} T_k$, which implies that $T_i \prec_{\hat{S}_H} T_k$. In this case, $op_i \prec_{\hat{S}_H} op_k$, which is a contradiction.

Therefore, there can be no such T_k, which satisfies the lemma. □

Lemma 118. *Given a consonant trace $\mathscr{T}$, and $T_i \in \mathscr{T}$, if T_j is the first element of $\psi_{\mathscr{T}}(T_j, x)$, then $\exists e_v = g_j(x)v \in \mathscr{T}|T_j$ that is initial and non-local, and either*

a) $v = 0$ and $\nexists T_k \in \mathscr{T}$ s.t. $e_u = s_k(x)v \in \mathscr{T}|T_k$ and $e_u \prec_{\mathscr{T}} e_v$,

b) $v \neq 0$ and $\exists T_k \in \mathscr{T}$ s.t. $e_u = s_k(x)v \in \mathscr{T}|T_k$ and $e_u \lessdot_{\mathscr{T}} e_v$.

Proof. Since T_j is in $\psi_{\mathscr{T}}(T_i, x)$ then by definition, either $k = i$ or $e_a = ¤s_j(x)v \in \mathscr{T}|T_j$. In either case $e_v = g_j(x)v \in \mathscr{T}|T_j$ s.t. e_v is initial and non-local (in the former case by definition of $\psi_{\mathscr{T}}(T_i, x)$ and in the latter by definition of recovery update consonance).

Since e_v is consonant and non-local, then either:

a) $v = 0$ and $\nexists T_k \in \mathscr{T}$ s.t. $e_u s_k(x)v' \in \mathscr{T}|T_k$ $e_u \prec_{\mathscr{T}} e_r$,

b) $v \neq 0$ and $\exists T_k \in \mathscr{T}$ s.t. $e_u = \circ s_j(x)v \in \mathscr{T}|T_k$, $i \neq k$, $e_u \lessdot_{\mathscr{T}} e_r$, e_u is consonant, and e_u is the ultimate routine update on x in $\mathscr{T}|T_k$, or

c) $\exists e_u \in \mathscr{T}$ s.t. $e_u = ¤s_j(x)v$ for some tr_k, $j \neq k$, $e_u \lessdot_{\mathscr{T}} e_r$, e_u is a consonant recovery event, and is the ultimate update on x in $\mathscr{T}|T_k$.

In the latter-most case, if such e_u exists in T_k then, $T_k \in \psi_{\mathscr{T}}(T_j, x)$ so that T_k preceded T_j in $\psi_{\mathscr{T}}(T_j, x)$. Thus, T_k would precede T_j in $\psi_{\mathscr{T}}(T_i, x)$, and therefore T_j is not the first element of $\psi_{\mathscr{T}}(T_i, x)$. Thus, the latter-most case is impossible. □

Lemma 119. *Given a consonant trace $\mathscr{T}$, and $T_i \in \mathscr{T}$, $\forall T_j \in \psi_{\mathscr{T}}(T_i, x)$ $(i \neq j)$, T_j is aborted or live in $\mathscr{T}$.*

Proof. Since $i \neq j$ then $\forall T_j \in \mathscr{T}$, $\exists e_a = ¤s_j(x)v\mathscr{T}|T_j$. Since $\mathscr{T}$ is consonant, then e_a is consonant, so e_a is dooming. Thus T_j is aborted or live in $\mathscr{T}$. □

Lemma 120. *Given a consonant, abort abiding trace $\mathscr{T}$ in obbligato, and a pair of transaction $T_i, T_j, \in \mathscr{T}$, and T_j is the first element in $\psi_{\mathscr{T}}(T_i, x)$, $\forall T_k \in \mathscr{T}$ if $T_j \dot{\prec}^x_{\mathscr{T}} T_k \dot{\prec}^x_{\mathscr{T}} T_i$ and $op_k w_k(x)v \to ok_k \in \mathscr{T}|T_k$ then T_k is aborted or live in $\mathscr{T}$.*

Proof. If $i = j$, then the lemma is vacuously true.

Since $T_j \dot{\prec}^x_{\mathscr{T}} T_k \dot{\prec}^x_{\mathscr{T}} T_i$, then $\exists e_u = \circ s_k(x)v' \in \mathscr{T}|T_k$ or $e_v = g_k(x)v' \in \mathscr{T}|T_k$. Hence, either e_u exists in $\mathscr{T}|T_k$ or it does not.

If e_u does not exist, then, from commit write obbligato, T_k cannot commit in *trace*, so T_k is either live or aborted in $\mathscr{T}$.

If e_u exists, then, since $T_j \dot{\prec}^x_{\mathscr{T}} T_k \dot{\prec}^x_{\mathscr{T}} T_i$ and from the definition of $\psi_{\mathscr{T}}(T_i, x)$, there is some pair of transactions T_α and $T_\beta \in \mathscr{T}$ s.t. $T_\alpha, T_\beta \in \psi_{\mathscr{T}}(T_i, x)$ and T_α immediately precedes T_β in $\psi_{\mathscr{T}}(T_i, x)$ and $T_\alpha \dot{\prec}^x_{\mathscr{T}} T_k \dot{\prec}^x_{\mathscr{T}} T_\beta$. Therefore $\exists e_\alpha = \sqcap s_\alpha(x)v_\alpha \in \mathscr{T}|T_\alpha$ and $e_\beta = g_\beta(x)v_\beta \in \mathscr{T}|T_\beta$ s.t. $e_\alpha \lessdot_{\mathscr{T}} e_\beta$. In addition, since e_α is consonant, then it is needed, so $\exists e'_\alpha = \circ s_\alpha(x)v'_\alpha \in \mathscr{T}|T_\alpha$ s.t. $e'_\alpha \lessdot_{\mathscr{T}} e_\alpha$. Also, from definition of isolation order, $e'_\alpha \prec_{\mathscr{T}} e_u \prec_{\mathscr{T}} e_\beta$. Then, $e'_\alpha \prec_{\mathscr{T}} e_u \prec_\alpha \lessdot_{\mathscr{T}} e_\beta$. Therefore, from the definition of abort accord, T_k is either live or aborted in $\mathscr{T}$. □

Lemma 121. *Given a consonant trace $\mathscr{T}$, and $T_i \in \mathscr{T}$, $\forall T_j, T_k \in \psi_{\mathscr{T}}(T_i, x)$ $(i \neq j)$, if T_j precedes T_k in $\psi_{\mathscr{T}}(T_i, x)$ then $T_j \dot{\prec}^x_{\mathscr{T}} T_k$.*

Proof. Given $\psi_{\mathscr{T}}(T_i, x)$, from Lemma 119, $\forall T_k \in \psi_{\mathscr{T}}(T_i, x)$, T_k is aborted or live in $\mathscr{T}$. In addition, since for all $T_m \in \psi_{\mathscr{T}}(T_i, x)$ except the first, where $e^m_v = g_m(x)v \in \mathscr{T}|T_m$ there is some T_n that directly precedes T_m in $\psi_{\mathscr{T}}(T_i, x)$ and contains $e^n_a = \sqcap s_n(x)v$ s.t. $e^n_a \lessdot_{\mathscr{T}} e^m_v$. Since e^n_a is conservative, there is a preceding view $e^n_v = g_n(x)v$ s.t. $e^n_v \lessdot_{\mathscr{T}} e^n_a$. Thus $e^n_v \lessdot_{\mathscr{T}} e^m_v$, so $T_n \dot{\prec}^x_{\mathscr{T}} T_m$. □

Corollary 24. *Given a consonant trace $\mathscr{T}$, and $T_i \in \mathscr{T}$, $\forall T_j \in \psi_{\mathscr{T}}(T_i, x)$ $(i \neq j)$, $T_j \dot{\prec}^x_{\mathscr{T}} T_i$.*

Lemma 122. *Given T_i, T_j s.t. $\hat{T}_j \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$, $\forall T_k$ if $\hat{T}_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_j)$, then $\hat{T}_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$ and $LVis(\hat{S}_H, T_i)|T_k = LVis(\hat{S}_H, T_j)|T_k$*

Proof. If T_k is committed in $\mathscr{T}$ and $\hat{T} \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_j,)$ then $\hat{S}_H|\hat{T}_k \subseteq LVis(\hat{S}_H, T_j,)$ and $\hat{T}_k \prec_{\hat{S}_H} \hat{T}_j$. Since $T_j \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i,)$, then $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_i$. Since T_k is committed in $\mathscr{T}$ and $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_i$, then $\hat{S}_H|\hat{T}_k \subseteq LVis(\hat{S}_H, T_i)$.

If T_k is not committed in $\mathscr{T}$ and $\hat{T} \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_j,)$ then $\hat{S}_H \overset{\diamond}{|} \hat{T}_k = LVis(\hat{S}_H, T_j, |)T_k$ and $\hat{T}_k \prec_{\hat{S}_H} \hat{T}_j$ and $T_k \not\prec_{\mathscr{T}} T_j$ and $\exists \xi(\mathscr{T}, T_k, T_j)$ (from Def. 61).

Since T_k is not committed in $\mathscr{T}$, and since $\mathscr{T}$ is commit abiding, then from Lemma 124, there cannot be $\xi(\mathscr{T}, T_k, T_j)$ s.t. T_j is committed. Thus T_j is not committed in $\mathscr{T}$. Thus, if $\hat{S}_H \overset{\diamond}{|} \hat{T}_j$ then $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_i$ and $T_j \not\prec T_i$ and $\exists \xi(\mathscr{T}, T_j, T_i)$.

If $\exists \xi(\mathscr{T}, T_k, T_j)$ and $\exists \xi(\mathscr{T}, T_j, T_i)$ then $\exists \mathscr{T} T_k T_i$.

Either T_k aborts in $\mathscr{T}$ (i.e. $res_k\big[A_k\big]$) or T_k is live in $\mathscr{T}$. In the latter case trivially $T_k \not\prec_{\mathscr{T}} T_i$. In the former case, from Lemma 125, also $T_k \not\prec_{\mathscr{T}} T_i$.

Since $\exists \xi(\mathscr{T}, T_{T_k}, T_{T_i})$ and $\hat{T}_k \prec_{\hat{S}_H} \hat{T}_i$ and $T_k \not\prec_{\mathscr{T}} T_i$ then $\hat{S}_H \overset{\diamond}{|} \hat{T}_k = LVis(\hat{S}_H, T_i, |)T_k$ (from Def. 61). □

Lemma 123. *Given T_i, T_j s.t. $\hat{T}_j \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$, $\forall T_k$ if $\hat{T}_k \overset{\diamond}{\not\subseteq} LVis(\hat{S}_H, T_j)$ and $\exists \xi(\mathscr{T}, T_{T_k}, T_{T_j})$ then $\hat{T}_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$.*

Proof. If $T_k \overset{\diamond}{\not\subseteq} LVis(\hat{S}_H, T_j)$ and $\hat{T}_k \prec_{\hat{S}_H} \hat{T}_j$ then $\hat{T}_k$ is not committed in $\mathscr{T}$.

If $\hat{S}_H \overset{\diamond}{|} \hat{T}_k \not\subset LVis(\hat{S}_H, T_j)$ and $\hat{T}_k \prec_{\hat{S}_H} \hat{T}_j$ then either $T_k \prec_{\mathscr{T}} T_j$ or $\nexists \xi(\mathscr{T}, T_i, T_j)$. The latter case contradicts the assumptions of the lemma, hence $T_k \prec_{\mathscr{T}} T_j$.

If $T_k \prec_{\mathscr{T}} T_j$, then $\exists r = res_k\big[A_k\big] \in \mathscr{T}|T_k$ s.t. for every event e in $\mathscr{T}|T_j$, $r \prec_{\mathscr{T}} e$. Since $\exists \xi(\mathscr{T}, T_j, T_i)$ then there is some view event e^i_v in $\mathscr{T}|T_i$ and some update event e^j_u in $\mathscr{T}|T_j$ s.t. $e^j_u \prec_{\mathscr{T}} e^i_v$. Therefore $r \prec_{\mathscr{T}} e^j_u \prec_{\mathscr{T}} e^i_v$.

Since no events can occur in $\mathscr{T}|T_k$ after v, then for all events in e in $\mathscr{T}|T_k$ apart from r, $e \prec_{\mathscr{T}} r$. So, for any $\xi(\mathscr{T}, T_k, T_i)$ for any update event $e_u^k = \circ s_k(x)v \in \mathscr{T}|T_k$, $e_u^k \prec_{\mathscr{T}} r \prec_{\mathscr{T}} e_v^i$.

From abort coda, $\exists e_a^l = \maltese s_l(x)v'$ s.t. $e_u^k \prec_{\mathscr{T}} e_a^l \prec_{\mathscr{T}} r$, and, from conservatism and unique routine updates, $v \neq v'$. Thus, since $e_u^k \prec_{\mathscr{T}} e_a^l \prec_{\mathscr{T}} e_v^i$ and $v \neq v'$, there cannot be such $\xi(\mathscr{T}, T_k, T_i)$ that satisfies chain isolation, and therefore $\nexists \xi(\mathscr{T}, T_k, T_i)$.

Therefore, from Lemma 61, $\hat{S}_H|\hat{T}_k \not\subseteq LVis(\hat{S}_H, T_i)$.

Thus, $\hat{S}_H|\hat{T}_i \not\subseteq LVis(\hat{S}_H, T_i)$. □

Lemma 124. *Given $\xi(\mathscr{T}, T_i, T_j)$, if T_j is committed in $\mathscr{T}$, then $\forall T_k \in \xi(\mathscr{T}, T_i, T_j)$, T_k is committed in $\mathscr{T}$.*

Proof. Given a pair of transaction $T_l, T_m \in \mathscr{T}$ s.t. $T_m \curvearrowleft T_l$, from commit accord, if T_m is committed in $\mathscr{T}$, then T_l is also committed in $\mathscr{T}$.

If $\xi(\mathscr{T}, T_i, T_j) = T_i \cdot T_j$, then n since T_j is committed in $\mathscr{T}$, then so is T_i.

Since, $\xi(\mathscr{T}, T_i, T_j) = T_i \xi(\mathscr{T}, T_i, T_k) \cdot T_j$ and T_k is such that $T_j \curvearrowleft T_k$, then since T_j is committed in $\mathscr{T}$, then so is T_k. This follows recursively for $\xi(\mathscr{T}, T_i, T_k)$.

Thus every transaction in $\xi(\mathscr{T}, T_i, T_j)$ is committed in $\mathscr{T}$. □

Lemma 125. *Given $\xi(\mathscr{T}, T_i, T_j)$, if T_i aborts in $\mathscr{T}$, then $T_i \not\prec_{\mathscr{T}} T_j$.*

Proof. Assume for the sake of contradiction that $T_i \prec_{\mathscr{T}} T_j$.

Thus, there exists $T_k \in \xi(\mathscr{T}, T_i, T_j)$ s.t. $T_k \curvearrowleft T_i$, so $\exists e_u^i = \circ s_i(x)v \in \mathscr{T}|T_i$ and $e_v^k = g_k(x)v \in \mathscr{T}|T_k$ and $e_u^i \prec_{\mathscr{T}} e_v^k$.

If T_i is aborted, then, from abort coda, $\exists e_a^l = \circ s_l(x)v'$ s.t. $e_u^i \prec_{\mathscr{T}} e_a^l \prec_{\mathscr{T}} res_i[A_i]$ and from unique routine updates $v \neq v'$.

Since T_j is in $\xi(\mathscr{T}, T_i, T_j)$, $\exists e_v^j = g_j(y)v''$ and since $T_i \prec_{\mathscr{T}} T_j$, then $res_i[A_i] \prec_{\mathscr{T}} e_v^j$. Thus, $e_u^i \prec_{\mathscr{T}} e_a^l \prec_{\mathscr{T}} e_v^j$.

This contradicts chain isolation, so it is not true that $T_i \prec_{\mathscr{T}} T_j$, so $T_i \not\prec_{\mathscr{T}} T_j$. □

Main Lemmas

Let there be a harmonious trace $\mathscr{T}$ and $H = Hist(\mathscr{T})$ from which $\hat{S}_H$ is generated. Let there be such $T_i \in \mathscr{T}$ that T_i is committed in $\mathscr{T}$. Then:

Lemma 126 (Unique Routine Updates)**.** *If $\mathscr{T}$ is consonant, and $\mathscr{T}$ has unique writes, then given any $s_i(x)v$ and $s_j(x)v'$ s.t. $v \neq v'$.*

Proof. Since both events are consonant, then for $s_i(x)v$ there exists $op_i = w_i(x)v^i \rightarrow ok_i$ s.t. $v = v^i$, and for $s_j(x)v'$ there exists $op_j = w_j(x)v^j \rightarrow ok_j$ s.t. $v' = v^j$. Since $\mathscr{T}$ has unique writes, then $v^i \neq v^j$, so $v \neq v'$. □

Lemma 127 (Non-local Read Consistency)**.** *For any $op_i \in \mathscr{T}|T_i$ s.t. $op_i = r_i(op_i) \rightarrow v$ and op_i is non-local, then either:*

1. *$v \neq 0$ and $\exists op_j \in Vis(\hat{S}_H, \hat{T}_i)$ for some T_j s.t. $op_j = w_j(x)v \rightarrow ok_j$, $op_j \lessdot_{Vis(\hat{S}_H, \hat{T}_i)} op_i$, or*
2. *$v = 0$ and $\nexists op_j \in Vis(\hat{S}_H, \hat{T}_i)$ s.t. $op_j = w_j(x)v \rightarrow ok_j$ and $op_j \prec_{Vis(\hat{S}_H, \hat{T}_i)} op_i$.*

Proof for Lemma 127. Since op_i is consonant and non-local, then $\exists e_v = g_i(x)v \in \mathscr{T}$, s.t. $op_i \rightsquigarrow e_v$ and e_v is consonant. Then, from e_v's consonance, either:

a) $v = 0$ and $\nexists e_u = s_j(x)v' \in \mathscr{T}$ for some $T_j \in \mathscr{T}$ s.t. $e_u \prec_{\mathscr{T}} e_v$.

In which case, if $\nexists op_j w_j(x)v' \to ok_i \in \mathscr{T}|T_j$ s.t. $op_j \prec_{\mathscr{T}} g_i(x)v$, then, $\nexists T_j \in \mathscr{T}$ s.t. $T_j \dot{\prec}_{\mathscr{T}} T_i$ and $op_j \in \mathscr{T}|T_j$. Thus, from construction of $\hat{S}_H$, $\nexists \hat{T}_j \in \mathscr{T}$ s.t. $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_i$ and $op_j \in \hat{S}_H|\hat{T}_j$. Thus, from construction of $Vis(\hat{S}_H, \hat{T}_i)$, for any such T_j, $\hat{S}_H|\hat{T}_j \not\subseteq Vis(\hat{S}_H, \hat{T}_i)$, so for any such T_j, $\underline{w_j(x)v \to ok_j \notin Vis(\hat{S}_H, \hat{T}_i) \text{ and } v = 0}$.

On the other hand, if $\exists op_j w_j(x)v' \to ok_i \in \mathscr{T}|T_j$ s.t. $op_j \prec_{\mathscr{T}} g_i(x)v$, then if T_i is committed in $\mathscr{T}$, then, from the definition of commit write obbligato, $\circ s_j(x)v' \in \mathscr{T}|T_j$, which contradicts the assumption of case *a*)). Thus, T_i is not committed in $\mathscr{T}$, so $\hat{T}_i$ is not committed in $\hat{S}_H$, and therefore $\hat{S}_H|\hat{T}_i \not\subseteq Vis(\hat{S}_H, \hat{T}_j)$. Thus for any such T_j, $\underline{w_j(x)v' \to ok_j \notin Vis(\hat{S}_H, \hat{T}_i) \text{ and } v = 0}$.

b) $v \neq 0$ and $\exists e_u = \circ s_j(x)v \in \mathscr{T}$ for some $T_j \in \mathscr{T}$ s.t. $e_u \lessdot_{\mathscr{T}} e_v$ and e_u is consonant.

Since e_u is consonant, then $\exists op_j = w_j(x)v \to ok_j \in \mathscr{T}|T_j$ s.t. op_j is non-local and consonant, and $op_j \lessdot \circ s_j(x)v$. Thus, since $e_u \lessdot_{\mathscr{T}} e_v$, $T_j \dot{\prec}^x_{\mathscr{T}} T_i$, then, by construction, $\hat{T}_j \prec \hat{S}_H \hat{T}_i$.

Since T_i is committed in $\mathscr{T}$ and $T_i \ddot{\sim} T_j$, and since $\mathscr{T}$ is commit-abiding, then T_j must be committed in $\mathscr{T}$. Thus $\hat{T}_j$ is also committed in $\hat{S}_H$. Thus, $\hat{S}_H|\hat{T}_j \subseteq Vis(\hat{S}_H, T_i)$, and therefore $op_j \prec_{Vis(\hat{S}_H, T_i)} op_i$. Then, from Lemma 117, $op_j \lessdot_{Vis(\hat{S}_H, T_i)} op_i$. Thus, $\underline{w_j(x)v \to ok_j \lessdot_{Vis(\hat{S}_H, \hat{T}_i)} op_i \text{ and } v \neq 0}$.

c) $\exists e_a = \natural s_j(x)v \in \mathscr{T}$ for some $T_j \in \mathscr{T}$ s.t. $e_a \lessdot_{\mathscr{T}} e_v$ and e_u is consonant.

Given $\psi_{\mathscr{T}}(T_i, x)$, from Lemma 119, $\forall T_k \in \psi_{\mathscr{T}}(T_i, x)$, T_k is aborted or live in $\mathscr{T}$. So, by construction, $\hat{T}_k$ is aborted in $\hat{S}_H$, and therefore excluded from $Vis(\hat{S}_H, T_i)$. Thus for any $T_k \in \psi_{\mathscr{T}}(T_i, x)$, $\forall op_k = w_k(x)v \to \in \mathscr{T}|T_k$, $op_k \notin Vis(\hat{S}_H, T_i)$.

Given $\psi_{\mathscr{T}}(T_i, x)$, from Lemma 118, $\exists e'_v = g_k(x)v \in \mathscr{T}|T_k$ s.t. T_k is the first element of $\psi_{\mathscr{T}}(T_i, x)$ that is initial and non-local, and either of the following is true:

i) $v = 0$ and $\nexists T_l \in \mathscr{T}$ s.t. $e'_u = s_l(x)v \in \mathscr{T}|T_l$ and $e'_u \prec_{\mathscr{T}} e'_v$.

Then, either $\exists e'_u \in s_l(x)v \in \mathscr{T}|T_l$ and $e'_v \prec_{\mathscr{T}} e'_u$ or $\nexists e'_u = s_l(x)v \in \mathscr{T}|T_l$.

If $\exists e'_u \in s_l(x)v' \in \mathscr{T}|T_l$ and $e'_v \prec_{\mathscr{T}} e'_u$, then from Lemma 118, $e'_u = \circ s_l(x)v'$. Thus, be definition of isolation order, $T_k \dot{\prec}^x_{\mathscr{T}} T_l$. Thus, if $T_l \dot{\prec}^x_{\mathscr{T}} T_j$, then, from Lemma 120, T_l is aborted or live in $\mathscr{T}$, so, by construction, $\hat{T}_l$ is aborted in $\hat{S}_H$. Therefore $\hat{T}_l \notin Vis(\hat{S}_H, T_i)$, so for any write operation execution $op_l = w_l(x)v' \to ok_l$ in any such T_l, $\underline{op_l \notin Vis(\hat{S}_H, T_i) \text{ (and } v = 0)}$. Alternatively, if $T_j \dot{\prec}^x_{\mathscr{T}} T_l$, then since $e_a \lessdot_{\mathscr{T}} e_v$, then it is not possible that $e_a \prec_{\mathscr{T}} e'_u \prec e_v$. By corollary, from the definition of isolation order, it is not possible that $T_j \dot{\prec}^x_{\mathscr{T}} T_l \dot{\prec}^x_{\mathscr{T}} T_i$. Then, by construction, $\hat{T}_i \prec_{\hat{S}_H} \hat{T}_l$, so $\hat{T}_l \notin Vis(\hat{S}_H, T_i)$. Therefore, for any write operation execution $op_l = w_l(x)v' \to ok_l$ in any such T_l, $\underline{op_l \notin Vis(\hat{S}_H, T_i) \text{ (and } v = 0)}$.

On the other hand, if $\nexists e'_u \in s_l(x)v \in \mathscr{T}|T_l$, then either $\mathscr{T}|T_l$ contains some write operation $op_l = w_l(x)v' \to ok_l$ or it does not. If it does not, then vacuously, for any write operation execution $op_l = w_l(x)v' \to ok_l$ in any such T_l, $\underline{op_l \notin Vis(\hat{S}_H, T_i) \text{ (and } v = 0)}$. On the other hand, if $op_l \in \mathscr{T}|T_l$, then from commit write obbligato, since $\nexists e'_u = \circ s_l(x)v \in \mathscr{T}|T_l$, then T_l is not committed in $\mathscr{T}$. Thus, $\hat{T}_l$ is aborted in $\hat{S}_H$ and $\hat{T}_l \notin Vis(\hat{S}_H, T_i)$. Thus, for any write operation execution $op_l = w_l(x)v' \to ok_l$ in any such T_l, $\underline{op_l \notin Vis(\hat{S}_H, T_i) \text{ (and } v = 0)}$.

ii) $v \neq 0$ and $\exists T_l \in \mathscr{T}$ s.t. $e'_u = \circ s_l(x)v \in \mathscr{T}|T_l$ and $e'_u \lessdot_{\mathscr{T}} e'_v$.

Since $\mathscr{T}$ is consonant, then e'_u is consonant, so $\exists op_l = w_l(x)v \to ok_l$ s.t $op_l \rightsquigarrow e'_u$.

Since for all $T_m \in \psi_{\mathscr{T}}(T_i, x)$, $e''_v = g_m(x)v''$ there is some T_n that directly

precedes T_m in $\psi_{\mathscr{T}}(T_i, x)$ and contains $e''_a = \text{¤}s_n(x)v''$ s.t. $e''_a \lessdot_{\mathscr{T}} e''_v$. Since e''_a is conservative, there is a preceding view $e'''_v = g_n(x)v''$ s.t. $e'''_v \lessdot_{\mathscr{T}} e''_a$. Thus $e'''_v \lessdot_{\mathscr{T}} e''_v$, so $T_n \dot{\prec}^x_{\mathscr{T}} T_m$. Therefore, $T_k \dot{\prec}^x_{\mathscr{T}} T_i$, and, by extension, since $e'_u \lessdot_{\mathscr{T}} e'_v$, $T_l \dot{\prec}^x_{\mathscr{T}} T_k$, then $T_l \dot{\prec}^x_{\mathscr{T}} T_i$. Since T_i is committed in $\mathscr{T}$, then since $T_l \dot{\prec}^x_{\mathscr{T}} T_i$, then, from commit coherence, T_l is either committed or aborted in T_j.

Transaction $T_l i$ cannot be aborted in $\mathscr{T}$, as follows. Let us assume by contradiction that T_l is aborted (i.e. $\exists r_a = res_l\big[A_l\big] \in \mathscr{T}|T_l$). Then, since $\mathscr{T}$ has coda, then for some T_n, $\exists e'''_a = \text{¤}s_n(x)v'''$ s.t. $e'_u \prec_{\mathscr{T}} e'''_a \prec_{\mathscr{T}} r$. Since e'_a is consonant, then since it is clean and $T_l \dot{\prec}^x_{\mathscr{T}} T_k$, then there is no recovery event following e'_v and preceding e'_a. In addition from commit coherence, T_l must abort before T_i commits, so, by extension e'''_a must precede $res_i\big[C_i\big]$ in $\mathscr{T}$. Thus either $e'''_a \prec_{\mathscr{T}} e'_v$ or $e'_a \prec_{\mathscr{T}} e'''_a \prec_{\mathscr{T}} r_a$. In the former case, if $e'''_a \prec_{\mathscr{T}} e'_u$, then this contradicts that e'''_a is consonant (ending), and if $e'_u \prec_{\mathscr{T}} e'''_a \prec_{\mathscr{T}} e'_v$, it contradicts that $e'_u \lessdot_{\mathscr{T}} e'_v$. On the other hand, if $e'_a \prec_{\mathscr{T}} e'''_a \prec_{\mathscr{T}} r_a$, then one of three scenarios is possible. If for some $T_m \in \psi_{\mathscr{T}}(T_i, x)$, s.t. $i \neq m$ and $g_m(x)\square \prec_{\mathscr{T}} e'''_a \prec_{\mathscr{T}} \text{¤}s_m(x)\square$, then this contradicts that $\text{¤}s_m(x)\square$ is clean. Alternatively, if for a pair $T_m, T_n \in \psi_{\mathscr{T}}(T_i, x)$, s.t. T_m directly precedes T_n in $\psi_{\mathscr{T}}(T_i, x)$, and $\text{¤}s_m(x)\square \prec_{\mathscr{T}} e'''_a \prec_{\mathscr{T}} g_n(x)\square$, then this contradicts that $\text{¤}s_m(x)\square \lessdot_{\mathscr{T}} g_n(x)\square$. Finally, if $e_v \prec e'''_a$, then this violates abort coda of $\mathscr{T}$ (case b), and is also a contradiction. Thus, there cannot be such e'''_a, and therefore T_l cannot be aborted in $\mathscr{T}$.

Hence, T_l must be committed in $\mathscr{T}$. Then since $T_l \dot{\prec}^x_{\mathscr{T}} T_i$ (so $\hat{T}_l \prec_{\hat{S}_H} \hat{T}_i$), $op_l \in Vis(\hat{S}_H, T_i)$.

Since e'_u is non-local, then it is not followed in $\mathscr{T}|T_l$ by another $e''_u = \circ s_l(x)v$. Since $op_l \rightsquigarrow e'_u$, $\nexists op'_l = w_l(x)v' \to ok_l$ s.t. $op_l \prec_{\mathscr{T}|T_i} op'_l$.

Let T_m be a transaction in $\mathscr{T}$ s.t. $T_l \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_j$. If $T_m \in \psi_{\mathscr{T}}(T_i, x)$, then from Lemma 119, T_m is not committed in $\mathscr{T}$. If $T_m \notin \psi_{\mathscr{T}}(T_i, x)$, then from Lemma 120, T_m is also not committed in $\mathscr{T}$. In either case, $\hat{T}_m$ is aborted in $\hat{S}_H$. Thus, for any T_m s.t. $T_l \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_j$, $\hat{S}_H|\hat{T}_m \not\subseteq Vis(\hat{S}_H, i)$. Therefore, for any write operation execution $op_m = w_m(x)v'' \to ok_m \in \mathscr{T}|T_m$, $op_m \notin Vis(\hat{S}_H, T_i)$.

Let $T_m \in \mathscr{T}$ be any transaction s.t. $T_j \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_i$. Since e_a consonant, it is is needed, so $\exists e''_u = s_j(x)v'' \in \mathscr{T}|T_j$ s.t. $e''_u \prec_{\mathscr{T}} e_a$. In addition, since $T_j \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_i$, then by definition of isolation order, $\exists e = s_m(x)v''' \in \mathscr{T}|T_m$, or $e = g_m(x)v''' \in \mathscr{T}|T_m$, so $e''_u \prec_{\mathscr{T}} e \prec_{\mathscr{T}} e_v$. Since $e_a \lessdot_{\mathscr{T}} e_v$, then $e''_u \prec_{\mathscr{T}} e \prec_{\mathscr{T}} e_u$. Thus, by definition of abort accord, T_m is not committed in $\mathscr{T}$, so, by construction, $\hat{T}_m$ is aborted in $\hat{S}_H$. Thus, for any T_m s.t. $T_j \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_i$, $\hat{S}_H|\hat{T}_m \not\subseteq Vis(\hat{S}_H, i)$. Therefore, for any write operation execution $op_m = w_m(x)v'' \to ok_m \in \mathscr{T}|T_m$, $op_m \notin Vis(\hat{S}_H, T_i)$.

Since no other write operation execution follows op_l in $\mathscr{T}|T_m$, and since there is no transaction $T_m \in \mathscr{T}$ s.t. $\hat{T}_l \prec_{\hat{S}_H} \hat{T}_m \prec_{\hat{S}_H} \hat{T}_i$ (and therefore $\hat{T}_l \prec_{Vis(\hat{S}_H, \hat{T}_i)} \hat{T}_m \prec_{Vis(\hat{S}_H, \hat{T}_i)} \hat{T}_i$) s.t. $\exists op_m = w_m(x)v'' \to ok_m \in \mathscr{T}|T_m$ and $op_m \in Vis(\hat{S}_H, \hat{T}_i)$, then $w_l(x)v \to ok_l \lessdot_{Vis(\hat{S}_H, \hat{T}_i)} op_i$ and $v \neq 0$.

□

Corollary 25 (Total Non-local Read Consistency)**.** *By extension of the above, since, by definition, if for some sequential history S, $\hat{S}_H|tr_j \in Vis(S, T_i)$, then $Vis(\hat{S}_H, T_j)$ is a prefix of $Vis(S, T_i)$, then for any $op_j \in \mathscr{T}|T_j$ s.t. $op_j = r_i(op_j) \to v$ either:*

1. *$v \neq 0$ and $\exists op_k \in Vis(\hat{S}_H, T_k)$ for some T_k s.t. $op_k = w_k(x)v \to ok_j$, $op_k \lessdot_{Vis(\hat{S}_H, \hat{T}_i)} op_j$, or*

2. $v = 0$ *and* $\nexists op_k \in Vis(\hat{S}_H, T_i)$ *s.t.* $op_k = w_k(x)v \to ok_k$ *and* $op_k \prec_{Vis(\hat{S}_H,T_i)} op_j$.

Lemma 128 (Local Read Consistency)**.** *For any* $op_i \in \mathscr{T}|T_i$ *s.t.* $op_i = r_i(op_i) \to v$ *and* op_i *is local, then* $\exists op'_i \in Vis(\hat{S}_H, T_i)$ *s.t.* $op'_i = w_i(x)v \to ok_i$, $op'_i \lessdot_{Vis(\hat{S}_H,T_i)} op_i$.

Proof. From local read consonance it follows that For any $op_i \in \mathscr{T}|T_i$ s.t. op_i is local, then $\exists op'_i \in \mathscr{T}|T_i$ and $op'_i \lessdot_{\mathscr{T}|T_i} op_i$. Thus, $op'_i \lessdot_{\hat{S}_H|\hat{T}_i} op_i$. Since $T_i \subseteq Vis(\hat{S}_H, T_i)$ then $op'_i \lessdot_{Vis(\hat{S}_H,T_i)} op_i$. □

Corollary 26 (Total Local Read Consistency)**.** *For any* $op_i \in \mathscr{T}|T_i$ *s.t.* $op_i = r_i(op_i) \to v$ *and* op_i *is local, then for any* T_j, $\exists op'_i \in Vis(\hat{S}_H, T_j)$ *s.t.* $op'_i = w_i(x)v \to ok_i$, $op'_i \lessdot_{Vis(\hat{S}_H,T_j)} op_i$.

Let there instead be such $T_i \in \mathscr{T}$ that T_i is either committed or not committed $\mathscr{T}$. Then:

Lemma 129 (Non-local Read Last-use Consistency)**.** *For any* $op_i \in \mathscr{T}|T_i$ *s.t.* $op_i = r_i(op_i) \to v$ *and* op_i *is non-local, then either*

i) $v \neq 0$ *and* $\exists op_j \in LVis(\hat{S}_H, \hat{T}_i)$ *for some* T_j *s.t.* $op_j = w_j(x)v \to ok_j$, $op_j \lessdot_{LVis(\hat{S}_H,\hat{T}_i)} op_i$, *or*

ii) $v = 0$ *and* $\nexists op_j \in LVis(\hat{S}_H, \hat{T}_i)$ *s.t.* $op_j = w_j(x)v \to ok_j$ *and* $op_j \prec_{LVis(\hat{S}_H,\hat{T}_i)} op_i$.

Proof. Since op_i is consonant and non-local, then $\exists e_v = g_i(x)v \in \mathscr{T}$, s.t. $op_i \backsim e_v$. Then, since $\mathscr{T}$ is consonant, then e_v is also consonant, so one of the following is true:

a) $v = 0$ and $\nexists e_u = s_j(x)\square \in \mathscr{T}$ for some $T_j \in \mathscr{T}$ s.t. $e_u \prec_{\mathscr{T}} e_v$.
In which case, if $\nexists op_j = w_j(x)\square \to ok_i \in \mathscr{T}|T_j$ s.t. $op_j \prec_{\mathscr{T}} g_i(x)v$, then, $\nexists T_j \in \mathscr{T}$ s.t. $T_j \dot{\prec}_{\mathscr{T}} T_i$ and $op_j \in \mathscr{T}|T_j$. Thus, from construction of $\hat{S}_H$, $\nexists \hat{T}_j \in \mathscr{T}$ s.t. $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_i$ and $op_j \in \hat{S}_H|\hat{T}_j$. Thus, from construction of $LVis(\hat{S}_H, \hat{T}_i)$, for any such T_j, $\hat{S}_H|\hat{T}_j \not\subseteq LVis(\hat{S}_H, \hat{T}_i)$, so for any such T_j, $\underline{w_j(x)v \to ok_j \notin LVis(\hat{S}_H, \hat{T}_i) \text{ and } v = 0}$.
On the other hand, if $\exists op_j w_j(x)\square \to ok_i \in \mathscr{T}|T_j$ s.t. $op_j \prec_{\mathscr{T}} g_i(x)v$, then if T_j is committed in $\mathscr{T}$, then, from the definition of commit write obbligato, $\exists \circ s_j(x)\square \in \mathscr{T}|T_j$, which contradicts the assumption of case *a*)). If, however, T_j is not committed in $\mathscr{T}$, then either T_j is decided on x in $\mathscr{T}$, or it is not. In the former of those two cases, from the definition of closing write obbligato, $\exists \circ s_j(x)\square \in \mathscr{T}|T_j$, which also contradicts the assumption of case *a*)). In the latter case, since $\hat{T}_j$ is neither committed in $\mathscr{T}$ nor decided on x in $\mathscr{T}$, then neither is it committed in $\hat{S}_H$ nor decided on x in $\hat{S}_H$. Therefore, by definition of $LVis(\hat{S}_H, \hat{T}_i)$, $\hat{S}_H|\hat{T}_j|x \notin LVis(\hat{S}_H, \hat{T}_i)$. Thus for any such T_j, $\underline{w_j(x)\square \to ok_j \notin LVis(\hat{S}_H, \hat{T}_i) \text{ and } v = 0}$.

b) $v \neq 0$ and $\exists e_u = \circ s_j(x)v \in \mathscr{T}$ for some $T_j \in \mathscr{T}$ s.t. $e_u \lessdot_{\mathscr{T}} e_v$.
Since e_u is consonant, then $\exists op_j = w_j(x)v \to ok_j \in \mathscr{T}|T_j$ s.t. op_j is non-local and consonant, and $op_j \lessdot \circ s_j(x)v$. Thus, since $e_u \lessdot_{\mathscr{T}} e_v$, $T_j \dot{\prec}^x_{\mathscr{T}} T_i$, then, by construction, $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_i$.
If T_i is not committed in $\mathscr{T}$, since $T_i \dot{\backsim} T_j$, and since $\mathscr{T}$ is decisive, then T_j is decided on x in $\mathscr{T}$. Thus, from Def. 61, $\hat{S}_H|\hat{T}_j|x \subseteq LVis(\hat{S}_H, T_i)$, and therefore $op_j \prec_{LVis(\hat{S}_H,T_i)} op_i$. Alternatively, if T_i is committed in $\mathscr{T}$, then, from commit accord, T_j is also committed in $\mathscr{T}$. Then, by definition of $LVis(\hat{S}_H, T_i)$, $\hat{S}_H|\hat{T}_j \subseteq LVis(\hat{S}_H, T_i)$, and thus $op_j \prec_{LVis(\hat{S}_H,T_i)} op_i$. Then, from Lemma 117, $op_j \lessdot_{LVis(\hat{S}_H,T_i)} op_i$. Thus, $\underline{w_j(x)v \to ok_j \lessdot_{LVis(\hat{S}_H,\hat{T}_i)} op_i \text{ and } v \neq 0}$.

c) $\exists e^j_a = \text{¤}s_j(x)v \in \mathscr{T}|T_j$ s.t. $e^j_a \lessdot_{\mathscr{T}} e_v$.
Since $\mathscr{T}$ is consonant, then e^j_a is consonant, so e^j_a is conservative, and thus $\exists e^j_v = g_j(x)v \in \mathscr{T}|T_j$.

From Corollary 24 $\forall T_n \in \psi_{\mathscr{T}}(T_i, x)$ $(n \neq i)$ $T_n \dot{\prec}^x_{\mathscr{T}} T_i$. So, by construction, for every such T_n, $\hat{T}_n$ is aborted in $\hat{S}_H$ and therefore not included as a whole in $LVis(\hat{S}_H, T_i)$. In addition, since e^n_a is needed there is a preceding routine update $e^n_u = \circ s_n(x)v'$. Because of unique writes, $v' \neq v$. Therefore, it is not true that $T_i \dot{\curvearrowright} T_n$. Furthermore, since $e^n_u \prec_{\mathscr{T}} e^n_a$ and $e^n_a \prec_{\mathscr{T}} e^j_a$ and $e^j_a \prec_{\mathscr{T}} e_v$, then, $e^n_u \prec_{\mathscr{T}} e^n_a \prec_{\mathscr{T}} e_v$, so, from chain isolation, $\nexists \xi(\mathscr{T}, T_n, T_m)$. Thus, from Def. 61 $\hat{S}_H \mathring{|} T_n$ is not included in $LVis(\hat{S}_H, T_i)$. Thus for any $T_n \in \psi_{\mathscr{T}}(T_i, x)$ $(n \neq i)$, $\forall op_n = w_n(x)v \rightarrow \in \mathscr{T}|T_n$, $op_n \notin LVis(\hat{S}_H, T_i)$. Given $\psi_{\mathscr{T}}(T_i, x)$, from Lemma 118, $\exists e^k_v = g_k(x)v \in \mathscr{T}|T_k$ s.t. T_k is the first element of $\psi_{\mathscr{T}}(T_i, x)$ that is initial and non-local, and either of the following is true:

i) $v = 0$ and $\nexists T_l \in \mathscr{T}$ s.t. $e^l_u = s_l(x)\square \in \mathscr{T}|T_l$ and $e^l_u \prec_{\mathscr{T}} e^k_v$.
Then, either $\exists e^l_u = s_l(x)v \in \mathscr{T}|T_l$ and $e^k_v \prec_{\mathscr{T}} e^l_u$ or $\nexists e^l_u = s_l(x)\square \in \mathscr{T}|T_l$.
If $\exists e^l_u = s_l(x)\square \in \mathscr{T}|T_l$ and $e^k_v \prec_{\mathscr{T}} e^l_u$, then from Lemma 118, $e^l_u = \circ s_l(x)\square \in \mathscr{T}|T_l$. Thus, by definition of isolation order, $T_k \dot{\prec}^x_{\mathscr{T}} T_l$.
Thus, if $T_l \dot{\prec}^x_{\mathscr{T}} T_i$, then, from Lemma 120, T_l is aborted or live in $\mathscr{T}$, so, by construction, $\hat{T}_l$ is aborted in $\hat{S}_H$. Since $\hat{T}_l$ is not committed in $\hat{S}_H$, then $\hat{T}_l|\hat{S}_H$ is not included as a whole in $LVis(\hat{S}_H, T_i)$. Furthermore, $\hat{S}_H \mathring{|} T_l$ can be omitted from $LVis(\hat{S}_H, T_i)$. From unique routine updates, there cannot be $T_n \in \xi(\mathscr{T}, T_l, T_i)$ s.t. $\exists \circ s_n(x)0$, so since $v = 0$ and from self-containment, $\nexists \xi(\mathscr{T}, T_l, T_i)$. Thus, from Def. 61, $\hat{S}_H \mathring{|} T_k \nsubseteq LVis(\hat{S}_H, T_i)$. Then, for any write operation execution $op_l = w_l(x)v' \rightarrow ok_l$ in any such T_l, $\underline{op_l \notin LVis(\hat{S}_H, T_i) \text{ (and } v = 0)}$.
Alternatively, if $T_i \dot{\prec}^x_{\mathscr{T}} T_l$, then since $e_a \lessdot_{\mathscr{T}} e_v$, then $e_v \prec_{\mathscr{T}} e^l_u$, so $T_i \dot{\prec}^x_{\mathscr{T}} T_l$, and thus $T_i \prec_{\hat{S}_H} T_l$, which means that $\hat{S}_H \mathring{|} T_k \nsubseteq LVis(\hat{S}_H, T_i)$. In either case for any write operation execution $op_l = w_l(x)v' \rightarrow ok_l$ in any such T_l, $\underline{op_l \notin Vis(\hat{S}_H, T_i) \text{ (and } v = 0)}$.
On the other hand, if $\nexists e^l_u = s_l(x)\square \in \mathscr{T}|T_l$, either $T_l \dot{\prec}_{\mathscr{T}} T_i$, or $T_l \dot{\nprec}_{\mathscr{T}} T_i$. In the latter case, if $T_i \dot{\prec}_{\mathscr{T}} T_l$, then, trivially, no subset of $\hat{S}_H|\hat{T}_l$ is contained in $LVis(\hat{S}_H, T_l)$. If $T_i \dot{\nprec}_{\mathscr{T}} T_l$, then there does not exist $\xi(\mathscr{T}, T_l, T_i)$, so, from Def. 61, no subset of $\hat{S}_H|\hat{T}_l$ is contained in $LVis(\hat{S}_H, T_l)$. If $T_l \dot{\prec}_{\mathscr{T}} T_i$, then either $\mathscr{T}|T_l$ contains some write operation $op_l = w_l(x)v' \rightarrow ok_l$ or it does not. However, from view write obbligato, since $T_l \dot{\prec}_{\mathscr{T}} T_i$, there must be $s_l(x)\square \in \mathscr{T}|T_l$, so, there is no such op_l. Then vacuously, for any write operation execution $op_l = w_l(x)\square \rightarrow ok_l$ in any such T_l, $\underline{op_l \notin LVis(\hat{S}_H, T_i) \text{ (and } v = 0)}$.

ii) $v \neq 0$ and $\exists T_l \in \mathscr{T}$ s.t. $e^l_u = \circ s_l(x)v \in \mathscr{T}|T_l$ and $e'_u \lessdot_{\mathscr{T}} e'_v$.
Since $\mathscr{T}$ is consonant, then e^l_u is consonant, so $\exists op_l = w_l(x)v \rightarrow ok_l$ s.t $op_l \rightsquigarrow e^l_u$.
Let us first assume that T_l is committed in $\mathscr{T}$. Then since $T_l \dot{\prec}^x_{\mathscr{T}} T_i$ (so $\hat{T}_l \prec_{\hat{S}_H} \hat{T}_i$), $op_l \in LVis(\hat{S}_H, T_i)$.
If, on the other hand, T_l is not committed in $\mathscr{T}$, then, since e^k_v is consonant, then e^l_u is the ultimate routine update event in $\mathscr{T}|T_l$. Therefore, from decisiveness, e^l_u is either the closing routine update event on x in $\mathscr{T}|T_l$, or $e^l_u \prec_{\mathscr{T}} res_l[C_l] \prec_{\mathscr{T}} e_v$. Since T_l is not committed in $\hat{S}_H$, e^l_u is the closing routine update event, so op_l is the closing write on x in T_l. Because of this, and since $\hat{T}_l \prec_{\hat{S}_H} \hat{T}_i$, $\hat{S}_H \mathring{|} T_l$ can be included in $LVis(\hat{S}_H, T_i)$. Then, since $T_i \dot{\curvearrowright} T_l$, then there exists $\xi(\mathscr{T}, T_l, T_i)$, so according to Def. 61, $\hat{S}_H \mathring{|} \hat{T}_l$ is included in $LVis(\hat{S}_H, T_i)$, and therefore $\underline{op_l \in LVis(\hat{S}_H, T_i)}$.
From minimalism, e^l_u is not followed in $\mathscr{T}|T_l$ by another $\circ s_l(x)\square$. Since $op_l \rightsquigarrow e^l_u$, $\nexists op'_l = w_l(x)\square \rightarrow ok_l$ s.t. $op_l \prec_{\mathscr{T}|T_i} op'_l$.
Let T_m be any transaction in $\mathscr{T}$ s.t. $T_l \dot{\prec} \mathscr{T} T_m \dot{\prec} \mathscr{T} T_i$. If $\nexists w_m(x)\square \rightarrow ok_m \in \mathscr{T}|T_m$, then trivially, for any such T_m $\nexists w_m(x)v \rightarrow ok_m \in LVis(\hat{S}_H, T_i)$. Thus,

let there be $op_m = w_m(x)v' \to ok_m \in \mathscr{T}|T_m$.
If $T_m \in \psi_{\mathscr{T}}(T_{\mathscr{T}}, x)i$, then, as shown above, $\hat{S}_H|\hat{T}_m \not\subseteq LVis(\hat{S}_H, T_i)$ and therefore $op_m \notin LVis(\hat{S}_H, T_i)$. Hence, let $T_m \notin \psi_{\mathscr{T}}(T_{\mathscr{T}}, x)i$.
Either $tr_m \dot{\prec}_{\mathscr{T}} T_i$ or $tr_m \dot{\not\prec}_{\mathscr{T}} T_i$. In the latter case, if T_m is not committed, T_m can be excluded from $LVis(\hat{S}_H, T_i)$. Since in that case there doe not exist $\xi(\mathscr{T}, T_m, T_i)$, then, by Def. 61, $\hat{S}_H|\hat{T}_m \not\subseteq LVis(\hat{S}_H, T_i)$ and therefore $op_m \notin LVis(\hat{S}_H, T_i)$. If T_m is committed, then if $w_m(x)v' \to ok_m \in \mathscr{T}|T_m$, then by commit write obbligato, there would have to exist $e_u^m = \circ s_m(x)v' \in \mathscr{T}|T_m$, which would imply that $T_m \dot{\prec}^x_{\mathscr{T}} T_i$, which contradicts the assumption that $tr_m \dot{\not\prec}_{\mathscr{T}} T_i$. Therefore, there is no such transaction.
If $T_m \dot{\prec}_{\mathscr{T}} T_i$, then, if $T_m \dot{\prec}_{\mathscr{T}} T_l$, then $\hat{T}_m \prec_{\hat{S}_H} \hat{T}_l$, and therefore $op_m \prec_{LVis(\hat{S}_H, T_i)} op_l$, which has no bearing on whether op_i is preceded by a corresponding write operation. Hence, let $T_l \dot{\prec}_{\mathscr{T}} T_m \dot{\prec}_{\mathscr{T}} T_i$. Then, T_m is either committed in $\mathscr{T}$ or not.
If T_m is committed, then from commit write obbligato $\exists e_u^m = \circ s_m(x)v' \in \mathscr{T}|T_m$. From isolation, it is impossible that $T_i \dot{\prec}^x_{\mathscr{T}} T_m$ or $T_m \dot{\prec}^x_{\mathscr{T}} T_l$, then $T_l \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_i$. Then, if $T_j \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_i$, since e_a is consonant, it is is needed, so $\exists e_u^j = s_j(x)\square \in \mathscr{T}|T_j$ s.t. $e_u^j \prec_{\mathscr{T}} e_a$. In addition, since $T_j \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_i$, then by definition of isolation, $\exists e = s_m(x)\square \in \mathscr{T}|T_m$, or $eg_m(x)\square \in \mathscr{T}|T_m$, so $e_u^j \prec_{\mathscr{T}} e \prec_{\mathscr{T}} e_v$. Since $e_a \lessdot_{\mathscr{T}} e_v$, then $e_u^j \prec_{\mathscr{T}} e \prec_{\mathscr{T}} e_u$. Thus, by definition of abort accord, T_m cannot be in $\mathscr{T}$, thus there is no such T_m.
If $T_l \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_j$, then, since $T_m \notin \psi_{\mathscr{T}}(T_{\mathscr{T}}, x)i$, then, from Lemma 120, T_m cannot be committed in $\mathscr{T}$, thus, there is also no such T_m.
Since there is no such T_m that is both committed and contains an operation execution such as op_m, then for any such T_m, $\hat{S}_H|\hat{T}_m \not\subseteq LVis(\hat{S}_H, T_i)$ and therefore $op_m \notin LVis(\hat{S}_H, T_i)$.
On the other hand, if T_m is not committed in $\mathscr{T}$, then, since op_m is consonant, either $op_m \rightsquigarrow e_u^m$ where $e_u^m = \circ s_m(x)v' \in \mathscr{T}|T_m$, or $\nexists \circ s_m(x)\square \in \mathscr{T}|T_m$. Since, from view write obbligato the latter case is impossible, then $\exists e_u^m = \circ s_m(x)v' \in \mathscr{T}|T_m$. Then, from the definition of isolation order, $T_l \dot{\prec}^x_{\mathscr{T}} T_m \dot{\prec}^x_{\mathscr{T}} T_i$. Since T_m is not committed, then $\hat{T}_m$ can be omitted in $LVis(\hat{S}_H, T_i)$. Due to unique routine updates, there cannot be any $T_n \in \mathscr{T}$ s.t. $\circ s_n(x)v''$ where $v'' = v$. Therefore, given any $\xi(\mathscr{T}, T_m, T_i)$ and there is no transaction to satisfy self-containment. Thus, there is no such $\xi(\mathscr{T}, T_m, T_i)$. Thus, by Def. 61, $\hat{S}_H|\hat{T}_m \not\subseteq LVis(\hat{S}_H, T_i)$ and therefore $op_m \notin LVis(\hat{S}_H, T_i)$.
Because there is no T_m s.t. $\hat{S}_H|\hat{T}_m \subseteq LVis(\hat{S}_H, T_i)$ and $op_m \in LVis(\hat{S}_H, T_i)$, then there is no write operation execution op' on x s.t. $op_l \prec_{LVis(\hat{S}_H, T_i)} op' \prec_{LVis(\hat{S}_H, T_i)} op_i$. Therefore $w_l(x)v \to ok_l \lessdot_{Vis(\hat{S}_H, \hat{T}_i)} op_i$ and $v \neq 0$.

□

Lemma 130 (All Non-local Read Last-use Consistency)**.** *If for some sequential history S, $\hat{S}_H|tr_j \in Vis(S, T_i)$, for any $op_j \in \mathscr{T}|T_j$ s.t. $op_j = r_i(op_j) \to v$ either:*

1. *$v \neq 0$ and $\exists op_k \in Vis(\hat{S}_H, \hat{T}_k)$ for some T_k s.t. $op_k = w_k(x)v \to ok_j$, $op_k \lessdot_{Vis(\hat{S}_H, \hat{T}_i)} op_j$, or*
2. *$v = 0$ and $\nexists op_k \in Vis(\hat{S}_H, \hat{T}_i)$ s.t. $op_k = w_k(x)v' \to ok_k$ and $op_k \prec_{Vis(\hat{S}_H, \hat{T}_i)} op_j$.*

Proof. Either $v \neq 0$ or $v = 0$.

1. If $v \neq 0$,from Lemma 129, since $v \neq 0$ then $\exists op_k = w_k(x)v \to ok_k \in \mathscr{T}|T_k$ s.t. $op_k \lessdot_{LVis(\hat{S}_H, T_j)} op_j$.

From Lemma 122, $\forall T_l$ if $\hat{T}_l \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_j)$ then $\hat{T}_l \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$ and $LVis(\hat{S}_H, T_i)|T_l = LVis(\hat{S}_H, T_j)|T_l$. Hence, $T_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$ and $op_k \in LVis(\hat{S}_H, T_i)$. Furthermore, if $T_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_j)$, then $T_k \prec_{\hat{S}_H} T_j$, so $op_k \prec_{LVis(\hat{S}_H,T_i)} op_j$.
For the sake of contradiction, let us assume there exists $op_l = w_l(x)v \to ok_l \in \mathscr{T}|T_l$ s.t. $op_k \prec_{LVis(\hat{S}_H,T_i)} op_l \prec_{LVis(\hat{S}_H,T_i)} op_j$. Since from Lemma 129, there is no such transaction in $LVis(\hat{S}_H, T_i)$, then T_l is such that $\hat{T}_l \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$ s.t. and $\hat{T}_l \overset{\diamond}{\nsubseteq} LVis(\hat{S}_H, T_j)$ (and $\hat{T}_l \prec_{\hat{S}_H} \hat{T}_j$).

If $\hat{T}_l \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$, then, by definition, T_l is not committed in $\mathscr{T}$.
If $T_l \dot{\prec} x \mathscr{T} T_j$, then this is a contradiction by analogy to Lemma 129.
If $T_j \dot{\prec} x \mathscr{T} T_l$, then, from Def. 60, $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_l$, which implies that $op_j \prec_{LVis(\hat{S}_H,T_j)} op_l$, which is a contradiction.
If $T_l \dot{\not\prec} x \mathscr{T} T_j$, then $\nexists e_u^k = \circ s_l(x)\square \in \mathscr{T}|T_l$. Hence, from Def. 60 , $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_l$, which imp'lies that $op_j \prec_{LVis(\hat{S}_H,T_j)} op_l$, which, again, is a contradiction.
Thus, there is no such T_l, and, therefore, $op_k \preccurlyeq_{LVis(\hat{S}_H,T_i)} op_j$ (and $v \neq 0$).

2. If $v = 0$, let us assume by contradiction, that there exists such T_k and op_k. From Lemma 129, since $v = 0$ then $\nexists op_l = w_l(x)\square \to ok_l \in \mathscr{T}|T_l$ s.t. $op_l \prec_{LVis(\hat{S}_H,T_j)} op_j$.
Hence, T_k must be such that $\hat{T}_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$ s.t. and $\hat{T}_k \overset{\diamond}{\nsubseteq} LVis(\hat{S}_H, T_j)$ (and $T_k \prec_{\hat{S}_H} T_j$).

From Lemma 123, $\forall T_l$ if $\hat{T}_l \overset{\diamond}{\nsubseteq} LVis(\hat{S}_H, T_j)$ and $\exists \xi(\mathscr{T}, T_l, T_j)$ then $\hat{T}_l \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$.
Thus, if T_k is such that $\hat{T}_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_j)$ and $\hat{T}_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$, then $\nexists \xi(\mathscr{T}, T_k, T_j)$.
If $\hat{T}_k \overset{\diamond}{\subseteq} LVis(\hat{S}_H, T_i)$, then, by definition, T_k is not committed in $\mathscr{T}$.
If $T_k \dot{\prec} x \mathscr{T} T_j$, then this is a contradiction by analogy to Lemma 129.
If $T_j \dot{\prec} x \mathscr{T} T_k$, then, from Def. 60, $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_k$, which implies that $op_j \prec_{LVis(\hat{S}_H,T_j)} op_k$, which is a contradiction.
If $T_k \dot{\not\prec} x \mathscr{T} T_j$, then $\nexists e_u^k = \circ s_k(x)\square \in \mathscr{T}|T_k$. Hence, from Def. 60, $\hat{T}_j \prec_{\hat{S}_H} \hat{T}_k$, which implies that $op_j \prec_{LVis(\hat{S}_H,T_j)} op_k$, which, again, is a contradiction.
Thus, there is no such T_k, and, therefore, $\nexists op_k \in LVis(\hat{S}_H, \hat{T}_i)$ s.t. $op_k = w_k(x)v' \to ok_k$ and $op_k \prec_{LVis(\hat{S}_H,\hat{T}_i)} op_j$ (and $v = 0$).

□

Lemma 131 (Local Read Last-use Consistency)**.** *For any $op_i \in \mathscr{T}|T_i$ s.t. $op_i = r_i(op_i) \to v$ and op_i is local, then $\exists op_i' \in LVis(\hat{S}_H, T_i)$ s.t. $op_i' = w_i(x)v \to ok_i$, $op_i' \preccurlyeq_{LVis(\hat{S}_H,T_i)} op_i$.*

Proof. From local read consonance it follows that for any $op_i \in \mathscr{T}|T_i$ s.t. op_i is local, then $\exists op_i' \in \mathscr{T}|T_i$ and $op_i' \preccurlyeq_{\mathscr{T}|T_i} op_i$. Thus, $op_i' \preccurlyeq_{\hat{S}_H|\hat{T}_i} op_i$. Since op_i and op_j operate on the same variable, then trivially, $op_i \in LVis(\hat{S}_H, T_i) \iff op_i' LVis(\hat{S}_H, T_i)$. Hence, $op_i' \preccurlyeq_{Vis(\hat{S}_H,T_i)} op_i$. □

Corollary 27 (Total Local Read Last-use Consistency)**.** *For any $op_i \in \mathscr{T}|T_i$ s.t. $op_i = r_i(op_i) \to v$ and op_i is local, then for any T_j, $\exists op_i' \in LVis(\hat{S}_H, T_j)$ s.t. $op_i' = w_i(x)v \to ok_i$, $op_i' \preccurlyeq_{LVis(\hat{S}_H,T_j)} op_i$.*

Lemma 132 (Total Write Consistency)**.** *For any T_i, T_j, $\forall op_i \in LVis(\hat{S}_H, T_j)$ s.t. $op_i = w_i(op_i)v \to ok_i \in \mathscr{T}|T_i$, v is in the domain of x.*

Proof. Follows from write consonance. □

Proof for Theorem 9

Proof for Theorem 9: Trace Last-use Opacity. For every transaction $T_i \in \mathscr{T}$, given $\hat{S}_H$ constructed by Def. 60,

i) If T_i is committed in H, from Corollary 25, $\forall op_j \in H|T_j$ s.t. $op_j = r_i(x) \to v$ and op_j is non-local, either:

  a) $v \neq 0$ and $\exists op_k \in Vis(\hat{S}_H, \hat{T}_k)$ for some T_k s.t. $op_k = w_k(x)v \to ok_j$, $op_k \lessdot_{Vis(\hat{S}_H,\hat{T}_i)} op_j$, or

  b) $v = 0$ and $\nexists op_k \in Vis(\hat{S}_H, \hat{T}_i)$ s.t. $op_k = w_k(x)v \to ok_k$ and $op_k \prec_{Vis(\hat{S}_H,\hat{T}_i)} op_j$.

  In addition, from Corollary 26, $\forall op_j \in H|T_j$ s.t. $op_j = r_i(op_j) \to v$ and op_j is local, $\exists op_j \in Vis(\hat{S}_H, \hat{T}_j)$ s.t. $op'_j = w_j(x)v \to ok_j$, $op'_j \lessdot_{Vis(\hat{S}_H,\hat{T}_i)} op_j$
  Furthermore, from Lemma 132, $\forall op'_j \in H|T_j$ s.t. $op_j = w_i(op_j)v \to ok_j$, v is in the domain of x.

  Thus, $Vis(\hat{S}_H, T_i)$ is legal, and therefore T_i is legal.

ii) If T_i is not committed in H, from Lemma 130, $\forall op_j \in H|T_j$ s.t. $op_j = r_i(x) \to v$ and op_j is non-local, either:

  a) $v \neq 0$ and $\exists op_k \in LVis(\hat{S}_H, \hat{T}_k)$ for some T_k s.t. $op_k = w_k(x)v \to ok_j$, $op_k \lessdot_{LVis(\hat{S}_H,\hat{T}_i)} op_j$, or

  b) $v = 0$ and $\nexists op_k \in LVis(\hat{S}_H, \hat{T}_i)$ s.t. $op_k = w_k(x)v \to ok_k$ and $op_k \prec_{LVis(\hat{S}_H,\hat{T}_i)} op_j$.

  In addition, from Corollary 27, $\forall op_j \in H|T_j$ s.t. $op_j = r_i(x) \to v$ and op_j is local, $\exists op_j \in LVis(\hat{S}_H, \hat{T}_j)$ s.t. $op'_j = w_j(x)v \to ok_j$, $op'_j \lessdot_{LVis(\hat{S}_H,\hat{T}_i)} op_j$
  Furthermore, from Lemma 132, $\forall op'_j \in H|T_j$ s.t. $op_j = w_i(x)v \to ok_j$, v is in the domain of x.

  Thus, $LVis(\hat{S}_H, T_i)$ is legal, and therefore T_i is last-use legal.

Since every committed transaction $T_i \in H$ is legal if it is committed and last-use legal if it is not committed, then H is final-state last-use opaque.

Since a prefix of a harmonious $\mathscr{T}$ is trivially also harmonious, then for every prefix $\mathscr{T}'$ of $\mathscr{T}$, $H' = Hist(\mathscr{T}')$ is also final-state last-use opaque. Thus, H is last-use opaque. □

Miscellaneous

A History with Early Release is not Opaque

Let H_{er} represent the history in Fig. 7.1:

$$H_{er} = \left[\ start_i \to ok_i, start_j \to ok_j, w_i(x)1 \to ok_i, r_j(x) \to 1, tryC_i \to C_i, tryC_j \to C_j\ \right].$$

Lemma 133. *H_{er} is final-state opaque.*

Proof. The completion $Compl(H_{er})$ of H_{er} is identical to H_{er}, because H_{er} contains only committed transactions (that is: for any transaction $T_i \in H_{er}$ it is true that $H_{er}|T_i = H'_i \cdot [tryC_i \to C_i]$).

Note that there exists a sequential history $S = H_{er}|T_i \cdot H_{er}|T_j$ that is equivalent to H_{er}. Since all transactions are concurrent in H_{er}, the real time order of H_{er} is empty ($\prec_{er} = \varnothing$). Then, trivially, $\prec_{er} \subseteq \prec_S$. So S satisfies Def. 12a.

S contains operations of transactions $\mathbb{T} = \{T_i, T_j\}$ on objects $Var = \{x\}$. Subhistory $Vis(S, T_i) = S|T_i$ is legal since $Vis(S, T_i)|x = [w_i(x)1 \to ok_i] \in Seq(x)$ and the value i is in the domain of x ($\mathbb{N}_0$). Hence T_i in S is legal in S. Subhistory $Vis(S, T_j) = S|T_i \cdot S|T_j$ is legal as well, since $Vis(S, T_j)|x = [w_i(x)1 \to ok_i, r_j(x) \to 1]$, since i is in the domain of x and $r_j(x) \to 1$ is directly preceded by an operation writing i to x. Hence T_j in S is legal in S.

Since every T_i in S is legal in S, then sequential history S satisfies Def. 12b. Therefore, H_{er} is final-state opaque. □

Lemma 134. *H_{er} is not opaque.*

Proof. let history P be a prefix of H_{er} created by removing the last 4 events of H_{er}, i.e.:

$$P = \left[\ start_i \to ok_i, start_j \to ok_j, w_i(x)1 \to ok_i, r_j(x) \to 1\ \right].$$

Furthermore, let $P' = Compl(P)$ s.t., $P'|T_i = P|T_i \cdot [tryC_i \to A_i]$ and $P'|T_j = P|T_j \cdot [tryC_j \to A_j]$. Note that, from definition of completion, P' is the only possible completion of P because only case (d) applies to both transactions in P.

There are two possible sequential histories equivalent to P'. The first one is $S' = P'|T_i \cdot P'|T_j$. Since T_i is aborted in P', then $Vis(S', T_j) = P'|T_j$ (that is, operation executions from $P'|T_i$ are excluded from $Vis(S', T_j)$). However, $Vis(S', T_j)$ is not legal because it contains operation execution $r_j(x) \to 1$ that is not preceded by any write and $v_0 \neq i$. Hence T_i in S' is not legal in S'. So S' does not bear out Def. 12b.

The second sequential history equivalent to P' is $S'' = P'|T_j \cdot P'|T_i$. Here, $Vis(S'', T_j) = P'|T_j$ (because T_j is not preceded by any other transaction in S''). Since, $Vis(S'', T_j) = Vis(S', T_j)$, then by analogy to the discussion above $Vis(S'', T_j)$ is not legal, so T_j in S'' is not legal in S''. Thus, S'' does not satisfy Def. 12b either.

In effect, there is no sequential history equivalent to P' that satisfies Def. 12. Therefore, P does not satisfy Def. 12, and, since P is a prefix of H_{er}, then H_{er} does not satisfy Def. 13 and so it is not opaque. □

B
Algorithms

This chapter contains full pseudocode of the various secondary variants of new algorithms introduced and discussed in the main text. Where pertinent, we highlight the changes from the original algorithm. We provide the pseudocode in this way for the sake of completeness, and for the convenience of any potential implementers.

```
1  proc start(Transaction T_i) {
2    for ⌈x⌋ ∈ ASet_i in order {
3      R ← location(⌈x⌋)
4      if owner(lk(R)) ≠ T_i
5        lock lk(R) → W
6    }
7    for ⌈x⌋ ∈ ASet_i {
8      gv(⌈x⌋) ← gv(⌈x⌋) + 1
9      pv_i(⌈x⌋) ← gv(⌈x⌋)
10   }
11   for ⌈x⌋ ∈ ASet_i {
12     R ← location(⌈x⌋)
13     if owner(lk(R)) = T_i
14       unlock lk(R)
15   }
16 }
17 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
18   wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
19   execute m on ⌈x⌋ returning v
20   return v
21 }
22 proc commit(Transaction T_i) {
23   for ⌈x⌋ ∈ ASet_i {
24     wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
25     lv(⌈x⌋) ← pv_i(⌈x⌋)
26   }
27   return C_i
28 }
```

(a) BVA.

```
1  proc start(Transaction T_i) {
2    for ⌈x⌋ ∈ ASet_i in order {
3      R ← location(⌈x⌋)
4      if owner(lk(R)) ≠ T_i
5        lock lk(R) → W
6    }
7    for ⌈x⌋ ∈ ASet_i {
8      gv(⌈x⌋) ← gv(⌈x⌋) + 1
9      pv_i(⌈x⌋) ← gv(⌈x⌋)
10   }
11   for ⌈x⌋ ∈ ASet_i {
12     R ← location(⌈x⌋)
13     if owner(lk(R)) = T_i
14       unlock lk(R)
15   }
16 }
17 proc access(Transaction T_i, Object ⌈x⌋, Method m) {
18   wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
19   execute m on ⌈x⌋ returning v
20   ac_i(⌈x⌋) ← ac_i(⌈x⌋) + 1
21   if ac_i(⌈x⌋) = supr_i(⌈x⌋)
22     lv(⌈x⌋) ← pv_i(⌈x⌋)
23   return v
24 }
25 proc commit(Transaction T_i) {
26   for ⌈x⌋ ∈ ASet_i {
27     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
28     if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
29       lv(⌈x⌋) ← pv_i(⌈x⌋)
30     ltv(⌈x⌋) ← pv_i(⌈x⌋)
31   }
32   return C_i
33 }
```

(b) SVA.

Figure B.1: Versioning algorithms with CGL version acquisition.

```
1  proc start(Transaction T_i) {
2    // Acquire private versions.
3    for ⌈x⌋ ∈ ASet_i in order
4      lock lk(⌈x⌋) → W
5    for ⌈x⌋ ∈ ASet_i {
6      gv(⌈x⌋) ← gv(⌈x⌋) + 1
7      pv_i(⌈x⌋) ← gv(⌈x⌋)
8      unlock lk(⌈x⌋)
9    }
10   // Asynchronously buffer read−only variables.
11   for ⌈x⌋ ∈ ASet_i: wub_i(⌈x⌋) = 0
12     async run :read_buffer(T_i, ⌈x⌋)
13       when pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
14   return ok_i
15 }
16 proc read(Transaction T_i, Object ⌈x⌋, Method m) {
17   // Read−only object.
18   if ⌈x⌋ is read-only {
19     join with :read_buffer(T_i,⌈x⌋)
20     execute m on buf_i(⌈x⌋) returning v
21     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
22     return v
23   }
24   // Object not previously released.
25   if (wc_i(⌈x⌋) < wub_i(⌈x⌋)
26       or uc_i(⌈x⌋) < uub_i(⌈x⌋)) {
27     if wc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
28       wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
29       if (wc_i(⌈x⌋) > 0)
30         apply log_i(⌈x⌋) to ⌈x⌋
31     }
32     execute m on ⌈x⌋ returning v
33     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
34     if (rc_i(⌈x⌋) = rub_i(⌈x⌋)
35         and wc_i(⌈x⌋) = wub_i(⌈x⌋)
36         and uc_i(⌈x⌋) = uub_i(⌈x⌋))
37       :release(T_i,⌈x⌋)
38     return v
39   }
40   // Object previously released.
41   if (wc_i(⌈x⌋) = wub_i(⌈x⌋)
42       and uc_i(⌈x⌋) = uub_i(⌈x⌋)) {
43     if write_buffer(T_i,⌈x⌋) is running
44       join with :write_buffer(T_i,⌈x⌋)
45     execute m on buf_i(⌈x⌋) returning v
46     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
47     return v
48   }
49 }
50 proc update(Transaction T_i, Object ⌈x⌋, Method m) {
51   if rc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
52     wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
53     if wc_i(⌈x⌋) > 0
54       apply log_i(⌈x⌋) to ⌈x⌋
55   }
56   execute m on ⌈x⌋ returning v
57   uc_i(⌈x⌋) ← uc_i(⌈x⌋) + 1
58   if (wub_i(⌈x⌋) = wc_i(⌈x⌋)
59       and uub_i(⌈x⌋) = uc_i(⌈x⌋)) {
60     buf_i(⌈x⌋) ← ⌈x⌋
61     :release(T_i,⌈x⌋)
62   }
63   return v
64 }
65 proc write(Transaction T_i, Object ⌈x⌋, Method m) {
66   // No preceding reads or updates.
67   if rc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
68       execute m on log_i(⌈x⌋)
69     wc_i(⌈x⌋) ← wc_i(⌈x⌋) + 1
70     if wc_i(⌈x⌋) = wub_i(⌈x⌋)
71       async run :write_buffer(T_i,⌈x⌋)
72         when pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
73   }
74   // Some preceeding reads or updates.
75   if rc_i(⌈x⌋) > 0 or uc_i(⌈x⌋) > 0 {
76     execute m on ⌈x⌋
77     wc_i(⌈x⌋) ← wc_i(⌈x⌋) + 1
78     if wc_i(⌈x⌋) = wub_i(⌈x⌋) {
79       buf_i(⌈x⌋) ← ⌈x⌋
80       :release(T_i, ⌈x⌋)
81     }
82   }
83 }
84 proc commit(Transaction T_i) {
85   for(⌈x⌋ ∈ ASet_i) {
86     if wub_i(⌈x⌋) = 0
87       join with read_comit(T_i,⌈x⌋)
88     else {
89       if (wc_i(⌈x⌋) = wub_i(⌈x⌋)
90           and rc_i(⌈x⌋) = uc_i(⌈x⌋) = 0)
91         join with write_buffer(T_i,⌈x⌋)
92       else {
93         if wc_i(⌈x⌋) + rc_i(⌈x⌋) = uc_i(⌈x⌋) = 0
94           wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
95           apply log_i(⌈x⌋) to ⌈x⌋
96         }
97       }
98       wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
99       if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
100        lv(⌈x⌋) ← pv_i(⌈x⌋)
101    }
102    ltv(⌈x⌋) ← pv_i(⌈x⌋)
103  }
104  return ok_i
105 }
106 proc :read_buffer(Transaction T_i, Object ⌈x⌋) {
107   buf_i(⌈x⌋) ← ⌈x⌋
108   :release(T_i,⌈x⌋)
109   async run :read_commit(T_i,⌈x⌋)
110     when pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
111 }
112 proc :read_commit(Transaction T_i, Object ⌈x⌋) {
113   if ∃⌈y⌋: rv_i(⌈y⌋) > cv(⌈y⌋)
114     return abort(T_i)
115   ltv(⌈x⌋) ← pv_i(⌈x⌋)
116 }
117 proc :write_buffer(Transaction T_i, Object ⌈x⌋) {
118   apply log_i(⌈x⌋) to ⌈x⌋
119   buf_i(⌈x⌋) ← ⌈x⌋
120   :release(T_i,⌈x⌋)
121 }
122 proc :release(Transaction T_i, Object ⌈x⌋) {
123   lv(⌈x⌋) ← pv_i(⌈x⌋)
124 }
```

Figure B.2: OptSVA-CF (commit-only).

```
1  proc start(Transaction T_i) {
2    // Acquire private versions.
3    for ⌈x⌋ ∈ ASet_i in order
4      lock lk(⌈x⌋) → W
5    for ⌈x⌋ ∈ ASet_i {
6      gv(⌈x⌋) ← gv(⌈x⌋) + 1
7      pv_i(⌈x⌋) ← gv(⌈x⌋)
8      unlock lk(⌈x⌋)
9    }
10   // Asynchronously buffer read−only variables.
11   for ⌈x⌋ ∈ ASet_i: wub_i(⌈x⌋) = 0
12     if T_i ∈ ℝ
13       async run :read_buffer(T_i, ⌈x⌋)
14         when pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
15     else
16       async run :read_buffer(T_i, ⌈x⌋)
17         when pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
18   return ok_i
19 }
20 proc read(Transaction T_i, Object ⌈x⌋, Method m) {
21   // Read−only object.
22   if ⌈x⌋ is read-only {
23     join with :read_buffer(T_i,⌈x⌋)
24     if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
25       return abort(T_i)
26     execute m on buf_i(⌈x⌋) returning v
27     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
28     return v
29   }
30   // Object not previously released.
31   if (wc_i(⌈x⌋) < wub_i(⌈x⌋)
32       or uc_i(⌈x⌋) < uub_i(⌈x⌋)) {
33     if wc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
34       if T_i ∈ ℝ
35         wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
36       else
37         wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
38       :checkpoint(T_i,⌈x⌋)
39       if (wc_i(⌈x⌋) > 0)
40         apply log_i(⌈x⌋) to ⌈x⌋
41     }
42     if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
43       return abort(T_i)
44     execute m on ⌈x⌋ returning v
45     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
46     if (rc_i(⌈x⌋) = rub_i(⌈x⌋)
47         and wc_i(⌈x⌋) = wub_i(⌈x⌋)
48         and uc_i(⌈x⌋) = uub_i(⌈x⌋))
49       :release(T_i,⌈x⌋)
50     return v
51   }
52   // Object previously released.
53   if (wc_i(⌈x⌋) = wub_i(⌈x⌋)
54       and uc_i(⌈x⌋) = uub_i(⌈x⌋)) {
55     if write_buffer(T_i,⌈x⌋) is running
56       join with :write_buffer(T_i,⌈x⌋)
57     if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
58       return abort(T_i)
59     execute m on buf_i(⌈x⌋) returning v
60     rc_i(⌈x⌋) ← rc_i(⌈x⌋) + 1
61     return v
62   }
63 }
64 proc update(Transaction T_i, Object ⌈x⌋, Method m) {
65   if rc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
66     if T_i ∈ ℝ
67       wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
68     else
69       wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
70     :checkpoint(T_i,⌈x⌋)
71     if wc_i(⌈x⌋) > 0
72       apply log_i(⌈x⌋) to ⌈x⌋
73   }
74   if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
75     return abort(T_i)
76   execute m on ⌈x⌋ returning v
77   uc_i(⌈x⌋) ← uc_i(⌈x⌋) + 1
78   if (wub_i(⌈x⌋) = wc_i(⌈x⌋)
79       and uub_i(⌈x⌋) = uc_i(⌈x⌋)) {
80     buf_i(⌈x⌋) ← ⌈x⌋
81     :release(T_i,⌈x⌋)
82   }
83   return v
84 }
85 proc write(Transaction T_i, Object ⌈x⌋, Method m) {
86   // No preceding reads or updates.
87   if rc_i(⌈x⌋) = 0 and uc_i(⌈x⌋) = 0 {
88       execute m on log_i(⌈x⌋)
89     wc_i(⌈x⌋) ← wc_i(⌈x⌋) + 1
90     if wc_i(⌈x⌋) = wub_i(⌈x⌋)
91       if T_i ∈ ℝ
92         async run :write_buffer(T_i,⌈x⌋)
93           when pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
94       else
95         async run :write_buffer(T_i,⌈x⌋)
96           when pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
97   }
98   // Some preceeding reads or updates.
99   if rc_i(⌈x⌋) > 0 or uc_i(⌈x⌋) > 0 {
100    if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
101      return abort(T_i)
102    execute m on ⌈x⌋
103    wc_i(⌈x⌋) ← wc_i(⌈x⌋) + 1
104    if wc_i(⌈x⌋) = wub_i(⌈x⌋) {
105      buf_i(⌈x⌋) ← ⌈x⌋
106      :release(T_i, ⌈x⌋)
107    }
108  }
109 }
110 proc commit(Transaction T_i) {
111   for(⌈x⌋ ∈ ASet_i) {
112     if wub_i(⌈x⌋) = 0
113       join with read_comit(T_i,⌈x⌋)
114     else {
115       if (wc_i(⌈x⌋) = wub_i(⌈x⌋)
116           and rc_i(⌈x⌋) = uc_i(⌈x⌋) = 0)
117         join with write_buffer(T_i,⌈x⌋)
118       else {
119         if wc_i(⌈x⌋) + rc_i(⌈x⌋) = uc_i(⌈x⌋) = 0
120           wait until pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
121         if (wc_i(⌈x⌋) > 0
122             and rc_i(⌈x⌋) = uc_i(⌈x⌋) = 0) {
123           :checkpoint(T_i,⌈x⌋)
124           if ∃⌈y⌋: rv_i(⌈y⌋) ≠ cv(⌈y⌋)
125             return abort(T_i)
126           apply log_i(⌈x⌋) to ⌈x⌋
127         }
128       }
129       wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
130       if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
131         lv(⌈x⌋) ← pv_i(⌈x⌋)
132       if (rc_i(⌈x⌋) + wc_i(⌈x⌋) + uc_i(⌈x⌋) > 0
133           and rv_i(⌈x⌋) = cv(⌈x⌋)
134           and pv_i(⌈x⌋) - 1 > lv(⌈x⌋))
135         cv(⌈x⌋) ← pv_i(⌈x⌋)
136     }
137   }
138   if ∃⌈y⌋: rv_i(⌈y⌋) > cv(⌈y⌋)
139     return abort(T_i)
140   for ⌈x⌋ ∈ ASet_i
141     ltv(⌈x⌋) ← pv_i(⌈x⌋)
142   return ok_i
143 }
144 proc abort(Transaction T_i) {
145   for ⌈x⌋ ∈ ASet_i {
146     wait until pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
147     if (rc_i(⌈x⌋) + wc_i(⌈x⌋) + uc_i(⌈x⌋) > 0
148         and pv_i(⌈x⌋) - 1 > lv(⌈x⌋)
149         and rv_i(⌈x⌋) = cv(⌈x⌋)
150         and wub_i(⌈x⌋) + uub_i(⌈x⌋) > 0) {
151       if wc_i(⌈x⌋) = wub_i(⌈x⌋)
152         join with :write_buffer(T_i,⌈x⌋)
153       :recover(T_i,⌈x⌋)
154     }
155     if pv_i(⌈x⌋) - 1 = lv(⌈x⌋)
156       lv(⌈x⌋) ← pv_i(⌈x⌋)
157     ltv(⌈x⌋) ← pv_i(⌈x⌋)
158   }
159   return A_i
160 }
```

Figure B.3: ROptSVA-CF+R.

```
161 proc :read_buffer(Transaction T_i, Object ⌈x⌋) {
162   rv_i(⌈x⌋) ← cv(⌈x⌋)
163   buf_i(⌈x⌋) ← ⌈x⌋
164   :release(T_i, ⌈x⌋)
165   async run :read_commit(T_i, ⌈x⌋)
166     when pv_i(⌈x⌋) - 1 = ltv(⌈x⌋)
167 }
168 proc :read_commit(Transaction T_i, Object ⌈x⌋) {
169   if ∃⌈y⌋: rv_i(⌈y⌋) > cv(⌈y⌋)
170     return abort(T_i)
171   ltv(⌈x⌋) ← pv_i(⌈x⌋)
172 }
173 proc :write_buffer(Transaction T_i, Object ⌈x⌋) {
174   :checkpoint(T_i, ⌈x⌋)
175   apply log_i(⌈x⌋) to ⌈x⌋
176   buf_i(⌈x⌋) ← ⌈x⌋
177   :release(T_i, ⌈x⌋)
178 }
```

```
179 proc :checkpoint(Transaction T_i, Object ⌈x⌋) {
180   st_i(⌈x⌋) ← ⌈x⌋
181   rv_i(⌈x⌋) ← cv(⌈x⌋)
182 }
183 proc :recover(Transaction T_i, Object ⌈x⌋) {
184   ⌈x⌋ ← st_i(⌈x⌋)
185   cv(⌈x⌋) ← rv_i(⌈x⌋)
186 }
187 proc :release(Transaction T_i, Object ⌈x⌋) {
188   cv(⌈x⌋) ← pv_i(⌈x⌋)
189   lv(⌈x⌋) ← pv_i(⌈x⌋)
190 }
```

Figure B.3: ROptSVA-CF+R.

Institute of Computing Science
Faculty of Computing
Poznań University of Technology

Typeset using LaTeX.

BibTeX:

```
@phdthesis{Sie16,
  author = "Konrad Siek",
  title = "{Distributed Pessimistic Transactional Memory: Algorithms and Properties}",
  school = "Pozna{\'n} University of Technology",
  address = "Pozna{\'n}, Poland",
  year = "2016"
}
```

TeX source statistics:

```
files: 129,              characters: 1728610,
words: 128950,           dollar signs: 24544,
lines: 37268,            backslashes: 38575,
comment lines: 10655,    vspaces: 31,
empty lines: 3483,       macros: 577,
emphs: 641,              expletives: 2.
```