Abstract: The difficulty of writing efficient and correct concurrent code with the use of low-level primitives lead researchers to introduce more advanced abstractions like Transactional Memory (TM). Unfortunately, TM's optimistic approach to concurrency control causes some new problems, including, specifically, low performance in high contention, as well as the inability to efficiently handle irrevocable operations. Hence, research on pessimistic TM is ongoing that means to ameliorate these problems. This research includes our prior work on a family of algorithms called versioning algorithms. In this paper we propose a method of increasing performance of these algorithms in high contention environments by introducing a simple first-fit decreasing scheduler that sorts incoming transactions as they negotiate the order of accessing shared objects. We show experimentally, that such an augmentation provides a performance improvement that is dependent on the saturation of transactions in the system that does not incur additional overhead in low contention.
Dear Prof. Hollingsworth,

We would like to submit a manuscript entitled *Transactions Scheduled While You Wait: Augmenting Transactional Memory with a Sorting Queue* for consideration for publication in the Elsevier Journal of Parallel Computing.

The research described in the manuscript is part of our ongoing work on distributed transactional memory (TM). The work is mainly focused on creating efficient and reliable transactional concurrency control algorithms that meet the requirements of distributed systems and that can themselves be distributed. The work presented in the manuscript explores the possibility of improving the efficiency of one such algorithm by extending it with a queue-based transaction scheduler. This work was not previously published or submitted for publication, and, to our knowledge, constitutes a novel and promising application of scheduling in transactional memory research.

Our previous work that has bearing on or direct relation to the submitted manuscript is as follows:


Please consider the corresponding author for this manuscript to be Konrad Siek and feel at liberty to direct any communication to his contact address, e-mail address (*konrad.siek@cs.put.edu.pl*), or telephone number (+48 61 665 3060). If necessary, Paweł Wojciechowski can be contacted via the same contact address, his email address (*pawel.t.wojciechowski@cs.put.edu.pl*) or his telephone number (+48 61 665 3021).

Your Sincerely,
Konrad Siek and Paweł T. Wojciechowski
* We define accessing objects by a TM as a scheduling problem (with resources).
* We extend SVA with a queue-based first-fit decreasing scheduler.
* We show performance improvement experimentally using idealized workloads.
* We evaluate the scheduler experimentally using realistic TM benchmarks.
Transactions Scheduled While You Wait: Augmenting Transactional Memory with a Sorting Queue

Konrad Siek\textsuperscript{a,1}, Paweł T. Wojciechowski\textsuperscript{a,1}

\textsuperscript{a}Institute of Computing Science
Poznań University of Technology
Piotrowo 2, 60–965 Poznań, Poland

Abstract
The difficulty of writing efficient and correct concurrent code with the use of low-level primitives lead researchers to introduce more advanced abstractions like Transactional Memory (TM). Unfortunately, TM’s optimistic approach to concurrency control causes some new problems, including, specifically, low performance in high contention, as well as the inability to efficiently handle irrevocable operations. Hence, research on pessimistic TM is ongoing that means to ameliorate these problems. This research includes our prior work on a family of algorithms called versioning algorithms. In this paper we propose a method of increasing performance of these algorithms in high contention environments by introducing a simple first-fit decreasing scheduler that sorts incoming transactions as they negotiate the order of accessing shared objects. We show experimentally, that such an augmentation provides a performance improvement that is dependent on the saturation of transactions in the system that does not incur additional overhead in low contention.

Keywords:
Concurrency control, Software transactional memory, Distributed systems

\textsuperscript{1}Corresponding author

Email addresses: konrad.siek@cs.put.edu.pl (Konrad Siek), pawel.t.wojciechowski@cs.put.edu.pl (Paweł T. Wojciechowski)

URL: http://www.cs.put.poznan.pl/ksiek (Konrad Siek),
http://www.cs.put.poznan.pl/pawelw (Paweł T. Wojciechowski)

Preprint submitted to Parallel Computing

November 19, 2014
1. Introduction

Writing parallel programs using the low-level synchronization primitives is notoriously difficult and error-prone. Over the past decade, there has been a growing interest in alternatives to lock-based synchronization by turning to the idea of software transactional memory (TM)[1, 2]. TM hides the details of synchronization under the transaction abstraction: roughly, the programmer needs only to specify where transactions begin and end, and TM manages the execution so that the transactional code executes correctly (e.g., transactions are typically at least serializable [3] and give stronger consistency guarantees by satisfying properties like opacity [4], transactional memory specification [5], or virtual world consistency [6]) and, ideally, reasonably well in terms of efficiency. Thus, the programmer puts in less effort into writing safe concurrent code. Plenty of research was conducted on TM algorithms and many systems were developed for various programming languages (see e.g., [7, 8, 9, 10, 11, 12, 13] among others).

Furthermore, the goal of writing safe and efficient code inspired proposals of distributed transactional memory, since distributed programming poses some of the same problems as multiprocessor programming. (Although there are additional problems in distributed TM stemming from communication properties and the possibility of partial failures). Distributed TM systems are usually constructed in one of two ways. The first one (e.g., [14, 15, 16, 17, 18]) involves using locally-scoped transactions which execute as if on a single machine, which is in fact a network of replicated nodes. That is, as operations are performed on one node, it passes on the modifications to other replicas, so that, they effectively all have the same state. The second one (e.g., [19, 20, 21]) uses a system model where each network node is distinct and each is a host to a number of shared resources. A transaction in this model may need to communicate with several different network nodes to access all the resources it requires, thus we refer to them as distributed transactions. They can either simply read data from the objects or write data to them (this is the more traditional data flow model) or they can ask for code to be run on shared resources to do processing in addition to data acquisition (in the control flow model). In other words, the data flow model treats shared resources as variables, and the control flow model treats them as objects.

Most concurrency control algorithms used in TM, whether for multiprocessors or distributed systems, are optimistic. That is, transactions execute
code speculatively and may be forced to abort and restart if there is a conflict, i.e., several transactions attempt to access the same resource concurrently, such that at least one of them applies modification to the resource. However, there may be problems stemming from speculative execution. For instance, in high contention—if many transactions attempt to access the same resource—it is likely that some of them will be forced to abort and re-execute (at least partially) several times, until they manage to commit. 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, through serialization of execution of conflicting transactions, to a dispatcher avoiding collisions based on a probability of conflict [22].

Another issue with optimistic TM is the problem of irrevocable operations i.e., operations that cannot be aborted and should not be repeated, such as I/O operations, network communication, or acquiring locks. If these appear inside a transaction and the transaction is forced to re-execute, the operations 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). Fixing the problem in the optimistic approach leads to complicated or cumbersome solutions. For instance in [23] an irrevocable transaction must wait until all other transactions are cleared from the system, so a conflict is impossible.

A simpler method of dealing with both the problems described above is using a pessimistic concurrency control algorithm. Pessimistic TM does not execute speculatively, but delays transactional operations so that they do not conflict. This means, that forced aborts are much less common or even impossible, and therefore high contention or irrevocability do not cause abort-related problems. Thus, some pessimistic TM research has appeared recently for multiprocessor TM [24, 25].

In our prior research [26] we proposed a family of versioning algorithms. These are distributed pessimistic TM algorithms where transactions use version numbers for each object to delay accesses and in this way guarantee mutual exclusion. We briefly describe one of the algorithms, the Supremum Versioning Algorithm (SVA) [20]. It is implemented in a concurrency control library for Java called Atomic RMI [27] using the control flow model. The control flow model is one where each shared object is immobile and located on a discrete network node (a node can host any number of shared objects) and clients issue method invocations to these shared objects. The algorithms require some a priori knowledge about which objects will be accessed by a
given transaction and how many accesses are expected at most. This information is used to release any object early, even before the transaction commits, if at some point it is known the transaction no longer needs that object.

Since the algorithm knows so much about transactions at the outset, it is worthwhile to consider what else that information can be used for except concurrency control. Since pessimistic TM avoids conflicts, there is no need to use the information for conflict avoidance, e.g., as in CAR-STM [22]. However, if execution of transactions is thought of as a schedule, the \emph{a priori} information can be used to re-arrange them, so that in some circumstances the resulting execution is more efficient than a first-come-first-served approach. The problem can be approached using scheduling on parallel processors (e.g., [28, 29, 30, 31]), with particular emphasis on scheduling under resource constraints (e.g., [32, 33]). SVA, as implemented in Atomic RMI, allows for rearranging transactions whenever they acquire version numbers for shared objects. When several transactions ask for versions at the same time, Atomic RMI can establish which order of transactions would be the most efficient and return the version numbers following that order. Since the instant of time in which this rearrangement can be done is relatively brief, the algorithm used for scheduling must be quick, and therefore, we believe, necessarily approximate. Hence, we introduce a simple heuristic first-fit decreasing scheduler into our implementation.

The paper is structured as follows: In Section 2 we describe SVA, the algorithm into which the scheduler is incorporated. We then give a description of the scheduler, along with an in-depth discussion of the problem itself in Section 3. To our best knowledge we are the first to propose the use of a transaction scheduler to improve the performance of pessimistic transactions (i.e., without the need for transactions to conflict before scheduling). We then evaluate the implementation in Section 4, first using a toy benchmark with ideal conditions and then using a set of realistic distributed TM benchmarks for evaluation. Finally, we give our concluding thoughts in Section 6.

2. Supremum Versioning Algorithm

The \emph{Supremum Versioning Algorithm} (SVA) [20, 26] is a fully-pessimistic concurrency control algorithm used by Atomic RMI [27], a distributed TM implementation. SVA guarantees the last-use opacity [34] safety property and strong progressiveness [4] in terms of liveness. We summarize the \emph{modus
operandi of SVA below only to the extent to which it pertains to the current discussion of extending the algorithm with a scheduler (detailed in the next section). In particular we omit the details of aborting transactions. An interested reader is encouraged to seek further details in the original papers.

SVA, like all versioning algorithms [35], uses version counters to determine which transactions are allowed to access the object at any given time. There are three counters per shared object \( x \): global version counter \( V_g(x) \), local version counter \( V_l(x) \), and private version object \( V_p(x, T_i) \). Note that each transaction has its own private version counter, while global and local version counters are shared. The system is fully distributed: each counter is hosted by the same server as the corresponding shared object.

Whenever some transaction \( T_i \) initializes, for each shared object it will potentially access (read from or write to) during its execution, i.e., \( x \in RWSet(T_i) \), the transaction increments \( V_g(x) \). Subsequently, \( T_i \) reads the current value of \( V_g(x) \) and stores it in \( V_p(x, T_i) \). The initialization phase gives each transaction a corresponding “unique value” \( (V_p(x, T_i)) \) for each object that follows a specific order. Hence, \( V_p \) values can be used to decide when it is a given transaction’s turn to access \( x \). Note that each transaction must perform these operations atomically. After the initialization procedure finishes, the transaction begins executing any local code or accesses to shared objects and ends with either a commit or abort.

Whenever \( T_i \) commits, for every \( x \in RWSet(T_i) \) it sets the value of \( V_l(x) \) to the value of \( V_p(x, T_i) \). This indicates to other transactions that \( x \) is no longer used by the transaction with version \( V_p(x, T_i) \); hence \( x \) is released, and that another transaction can start accessing it.

Finally, when \( T_i \) attempts to access \( x \) (i.e., by writing to \( x \), reading from it, or executing its method), it waits until the following condition holds:

\[
V_p(x, T_i) - 1 = V_l(x).
\]

That is, \( T_i \) waits until some preceding transaction \( T_k \) releases \( x \). In general, this means that transactions access \( x \) in the order of initialization.

What makes SVA characteristic is that transaction \( T_i \) can release shared object \( x \) before committing. This is called early release. Transaction \( T_i \) can only release \( x \) early provided that it will not attempt to access \( x \) afterward. This is ensured by counting the accesses to each object as they happen (let the number of accesses to \( x \) by \( T_i \) be denoted \( C(x, T) \)). Then, given the upper bound on the number of times \( T_i \) can potentially access \( x \)—the supremum,
denoted $S(x, T) - T_i$ must simply check whether the following condition is true after each access to $x$:

$$C(x, T) = S(x, T).$$

Once this is true $T_i$ sets the value of $V_i(x)$ to the value of $V_p(x, T_i)$ to release $x$ early.

SVA requires that suprema are known \textit{a priori}, and that they are not lower than the actual number of accesses a transaction will perform on a particular object (if the supremum is not tight, it will result in loss of performance, but execution will remain correct). In practice the suprema can be derived statically with reasonable precision [36], or through an extended type systems [35]. When all else fails, they can be given by the programmer manually. The programmer also has the option to manually release a variable after last use, although this is less efficient than releasing objects through suprema (see [27]).

3. Transactions Scheduled While You Wait

As we noted in Section 2, SVA arranges all transactions that try to access a common subset of shared objects according to the order in which these transactions initialized. This order is enforced by the values of each private transaction version counters, which are assigned during initialization. In addition, SVA requires that all active transactions synchronize during initialization, as the private versions are atomically assigned. This means that if several transactions start at roughly the same time, only the first would actively initialize, while others would wait on a barrier. These circumstances provide a point in which the TM system could re-order transactions by assigning different private version value to the initializing transactions than would have been assigned normally. In this section we describe our proposition of using this opportunity to affect an improvement in performance. We do so by defining the problem in terms of scheduling and by proposing an implementation of a simple scheduler, which we use in the next section for an experimental evaluation.

3.1. Scheduling Problem Definition

In general terms, when a number of transactions are waiting to begin execution at a point in time, they can be described as the following deterministic scheduling problem (using a notation proposed in [33, 37] and
explained below):

\[ Pm \mid res11 \mid T_{max}. \]

This definition assumes that the set of transactions undergoing scheduling is the subset of transactions that are currently being initialized and wait at the appropriate barrier. Furthermore, even though a TM transaction can execute any local code before it tries to read or write a remote object, for the sake of clarity, we ignore such code for the purpose of this discussion. We do so since, by definition, it can be executed in parallel with any other code. As a result we assume that any transaction’s start is synonymous with its first access to a shared object.

The processor environment \((Pm)\) consists of a set \(P = \{P_1, P_2, \ldots, P_m\}\) of \(m\) identical machines, each representing the processing ability of one shared remote object.

There is also a set \(T = \{T_1, T_2, \ldots, T_n\}\) of transactions (usually referred to as \textit{tasks} in scheduling theory) awaiting execution, available at time 0. Furthermore, since transactions have to ensure the properties of \textit{isolation} and \textit{atomicity} their non-preemption is a requirement. Further still, transactions define neither precedence constraints (they are \textit{independent}) nor \textit{deadlines}. (These properties are implicitly assumed in the notation, and therefore not explicitly described).

Shared objects will be represented by resources, where each shared object instance is a different type of resource. Then, \(res11\) expresses that the system contains \(s\) types of additional renewable resources \(R_1, R_2, \ldots, R_s\), each representing access to a remote object. There is only one copy of each instance of a shared object, so one unit of each resource is available in the system. Depending on whether a shared object is in a transaction’s access set \(RWSet\), a transaction requires 0 or 1 unit of the corresponding resource, and it requires a unit of at least one type of resource to begin execution. The resource requirement set for each transaction \(R(T_j) = \{R_1(T_j), R_2(T_j), \ldots, R_s(T_j)\}\) is derived from its preamble (defined below): a non-zero declaration of access to a resource \((R_i(T_j) > 0)\) is a requirement for transaction \(T_j\) to obtain one unit of that resource.

Each transaction provides a specification defining the maximum number of times (the supremum) each remote object will be accessed by the transaction: a \textit{preamble} \(P(T_j) = \{S(R_1, T_j), S(R_2, T_j), \ldots, S(R_s, T_j)\}\). Thus, the preamble defines resource requirements. Since the purpose of the transaction is to handle operations on shared objects we further assume that the
time to execute code that does not involve shared objects is uniform and proportional to the number of shared object accesses and again omit it in our discussion for the sake of clarity. Hence, under this assumption, the sum of the suprema for all of a transaction access set determines the processing time of the transaction. In effect, the processing time $p_j$ associated with each transaction $T_j$ can be estimated as equal to the number accesses to remote objects it makes and derive it from the transaction’s preamble, and thus:

$$p_j = \sum_{i=1}^{s} S(R_i, T_j).$$

(This is also implicit in the notation of the problem definition.)

The optimality criterion ($T_{max}$) for the problem is to minimize schedule length and thus achieve greater throughput defined as the number of completed transactions per second.

3.2. Impact of Operation Order

The problem definition in Section 3.1 does not take into account the order of operations within transactions. This is because this order is unknown at the outset. It can also dynamically change during the execution as the state of shared objects changes. In effect, there is no way to model this within the problem definition. However, since SVA allows for early release, the order of operations has an effect on the length of the schedule, as we explain below.

First, let us suppose that we have transactions $T_1$, $T_2$, $T_3$, and $T_4$ appearing in the system respectively at times $\tau_1$, $\tau_2$, $\tau_3$, and $\tau_4$, in a consecutive manner so that $\tau_i < \tau_j$ iff $i < j$. There are two shared objects, $x$ and $y$, which we can treat as resources, such that each transaction requires a non-negative number of units of each resource which they will request at some (unknown) point during their execution and use them exclusively for some (again, unknown) amount of time. Let the resource requirements be as follows:

$$
\begin{bmatrix}
T_1 & T_2 & T_3 & T_4 \\
x & 1 & 1 & 1 & 0 \\
y & 1 & 0 & 1 & 1
\end{bmatrix}.
$$

For the sake of simplicity let us assume that the execution time required to access a resource and do the necessary processing connected with it is equal to exactly two units. Therefore, transactions $T_1$, $T_2$, $T_3$, and $T_4$ have the following execution times (respectively): $4$, $2$, $4$, $2$. 

8
Let us consider the worst-case scenario of executing these transactions presented in Fig. 1. Since $T_1$ is the only transaction in the system when it arrives, it is allowed to execute any accesses without waiting. Given the order of accesses in each transaction, the remainder of the transactions all must wait until the preceding transaction releases all of its shared objects (i.e., commits) before they can proceed to execute their own accesses. In effect, the total execution time $\tau_{\text{max}}$ of these transactions is 12 units.

![Figure 1: Transactions executed in order of arrival (worst case).](image)

However, the efficiency of a system can be improved by re-arranging the order in which transactions are allowed to execute while they wait to be allowed access to the resources they require. Scheduling transactions to start in a new order may lead to the system achieving a higher transaction throughput.

If we take the example above and consider the fact that $T_2$ and $T_4$ can be executed simultaneously, having disjoint resource requirements, transaction $T_4$ can be scheduled to execute at time $\tau_5$ along with $T_2$. That is, the order of transactions accessing $y$ stemming from their values of $V_p$ for $y$ will be $\sigma^y = [T_1, T_4, T_2]$ (as opposed to the original $\sigma^y_{\perp} = [T_1, T_2, T_4]$). Such a case is shown in Fig. 2, where it is apparent that such a rearrangement results in the total execution time $\tau_{\text{max}}$ of these transactions equal to 10 units, which improves upon the execution in presented in Fig. 1. Hence rescheduling transactions yields a significant performance benefit.

On the other hand, since the order in which transactions access shared objects is initially unknown, re-ordering transactions can yield negative performance with respect to the default execution. For instance, taking the example from Fig. 1, where transactions are scheduled in order of arrival, if $T_1$ executes its access to $x$ before $y$, then $T_1$ can release $x$ early at time $\tau_3$. Consequently, transaction $T_2$ can begin at $\tau_3$ rather than $\tau_5$. This, in turn, allows $T_3$ to begin at $\tau_5$ rather than $\tau_6$, when $T_1$ and $T_2$ finally release both
Figure 2: Transactions executed in custom order (performance gain).

resources. If $T_3$ then also reverses the order in which it accesses resources, then it releases resource $y$ at $\tau_6$ and transaction $\tau_4$ can begin at that time. When both $T_3$ and $T_4$ finish, all four transactions could be completed within 8 time units.

Figure 3: Transactions executed in order of arrival (best case).

However, let us assume the same ordering of shared object accesses as in Fig. 3, but let us apply the schedule chosen for Fig. 2. This example is shown in Fig. 4. In this scenario $T_1$ releases resource $x$ early at time $\tau_2$. Then, transaction $T_2$ can start executing at $\tau_3$. However, transaction $T_4$ still cannot start execution before $T_1$ commits at $\tau_5$, nor can $T_3$ start executing before $T_4$ finishes executing at $\tau_6$. Thus, all transactions finish at time $\tau_{\text{max}}$ equal to the initial prediction in Fig. 2, but higher than the $\tau_{\text{max}}$ that would have been achieved by an execution of transactions without re-ordering.

These examples show that the lack of knowledge about the order in which transaction will access shared objects coupled with the effects of the early release mechanism will cause any scheduler to be only approximate.

3.3. First Fit Decreasing Scheduler Implementation

Given the point of synchronization during the initialization of each transaction, and the a priori information gathered for the purposes of controlling
concurrent execution via SVA we propose to install a simple scheduler that solves the problem described with the arrival of each additional transaction to the system.

Note however, that the initialization phase for each transaction consists of $|S|$ increment operations of $V_g$ and as many writes to $V_p$. These operations can be considered quick (especially in relation to the execution time required for shared object accesses). Hence, the execution time of the re-ordering procedure must be just as low, in order to prevent incurring overhead on the system. More so, due this small time of initialization, the scheduler will only be effective when large numbers of transactions try to initialize simultaneously. Thus, the scheduler is expected to have its main use in high contention environments, where additional overhead should be avoided.

Our main goal, as stated in the problem definition, is to maximize throughput—the number of transaction executions per second. This is calculated as the total number of transactions divided by schedule length. Schedule length (denoted $\tau_{\text{max}}$) is the time it takes the last transaction in the system to finish executing. We start measuring schedule length as the first transaction starts and finish when all transactions have committed. We will also follow with a brief discussion of the impact of the scheduler on the average time between a transaction’s arrival in the system and the time it finishes execution. This is mean flow time (denoted $\Sigma C_j$).

The scheduler itself will operate as a first-fit decreasing scheduler, meaning it will attempt to allow those transactions to execute first, which can meet all their resource requirements and are the shortest. Such algorithms constitute approximate solutions to various scheduling problems with resource constraints [33, 38].

In the implementation, we introduce a queue that sorts transactions by length as they are enqueued at the barrier during initialization. We define a
transaction’s length as the sum of its maximum remote method invocation counts. This definition is apt for control-flow distributed TM in practice (in particular, for Java RMI-based implementations), where remote method calls are among the most costly operations within transactions. We also test a variant with increasing sorting rather than decreasing, which allows shorter transactions to initialize first. Both policies have interesting advantages, which we show further below.

We present the simplified pseudocode of our scheduler in Fig. 5. Procedures lock and unlock replace those used to guard the critical section in the initialization procedure in SVA (Atomic RMI). Procedure lock defines one parameter—an ordered pair \((T, s)\) whose first element is a transaction’s identifier, and the second is its length. Procedure unlock has no parameters. Both procedures have access to queue \(tq\) which contains transaction identifier-length pairs and global lock \(l\) which prevents multiple transactions from initializing simultaneously.

When procedure lock is called and queue \(tq\) is empty, transaction \(T\) tries to acquire lock \(l\) and proceeds to initialize if successful. But if there are other transactions already waiting in \(tq\) or one is holding lock \(l\) then \(T\) is enqueued in \(tq\) and the queue is sorted by transaction lengths (increasingly or decreasingly, depending on policy). The transaction then begins to wait. When it is notified it acquires \(l\) and begins initializing. When procedure unlock is called, the first transaction is removed from \(tq\) and notified, and lock \(l\) is released. The transaction calling unlock may then start executing (subject to the versioning algorithm).
4. Experimental Evaluation

We experimentally evaluated the simple scheduler presented in Section 3.3 using an implementation of Atomic RMI extended with the implementation of the scheduling queue. First, we perform a proof-of-concept (PoC) evaluation using simplistic toy data in a high contention environment. Then, we evaluate the extended TM system using a suite of benchmarks from HyFlow\cite{19} that were designed specifically to evaluate control-flow and data-flow distributed TM systems. These benchmarks allow us to perform more comprehensive tests using realistic applications, as well as a more realistic contention setting.

The tests were performed on a 9-node high performance cluster connected by a dedicated 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). We also ran tests using other runtimes, but the results did not show significant differences (see [27]).

The TM frameworks we use in our realistic evaluation are the standard Atomic RMI, Atomic RMI with a decreasing sorting queue extension (denoted Atomic RMI Incr. Sort), and Atomic RMI with an increasing sorting queue extension (Atomic RMI Decr. Sort). We evaluate Atomic RMI with specified tight (least) suprema. We also use Atomic RMI without specified suprema (denoted Atomic RMI No Suprema) as an anchor. Such a use of Atomic RMI means that all objects are acquired at the start of the transaction and released on commit, so while transactions that have disjoint access sets may execute in parallel, any transactions which use some subset of the same objects will execute in series. Since our discussion here only concerns the performance improvement of Atomic RMI, we do not compare our solution with other TM systems. A comparison between Atomic RMI and other distributed concurrency control methods was done in [27] and can be used to extrapolate a similar comparison for the scheduler-extended version of Atomic RMI.

4.1. Proof of Concept Benchmark

Using a toy example we aim to show that Atomic RMI’s implementation of SVA with a sort-queue–based scheduler enables better concurrency, and thus improves efficiency in comparison to the original SVA in ideal conditions.
We suggest that a simple low-cost scheduler to Atomic RMI will improve efficiency even more in specific cases and produce low-enough overhead to be negligible in all other cases.

Our testing environment consists of 8 or 24 clients uniformly distributed across machines simultaneously performing one transaction each (we use the remaining node as the server for the resources). Transactions invoke methods of remote objects (resources) located on a single server. There are four resources \( x, y, z, v \) each providing a remote method that takes 1 second to execute. A short transaction accesses objects \( x, y \) and \( z \) and commits. A long transaction accesses \( z, v, z, v, x, y \) and commits. Half the clients execute short transactions, and half execute long ones. The transactions initialize simultaneously, as if they just spawned and were waiting for another transaction (some hypothetical \( T_0 \)) to release resources.

This setup favors separating long and short transactions since they mutually block each other’s execution—a long transaction has to wait until a short transaction finishes, and a short transaction can only start as a long transaction starts to perform its last remote object access to \( y \). On the other hand a second long transaction can start when the first long transaction releases \( z \) in the mid-point of the transaction’s execution, and a second short transaction can start after the first performs its remote call to \( x \). Note that we use such a simple microbenchmark to specifically show the scheduler having an impact on performance. This simplicity also allows us to analyze all possible interweavings and interpret the results we obtain. We use a more complex benchmark for full evaluation in Section 4.2.

4.1.1. Example

The benchmark scenario for 4 clients is modeled in the example in Fig. 6. The bottom axis represents the flow of time in time units approximately 1 second each (with \( \Sigma C_j \) and \( \tau_{\text{max}} \) indicated on it). Each transaction arrives in the system at some point (indicated by a black dot), waits if necessary (a dashed line), initializes (a hatched box), and calls remote objects as it executes (white area with named resources). The initialization time in the figure is exaggerated.

In the example, some transaction \( T_0 \) arrives in the system at time 0 and begins initialization. During its initialization transactions \( T_1, T_2, T_3 \), and \( T_4 \) arrive, which are the subject of the scheduling. \( T_2 \) and \( T_4 \) are long transactions (6 remote object accesses) and \( T_1 \) and \( T_3 \) are short (3 accesses).

In Fig. 6a transactions initialize in order of arrival. This example delib-
erately presents the worst-case scenario, where long and short transactions arrive alternately and force each other to wait idly for resources to be released: $T_2$ and $T_3$ have to wait for $T_1$ and $T_3$ finish, and $T_3$ has to wait until $T_2$ releases resource $x$. This causes schedule length for this interweaving to be an estimated 19 time units and mean flow time $\Sigma C_j$ to be 10.2 units.

In Fig. 6b the scheduler reorders initialization so that longer transaction start first which means that more remote accesses can be performed concurrently overall and so $\tau_{\text{max}}$ is reduced to 14 units. The value of $\Sigma C_j$ changes little and remains at 9.8 units. In in Fig. 6c the reordering allows shorter transactions to initialize first which leads to smaller $\Sigma C_j$ of 8.2 units, while
\( \tau_{\text{max}} \) is 15 units.

4.1.2. Results

An iteration of the evaluation consists of each client simultaneously running a single transaction—the time it takes for the transaction to finish is measured as flow time \( (C_j) \). Then clients wait for all transactions to finish—this time is the schedule length \( (\tau_{\text{max}}) \). We run 13 such iterations per test, but we discount the measurements from the first 3 iterations as warm-up time. We then run 10 such tests complete with restarting the JVMs and the RMI registry. Thus we obtain 100 measurements of mean flow time and schedule length per client.

The results of our test are presented in Fig. 7. The rows contain results...
for 8, and 24 clients respectively. Each row contains a diagram showing mean flow time ($\sum C_j$) and throughput (number of clients by schedule length, $n/\tau_{\text{max}}$). Each diagram shows the type of concurrency control mechanism on the x-axis; a mean flow time diagram shows time in seconds on the y-axis and a throughput diagram, the number of transactions finished per second. We present the results in the form of boxplot diagrams. Boxplots present each set of results as a box with a line in the middle and two whiskers. The top and bottom of the box represent upper and lower quartiles and stand for the 75th and 25th percentile. The central line is the median of the result set. The top and bottom whiskers end at 1.5 IQR of the upper and lower quartiles respectively (IQR is the difference between the upper and lower quartiles). The points above and below the whiskers are outliers. The height between the ends of both whiskers shows the spread of results. The height of the box shows where most results fall.

Our measurements of mean flow time and throughput for 8 transactions show a very slight difference between the variants of Atomic RMI. The reason for this is that transactions arrive too sparsely which prevents the scheduler from activating often (or even not at all). If the scheduler does activate, the reordering it imposes involves only two or three transactions, which does not provoke a particularly large change in flow time. It can be seen that mean flow time of transactions using Atomic RMI No Suprema depends greatly on the order in which they arrive and therefore varies from instance to instance. Furthermore, the No Suprema variant performs worse on average than the other Atomic RMI test programs, because any No Suprema transactions that contend for shared objects are executed serially, while other Atomic RMI variants allow partial parallelization through early release. Predictably Atomic RMI without upper bounds performs the worst by a large margin in terms of transaction throughput at about 0.18 transactions per second—this will continue to be true no matter the number of clients. The measurements of throughput indicate a small but significant increase in efficiency of Atomic RMI with a scheduler over a version RMI without it, with increasing order sorting performing slightly better than decreasing order.

The results for 24 transactions decidedly show that the increasing scheduler greatly reduces mean flow time of transactions over that of Atomic RMI, since the scheduler has a high chance of reordering a large set of transactions. Increasing scheduling performs best in this regard, since short transactions do not wait for long ones. The throughput of transactions significantly increases when scheduling is enabled with more clients. The difference between sched-
uled and ordinary Atomic RMI becomes more pronounced, and increasing sorting performs slightly better still than decreasing. All varieties of Atomic RMI variant perform significantly better than the No Suprema variant in all aspects. The results indicate that the differences between various sorts of Atomic RMI become increasingly pronounced as the number of clients (thus contention) increases.

In addition, we tested the difference between the overhead pertinent to transaction initialization when using Atomic RMI with and without a scheduler using the same benchmark. The time of transaction initialization in Atomic RMI without a scheduler was 10.42 milliseconds with a standard deviation of 4.43. In Atomic RMI with scheduling initialization time was 11.03 with a standard deviation of 7.72. Even though the differences between the two are statistically significant they are on the order of 1 millisecond and therefore inconsequential to Atomic RMI’s overall performance. Thus, in low-contention the scheduler does not visibly degrade performance, even if it does not improve it. So, even if nothing is gained in term of performance from using a simple scheduler, it does not bear any cost.

However, it is important to stress that the improved efficiency is a result of applying this simple scheduler to a specific type of traffic. In many cases scheduling will offer no improvement, and it is conceivable that some schedules will be less efficient than if transactions initialized in order of arrival. We retain this type of scheduler in our efforts nevertheless, because the static analyzer in its current form provides all the required information and the simplicity of the scheduler means its overhead is virtually nil.

4.2. Distributed TM Benchmark Suite

While evaluation in the previous section showed that a simple scheduler improves efficiency in some scenarios while maintaining a low overhead, in this section we proceed to evaluate the efficiency improvement of this solution in a more realistic environment.

4.2.1. Benchmarks

We used two complex benchmarks for our realistic evaluation: Loan, and Vacation. We based our implementation of the benchmarks on the one included in HyFlow [19].

The Loan benchmark presents a complex distributed application where the execution of the transaction is itself 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.

The Vacation benchmark is a complex benchmark (originally a part of STAMP [39]), 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. This transaction may require programmatic use of rollback. 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.

4.2.2. Results

The results are presented in Fig. 8. Each benchmark is presented on two graphs: we conducted each test using 20% and 80% read/write operation ratio. Points on the graph represent the mean throughput (on the y-axis) from the given benchmark run on a particular number of nodes (on the x-axis). The legend in the top-right corner of each graph describes the symbols used for each concurrency control mechanism. The vertical line intersecting each point shows the variance of a given result. The dashed lines joining the points show trends for mechanisms as the number of nodes grows.

The results from the Loan benchmark present a situation where Atomic RMI concurrency control performs very similarly regardless of the scheduler, with Atomic RMI No Suprema visibly trailing. Because read and write transactions do not differ in Loan, the number of remote objects they use or the required computational power, there is neither a gain in performance from any scheduler (transactions are indistinguishable by length) nor is there
Figure 8: Evaluation results by benchmark (Loan and Vacation) for two workloads: 20% read transactions and 80% read transactions.

degraded performance with more of either type of transaction. Note, however, that the scheduler does not cause a statistically significant decrease in performance when it is not being used to any effect. Hence, no additional overhead is introduced by the scheduler.

The measurements of throughput from the Vacation benchmark show more interesting results in the 20% read scenario. First of all, since there are 4 different types of transactions, and the contention in the benchmark is fairly high at all times, it is possible for the scheduler to select between the transactions and actually rearrange them. Both the increasing and decreasing rearrangement will attempt to execute long transactions together, which
allows SVA to use its early release capability to a better extent, and therefore, to execute transactions with a higher degree of parallelism. Thus, the scheduler provides an edge to Atomic RMI making Atomic RMI Incr. Sort and Atomic RMI Decr. Sort better than their non-scheduled counterpart, with a statistically significant throughput gain of up to about 20%. However the difference disappears, when 80% of the transactions are of the same type: there is only one type of read-only transactions which constitutes the 80% read-only transactions while there are three types of write transactions that together constitute the remaining 20%.

Hence, given a high contention environment and an appropriately heterogeneous workload, the scheduler can provide a performance gain. On the other hand, when its use does not yield a benefit, it introduces little to no visible overhead.

5. Related Work

Our transaction scheduling bears resemblance to contention management in optimistic transactional memory systems. However, in contention management a different approach is taken and transactions are not reordered or reallocated as they arrive but only when conflicts are detected [40, 41]. Contention management can also be treated as non-clairvoyant job scheduling [42]. The marriage of scheduling and TM has been explored with respect to collision avoidance and adaptive transactional scheduling (ATS). In CARSTM [22] transactions are allocated to processor core queues by a transaction dispatcher on the basis of an estimated probability of conflicting with currently running transactions. In [43] the authors describe an ATS scheme for TM systems where in conditions of high contention transactions form a queue managed by a centralized scheduler which dispatches them sequentially. This work in particular stresses much of the same points as our scheduling—acting prior to transaction execution, low cost of operation when contention is low, performance gains when it is high—although our scheduler does not receive feedback from the core system. In [44], the authors propose a system where scheduling is used as the sole means of concurrency control. Like our system, they use static analysis to derive read-write sets for transactions, although it is not the authors’ focus and the analysis they use currently is very rudimentary. The read-write sets are modified dynamically during run-time to better represent actual resource requirements of transactions. The scheduler then dispatches transactions that do not conflict on their basis.
6. Conclusions

In this paper we discussed the possibility of including a first-fit decreasing scheduler to enhance the performance of pessimistic distributed TM systems based on versioning algorithms. We described the problem, as well as the scheduler, and implemented it as part of Atomic RMI to undergo an experimental evaluation.

The results of the evaluation show that a queue-based scheduler can achieve a throughput gain when added to Atomic RMI, provided that it operates in a high contention environment with a heterogeneous set of transactions. The evaluation also showed that, while there is no gain in an environment where these conditions are not met, the scheduler does not introduce a detrimental overhead to the TM. Hence, this simple addition to a TM system can be used universally and provide an edge whenever high contention occurs.

The work presented in this paper opens up an avenue to experiment further with scheduling transactions during transaction initialization in distributed TM, but many problems remain to be resolved. Most importantly, a more refined definition of the problem needs to be found that incorporates the uncertainty over operation ordering within transactions. Then, more sophisticated, heuristics-based algorithms can be researched that use data collected from transaction processing. Such algorithms could find schedules closer to the optimum in the relatively short time allowed for by the initialization phase.

Acknowledgments. The project was funded from National Science Centre funds granted by decision No. DEC-2012/06/M/ST6/00463. Early work on Atomic RMI was funded by the Polish Ministry of Science and Higher Education grant no. POIG.01.03.01-00-008/08.


