Statically Computing Upper Bounds on Object Calls for Pessimistic Concurrency Control

Konrad Siek
Poznań University of Technology
Konrad.Siek@cs.put.poznan.pl

Paweł T. Wojciechowski
Poznań University of Technology
Pawel.T.Wojciechowski@cs.put.poznan.pl

Abstract

Atomic RMI is a library for pessimistic concurrency control in distributed systems that uses *versioning algorithms*. These algorithms require *a priori* knowledge about execution of transactions to achieve maximum efficiency, i.e. the maximum number of times specific objects will be accessed. This paper presents an algorithm and a tool that establishes these upper bounds on objects accessed through static program analysis.

1. Introduction

The unpredictable manner in which concurrent data access is performed in shared memory systems causes concurrency control mechanisms continuously to gain importance, especially with the mounting popularity of multi-core processors. Therefore it is usual for programming languages to supply a set of constructs that facilitate the synchronization of concurrent processes, among them critical sections, monitors, locks, conditional variables, and semaphores. Nonetheless, these mechanisms are considered to be difficult to use and the developer must manage the blocking mechanisms unaided and on a low-level, as well as identify the sections of code that require synchronization—both being error-prone tasks. In addition, the low-level constructs are neither directly re-usable nor composable (if every single component is safe by itself, it does not necessarily follow that their combination is safe).

Many authors have recently proposed to solve the problems of concurrent programming by turning to the idea of Software Transactional Memory (STM) [27]: STM allows the declaration of *atomic transactions* in concurrent programming (similar to transactions known from Database Management Systems) whose correct processing is enforced by the system, thus ensuring synchronization. Several such systems have been designed and implemented, though mostly non-distributed (see e.g. [10–13, 24, 26] among others). Another line of research is on type systems for specifying and verifying the atomicity of methods in multithreaded programs (see [6] and follow-up papers). Roughly, it is guaranteed that the effect of an execution (i.e. the state of the system) of two or more concurrent atomic blocks of code will be equivalent to the effect of a sequential execution of those blocks.

In our work, we are interested in extending the idea of STM to distributed systems. In distributed systems, when two or more clients want to access shared remote resources consistently, it is necessary to control concurrent access to these resources. Since the distributed system can be loosely coupled and shared resources can be whatsoever, we exclude heavy-weight solutions based on distributed database systems. Several authors proposed *Distributed STM*, meaning a system with locally-scoped transactions executing on different network nodes, with shared data replicated on these nodes and kept consistent (see e.g. [3, 34] among others). In general, Distributed STMs can be seen as distributed shared memory systems (implementing distributed global address space), extended with the notion of atomic transactions. Alternative approaches include *distributed transactions* that can span many nodes; they are aimed at loosely-coupled distributed systems, such as the Internet. We consider STMs and distributed transactions that are *object-based*, in the sense that transactional synchronization works by intercepting and synchronizing calls to object methods, leveraging the infrastructure provided by object-oriented languages. An example of an object-based STM is DSTM2 [13], while JSTM [34] can be seen as an object-based Distributed STM.

We develop Atomic RMI [19, 32]—an object-based concurrency control library built on top of Java RMI which ensures linearizable distributed objects in Java by allowing the developer to define specific sequences of method calls on remote objects as distributed transactions. The library uses pessimistic concurrency control algorithms that suspend (block) methods calls when necessary to satisfy linearizability. We are currently extending our algorithms and implementation to support a rollback mechanism. Thus, the programmer will be offered a mechanism equivalent to distributed atomic transactions, in which shared data can be any objects defined as 'remote' (in the sense of Java RMI) and accessed by any of their methods. We have already designed and implemented a non-distributed variant of our library that supports rollback [17].

The result of the use of Atomic RMI, is lifting the chore of manual management of blockades from the developer, therefore reducing his efforts to declaring distributed transactions. However, the benefits in ease of use are offset by the requirements of the algorithms used by Atomic RMI. In order to schedule the execution of method calls, the algorithms require some input data about a transaction to be known *a priori*—i.e. before the transaction is spawned. More precisely, the developer needs to find the number of times a particular object will be accessed during run-time, or at least an upper bound on that number.

This paper proposes a method to gather the upper bounds on the number of calls to specific objects by static analysis in order to aid the programmer in using Atomic RMI transactions. This method was implemented in the form of a tool that is meant to be an integral part of Atomic RMI. However, our algorithm could also be useful in Distributed STMs and non-distributed STMs that use pessimistic concurrency control.

[Copyright notice will appear here once 'preprint' option is removed.]

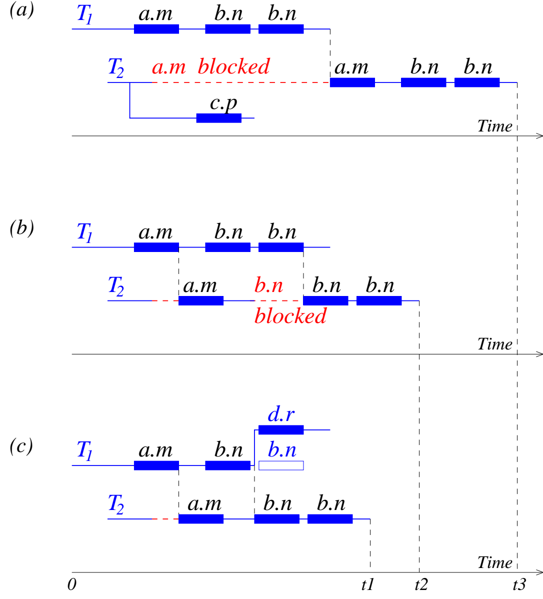

Figure 1. Transaction scheduling: (a) BVA, (b) SVA, and (c) RVA

There is a large body of research related to worst case analysis of programs that aims at deriving information about execution patterns statically (we sketch some of these in Section 5). However, we do not know examples of using this information for optimizing the execution of distributed transactions nor STM (or Distributed STM). Our paper shows how the information gathered from transactional code at compile time can make the execution of distributed transactions in Atomic RMI more efficient. However, we think that our approach is general enough to be applicable in the implementations of STM too.

The paper has the following structure. First, the concepts of pessimistic concurrency control in Atomic RMI are described. Then, the algorithm and tool for static analysis are shown. Next, similar work on worst case analysis of programs is described. Finally, our conclusions are presented.

2. Concurrency Control in Atomic RMI

With respect to concurrency control, Atomic RMI uses *versioning algorithms* [30, 31, 33]. The key idea is that an object method called by a transaction is executed if that transaction holds a matching *version number* of the object. Otherwise, the call is blocked (put to sleep), waiting passively for the version upgrade. Thus, version numbers determine the order of executing object methods by transactions; this order agrees with the isolation property. This simple mechanism protects objects from being accessed by transactions that—in order to satisfy the isolation property—should wait till other transactions access these objects.

Figure 1(a) shows transaction scheduling using the *BVA* versioning algorithm. Note that a call to critical method $a.m$ of transaction T_2 is postponed until transaction T_1 has completed. This gives an almost serial execution of transactions. However, some other critical method p of another (non-shared) object c can be executed by transaction T_2 in parallel with transaction T_1; note that $c.p$ is executed by a separate thread that belongs to transaction T_2 (e.g. as a result of an asynchronous call to the remote object c). The isolation property is still satisfied since the effects of executing $c.p$ cannot be observed by transaction T_1.

Figure 1(b) shows transaction execution using the *SVA* algorithm. The algorithm must know *a priori*, a *least-upper-bound* (or *supremum*) on the number of times a given object may be accessed by a transaction. In our example, method m of object a is called by transactions T_1 and T_2 only once, while method n of object b is called twice. This quantitative information allows the algorithm to permit more parallelism than *BVA*. For example, consider transactions T_1 and T_2. Note that the critical method $a.m$ of transaction T_2 (that had been postponed) is now executed soon after transaction T_1 has completed executing $a.m$. This is permitted since the *SVA* algorithm already knows that the supremum on the number of times object a (providing method m) can be called by transaction T_1 is equal 1, i.e. the method $a.m$ will not be executed again by k_1, and so k_2 can call a and execute m. If supremum cannot be reached for some object, e.g. because it has been over-calculated then the *SVA* algorithm performs like *BVA*. The latter guarantees that all blocked objects are released.

The *RVA* improves upon the *SVA* by exploring the partial information about transaction execution. Figure 1(c) shows an example execution of two transactions, T_1 and T_2, under control of the *RVA* algorithm. For each transaction, the algorithm must know *a priori*, a causal relation between object calls made by the transaction.

As the example indicates, in relation to *BVA*, the *RVA* and *SVA* can be expected to shorten the system's response time (the time before a transaction terminates) and increase its capacity for simultaneously running transactions. We are mostly concerned with *SVA* at this time, although extending our efforts to encompass *RVA* may prove to be an interesting topic for future work.

3. Finding Upper Bounds on Method Calls

We designed an algorithm, dubbed the *Object Call Count Analysis (OCCA)*, to count upper bounds on the number of calls issued to each of a predefined set of objects. The algorithm statically analyzes source code in the Simple Language (SL), a small subset of the Java language. SL consists of only the most basic constructs, like variable declaration, assignment, `if` statements, `while` loops, etc. (the complete definition of the language has been described in TR [28]). [1] If the upper bound cannot be computed, then the algorithm returns infinity, which reduces efficient scheduling of transaction operations to serial execution of transactions. Thus, our algorithm guarantees safety but not completeness. The following properties of the algorithm are proposed.

PROPOSITION 1 (Supremum). *The OCCA algorithm computes the* upper bounds (supremum) *on objects' method calls for any correct program in the SL language. All method invocations in code (that are not dead code) are considered, and the estimated upper bound is never lower than the actual number of method invocations.*

PROPOSITION 2 (Safety). *The OCCA algorithm is safe: it cannot produce any upper bound on the number of object accesses that is lower than the actual number of this objects' method calls during any execution of the analyzed program.*

PROPOSITION 3 (Termination). *The OCCA finishes in a finite number of steps, even on occurrence of recursion or infinite loops.*

The algorithm's input takes the form of source code in SL expressed as an expression tree. The algorithm's output is a map where keys are unique object identifiers pointing upper bounds on the number of times the objects were used to make a method call.

[1] Our implementation supports a subset of Java that is a superset of SL, and therefore is more fit for use.

This number can be zero, any finite number, or ω—the symbol representing an infinite or unpredictable result.

The algorithm is started by the application of function **analyze_expr**, which takes the expression tree (E) and the number of executions of this fragment of code, as arguments. The function then decides what particular type of expression it was provided with, and then delegates the analysis to one of the specialized functions. The function is given in Figure 2. The type of the main function and the specialized functions is defined as:

$$analyze_expr : expression \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{id \rightarrow \mathbb{N}_0 \cup \omega\} \quad (1)$$

where, id belongs to the set of all possible identifiers.

The most interesting aspects of the analysis are handling of loops, conditional expressions, and method calls, and these are explained below. Detailed information can be found in TR [28].

3.1 Method call analysis

When a method call has been found in the code, the function indicates that the object implementing the analyzed method will be called at most a certain number of times. The function returns a single-element map: if the analyzed expression is evaluated only once, then the object identifier will be mapped to the value of one; if the expression may be evaluated more times (e.g. in a loop) then the number of evaluations is included in the map. Later, the number from all such maps will be totaled to produce a global estimate.

If the body of the analyzed method is known, then the method's arguments and its body are further analyzed. Additionally, every time a method's body is analyzed, a global visit counter for that method is incremented, and if the counter exceeds a certain threshold, it must be assumed that infinite recursion occurred, and no more analysis of that method's body should is allowed.

In cases where the method body is not analyzed, the states of values that may be potentially modified within the body should be set to unknown. If the body is of the method known, the variables that need to be subjected to this may be derived by analysis.

The function can be expressed formally as,

$$\begin{array}{l} analyze_call(E, n) = \\ \quad \begin{cases} \{object(variable(E)) \mapsto n\}, & \text{if } \neg has_body(E) \\ add(\{object(v) \mapsto n\}, & \text{if } V < M \\ \quad analyze_block(P \cup B, n)) & \quad \wedge has_body(E) \\ \{object(variable(E)) \mapsto \omega\}, & \text{if } V \geq M \end{cases} \\ \text{where} \\ \quad v = variable(E), B = method_body(E), \\ \quad P = method_argument_assignments(E), \\ \quad V = visited(E), M = MAX_ITERATIONS \end{array} \quad (3)$$

3.2 Conditional block analysis

The function attempts to evaluate the guard condition whose outcome may be either *true*, *false*, or *uncertain*. In the foremost case the positive block is then further analyzed, while in the case of evaluation to *false* the positive block is ignored, and if the alternative block is analyzed if it is defined. If the guard condition is not a literal, but another expression, it must also be analyzed.

In the case of *unpredictable*, both blocks must be analyzed further and the maximum of both analyses must be derived using **join**. In addition, if there are assigned values that differ between blocks, the variable must be noted to have an unknown value.

The **join** operation $\bigvee$ takes two maps and returns a new map, which contains all key-value pairs from both maps. If a key is present in both input maps, the referenced values are compared, and the higher of the two is used in the output map. The **join** operator ensures that the maximum number of method calls will be selected as a result of the analysis.

The conditional analysis function is defined as follows,

$$\begin{array}{l} analyze_conditional_block(E, n) = \\ \quad \begin{cases} analysis_proper(E, C, n) & \text{if } is_literal(C) \\ add(analyze_expr(C, n), & \\ \quad analysis_proper(E, C, n)) & \text{if } \neg is_literal(C) \end{cases} \\ analysis_proper(E, C, n) = \\ \quad \begin{cases} analyze_expr(E_{positive}, n) & \text{if } is_true(C) \\ analyze_expr(E_{alternative}, n) & \text{if } is_false(C) \\ analyze_expr(E_{positive}, n) & \text{if } \neg is_true(C) \\ \bigvee analyze_expr(E_{alternative}, n) & \wedge \neg is_false(C) \end{cases} \\ \text{where} \\ \quad C = condition(E), \\ \quad E_{positive} = positive_block(E), \\ \quad E_{alternative} = alternative_block(E) \end{array} \quad (4)$$

3.3 Loop block analysis

The function attempts to analyze an expression E representing a loop. The loop's condition is evaluated: if the result is *true* both the condition C and the body B of the loop are further analyzed before the next round of loop analysis is conducted; whereas if the result is *false* only the condition is analyzed and the analysis of the loop ends. If the condition evaluates to neither *true* nor *false* but is indefinite, the condition and the body are both analyzed using unknown values.

Since it is possible for the analysis never to terminate if the condition always evaluates to *true*, the parameter i tracks the number of iterations, and when it exceeds a threshold the analysis assumes the loop will run indefinitely and treats the loop as if the condition evaluated to an unknown value. If the condition of the loop is unknown, an analysis is performed, which marks all of the values that may be written to as unknown.

$$\begin{array}{l} analyze_loop_block(E, n) = \\ \quad analyze_loop(E, n, MAX_ITERATIONS) \\ analyze_loop(E, n, i) = \\ \quad \begin{cases} add(analyze_expr(C, n), & \\ \quad analyze_expr(B, n), & \\ \quad analyze_loop(E, n, i-1)) & \text{if } is_true(C) \wedge i > 0 \\ analyze_expr(C, n) & \text{if } is_false(C) \wedge i > 0 \\ add(analyze_expr(C, \omega), & \text{if } \neg(is_true(C) \\ \quad analyze_expr(B, \omega)) & \vee\, is_false(C)) \vee i \leq 0 \end{cases} \\ \text{where} \\ \quad C = condition(E), B = body(E) \end{array} \quad (5)$$

This approach is resource-intensive, so a more light-weight alternative is considered: an auxiliary function **iteration_count** predicts the maximum number of loop iterations beforehand, then the condition and the body of the loop are analyzed, with the iteration count as an argument.

$$\begin{array}{l} analyze_loop_block(E, n) = \\ \quad \begin{cases} add(analyze_expr(C, n * m), \\ \quad analyze_expr(B, n * m)) \end{cases} \\ \text{where} \\ \quad C = condition(E), B = body(E), \\ \quad m = iteration_count(E) \end{array} \quad (6)$$

A simple and imprecise function establishing the maximum number of loop iterations can determine if the loop will be evaluated at all, in which case it returns 0, and otherwise returns the unknown value ω. The conception of better functions is widely covered in [2, 4, 7, 8].

$$
analyze_expr(E,n) = \begin{cases} analyze_declaration(child(E),n) & \text{if } type(E) = \text{variable_declaration} \\ analyze_initialization(child(E),n) & \text{if } type(E) = \text{object_initialization} \\ analyze_assignment(child(E),n) & \text{if } type(E) = \text{variable_assignment} \\ analyze_call(child(E),n) & \text{if } type(E) = \text{method_call} \\ analyze_block(child(E),n) & \text{if } type(E) = \text{block} \\ analyze_conditional_block(child(E),n) & \text{if } type(E) = \text{conditional_block} \\ analyze_loop_block(child(E),n) & \text{if } type(E) = \text{loop_block} \end{cases} \quad (2)
$$

Figure 2. The main function of the OCCA algorithm

4. Code Generation

A prototype static analyzer was built based on the algorithm described above. Apart from implementing the various analyses, the tool generates preambles to transactions that it finds in the code, with the upper bounds defined using the Atomic RMI API.

The code generation itself is done by injecting tokens into the processed code on the lexical level. Before it is done, the appropriate places in the token stream must be found, and this is done during the analysis of the expression tree.

5. Related work

A large body of work is available on the use of static analysis in the Worst-Case Execution Time (WCET) problem [16, 29], where methods like path analysis [9], symbolic analysis [18], and partial evaluation [14] are employed to establish upper bounds on the time code will take to run. Emphasis is placed on finding in advance the maximum number of loop iterations [2, 4, 7, 8]. Frameworks for WCET are available, like aiT [5] or Bound-T [15].

This work also bears resemblance to finding user-definable usage bounds (for Java [20, 21, 23] in particular, but also other languages [22]) where prediction of the maximum number of uses of particular resources is attempted by code analysis.

In the context of code generation, a number of tools employ static analysis to deduce properties of the code that the following generation requires, e.g. AutoBinder [25] a Python binding generator for the C language, or Jass [1] (Java with Assertions), a tool for injecting assertions based on defined invariants.

6. Conclusions

The investigation of the subject has led to the conception of the algorithm, as well as the implementation of a prototype tool for static analysis that is able to process a subset of the Java language (extended in relation to SL) and generate safe, execution-ready code. The tool, therefore, relieves the developer of having to create these preambles manually.

Both the tool and the algorithm itself are designed with extension in mind, to allow future expansion beyond the limited capabilities of the SL, as well as to refine the analyses to produce more precise bounds. Specifically, future work could involve improving the method for handling loops.

In addition, support for *RVA* in Atomic RMI may become a topic for future work, resulting in still improved response times.

References

[1] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[2] Mikael Sjödin Christopher A. Healy and David Whalley. Bounding loop iterations for timing analysis. In *Proceedings of RTAS '98: the 4th IEEE Real-Time Technology and Applications Symposium*, pages 12–21, June 1998.

[3] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proceedings of PRDC '09: the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 307–313, November 2009.

[4] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proceedings of WCET '07: the 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2007.

[5] Christian Ferdinand and Reinhold Heckmann. AiT: Worst case execution time prediction by static program analysis. In *Proceedings of WCC '04: the 18th IFIP World Computer Congress*, pages 377–384, August 2004.

[6] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI '03: Conference on Programming Language Design and Implementation*, June 2003.

[7] Sumit Gulwani. Speed: Symbolic complexity bound analysis. In *Proceedings of CAV '09: the 21st International Conference on Computer Aided Verification*, volume 5643 of *LNCS*, pages 51–62, June 2009.

[8] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of PLDI '09: Conference on Programming Language Design and Implementation*, pages 375–385, June 2009.

[9] Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond 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*, pages 47–57, April 2008.

[10] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of PPoPP '05: the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[11] Timothy Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA '03: the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.

[12] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC '03: the 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[13] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of OOPSLA '06: the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–262, October 2006.

[14] Niklas Holsti. Analysing switch-case tables by partial evaluation. In *Proceedings of WCET '07: the 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2007.

[15] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proceedings of EUSIPCO 2000: the 10th European Signal Processing Conference*, September 2000.

[16] Raimund Kirner and Peter Puschner. Obstacles in Worst-Case Execution Time Analysis. In *Proceedings of ISORC '08: the 11th IEEE*

International Symposium on Object-Oriented Real-Time Distributed Computing, pages 333–339, May 2008.

[17] Piotr Kryger and Paweł T. Wojciechowski. Atomic Locks: Projekt i implementacja mechanizmu pamięci transakcyjnej w języku Java. Technical Report TR-ITSOA-OB2-2-PR-09-6, Institute of Computing Science, Poznań University of Technology, December 2009.

[18] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, 1999.

[19] Mariusz Mamoński and Paweł T. Wojciechowski. Atomic RMI: Projekt i implementacja mechanizmu rozproszonych operacji atomowych w języku Java. Technical Report TR-ITSOA-OB2-2-PR-09-4, Institute of Computing Science, Poznań University of Technology, December 2009.

[20] Mario Méndez-Lojo and Manuel V. Hermenegildo. Precise set sharing analysis for Java-style programs. In *Proceedings of VMCAI '08: the 9th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 4905 of *LNCS*, pages 172–187, January 2008.

[21] Mario Méndez-Lojo, Jorge Navas, and Manuel V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *Proceedings of LOPSTR '07: the 17th Symposium on Logic-based Program Synthesis and Transformation*, volume 4915 of *LNCS*, pages 154–168, August 2007.

[22] Jorge Navas, Edison Mera, Pedro López-García, and Manuel V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *Proceedings of ICLP '07: the 23rd International Conference on Logic Programming*, volume 4670 of *LNCS*, pages 348–363, September 2007.

[23] Jorge Navas, Mario Méndez-Lojo, and Manuel V. Hermenegildo. User-definable resource usage bounds analysis for Java bytecode. *Electronic Notes in Theoretical Computer Science*, 253(5):65–82, 2009.

[24] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings of OOPSLA '08: the 23rd ACM SIGPLAN Conference on Object-oriented Programming, Systems Languages and Applications*, October 2008.

[25] Tristan Ravitch, Steve Jackson, Eric Aderhold, and Ben Liblit. Automatic generation of library bindings using static analysis. In *Proceedings of PLDI '09: Conference on Programming Language Design and Implementation*, pages 352–362, June 2009.

[26] Michael F. Ringenburg and Dan Grossman. AtomCaml: first-class atomicity via rollback. In *Proceedings of ICFP '05: the 10th ACM SIGPLAN International Conference on Functional Programming*, September 2005.

[27] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of PODCS '95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1995.

[28] Konrad Siek and Paweł T. Wojciechowski. Design and implementation of the precompilation tool in Java for Atomic RMI. Technical Report TR-ITSOA-OB2-2-PR-09-5, Institute of Computing Science, Poznań University of Technology, December 2009.

[29] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem — overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[30] Paweł 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.

[31] Paweł T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. Wydawnictwo Politechniki Poznańskiej, 1st edition, 2007.

[32] Paweł T. Wojciechowski. Extending atomic tasks to distributed atomic tasks. In *Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly* $(EC)^2$ *(co-located with CAV '08)*, July 2008.

[33] Paweł T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, April 2004.

[34] XSTM. `http://www.xstm.net`, 2010.