Deconstructing Redundant Memory Synchronization

Christoph von Praun
IBM T.J. Watson Research Center
Yorktown Heights

Abstract

In multiprocessor systems with weakly consistent shared memory, memory fence (also know as barrier) instructions are necessary to establish memory consistency at synchronization points in a parallel program. Programs that follow an acquire-release synchronization protocol (e.g., Java) make frequent use of such fence instructions and hence the thrifty use and efficient implementation of such instructions is an important performance aspect. Perhaps surprisingly, this paper demonstrates on the example of multi-threaded Java benchmarks that most fences occurring in a program execution are - ex post - unnecessary according to the rules of memory consistency demanded by the software (max. 99.9%, avg. > 98%). We conclude that the model for memory synchronization offered by current multiprocessor systems is not well aligned with the requirements of the widely used acquire-release software synchronization protocol.

1. Introduction

Modern multiprocessor systems provide a weakly consistent view of memory to individual processors. This means that different processors may observe the shared memory in different states at the same time. The weak memory model results from mechanisms inside individual processors that serve to optimize the memory access path (caches) and aggressively reorder memory accesses [8].

1.1 Acquire-Release synchronization protocol

The *acquire-release* protocol [6] defines the rules of memory synchronization that threads must observe if their control flow crosses a specific *synchronization point*, i.e., an access to a variable or construct that is designated for synchronization purpose (e.g., a lock). The control flows of synchronizing threads conceptually "meet" at such synchronization points. To be general, we refer in this presentation to *synchronization variables* and *synchronization point*, not to locks and lock access.

Definition 1: Acquire-release protocol

Acquire and release operations are associated with accesses to synchronization variables. All acquire and release operations that occur for the same synchronization variable are totally ordered. The acquire-release protocol imposes the following obligations on threads that access the *same* synchronization variable:

- A *release* operation guarantees that updates to shared memory that the current processor has issued prior to the release can be visible to other processors.

- An *acquire* operation guarantees that the current processor can correctly observe updates of shared memory that other processors have made available through a release operation (preceding the acquire).

These memory consistency guarantees do *not* necessarily hold for acquire and release that occur on *different* synchronization variables. □

Acquire and release are conceptual operations. The implementation at the hardware-level depends on the shared memory model that is supported by the machine architecture. Some architectures associate memory synchronization semantics with atomic operations, e.g., [9], while other architectures separate the aspects of memory synchronization and atomic operations, e.g., [18]. The discussion and observations made by this paper are independent of a specific processor architecture. Hence throughout this paper the term *synchronization operation* is used to refer to the conceptual operations acquire or release. The term *memory fence* shall refer to a specific implementation.

At the programming level, acquire and release operations are implied by higher-level synchronization primitives. Several thread packages implement this model of memory synchronization, and recently also the Java programming language adopted these memory semantics for its `synchronized` and `volatile` language constructs [14].

1.2 Example: PowerPC

Figure 1 illustrates the implementation of a lock with acquire-release semantics on the PowerPC architecture [18].

The load-reserve (`lwarx`) and store-conditional (`stwcx`) instructions are executed in a loop to achieve an atomic read and update of the lock variable. Once a thread succeeds to store its id `<tid>` in the lock variable, it won the race for the lock. This code is simplified and does not contain provisions for back-off and queued waiting. The `isync` instruction ensures that preceding instructions complete and discards the results of instructions that follow (in program order) that may have speculatively begun execution in the out-of-order (OOO) execution pipeline. In particular, all read memory accesses that precede `isync` will perform be-

```
 1   // lock
 2   while (true) {
 3       tmp = lwarx(lock);
 4       if (!tmp && stwcx(<tid>, &lock))
 5           break;
 6       isync;  // memory acquire operation
 7   }
 8
 9   <critical region>
10
11   // unlock
12   sync;          // memory release operation
13   lock = 0;
```

Figure 1: Lock implementation with acquire-release memory semantics on the PowerPC.

fore the read accesses that follow `isync`. The `sync` instruction is similar to `isync` instruction but more comprehensive. In addition to the local sequencing of instructions that preceded and respectively follow it, `sync` ensures that the underlying memory subsystem performs loads and stores due to instructions that preceded `sync`, before loads and stores that are due to instruction that follow `sync` (in program order). While the guarantees of `sync` are actually stronger than those demanded by the acquire-release protocol[1], we do not elaborate the discussion in this paper and refer the reader for a comprehensive specification of memory fences to [8, 18]. The precise definitions of the terms 'complete' and 'has performed' are given in in [6].

Table 1 reports the approximate number of cycles taken to execute memory fence and atomic exchange instructions. The numbers have been determined with a single-threaded microbenchmark on a 8-way Power4 1.1 GHz system and a 4-way Power5 1.6GHz system. `sync`, `isync`, and `lwsync` are variants of memory synchronization instructions; the combination of the `lwarx` and `stwcx` read and update a variable atomically. Note that the cycle time of these instructions is not constant and depends on the dynamic context (like the number of pending stores); the cycle times reported in the table demonstrate nevertheless that cost of memory fence instructions is substantial and can be several times to cost of a memory access.

instruction	Power4	Power5
sync	~ 140	~ 50
lwsync	~ 110	~ 25
isync	~ 30	~ 30
lwarx/stwcx	~ 80	~ 75

Table 1: Approximate cycle times for memory synchronization and atomic-read-update instructions on a Power4 and Power5 multiprocessor server.

1.3 Claims and contribution
This work studies the execution of multi-threaded programs and classifies the occurrence of acquire and release operations according to their role in establishing a consistent memory view among software threads. The classification of memory synchronization operations in the execution trace of a program is done ex post, i.e., after all synchronization events during a program execution are known. The classification is solely based on the synchronization structure in the execution trace and independent of the processor architecture and events that control memory or cache coherence at the hardware level.

Based on this classification, it is argued that on average more than 98% (max 99.9%) of the dynamic memory synchronization operations are unnecessary, i.e., they could have been omitted without compromising the acquire-release memory synchronization protocol.

2. Sources of redundancy
This section describes different models for the classification of memory synchronization operations in an execution and defines the condition in each model under which a synchronization operation is "unnecessary". An empirical assessment of each model and its capability to assess the redundancy of synchronization operations is given in Section 3.

2.1 Thread-confined synchronization variables
A variable is said to be *thread-confined* if it is accessed only by a single thread. In this study thread-confinement is determined dynamically, based on all accesses to a specific synchronization variable. If *all* accesses are done by the same thread, then the variable and the accesses are thread-confined. In specific cases, a static program analysis can determine thread-confinement, e.g., [1, 3, 23, 17]. Such a static analysis is however necessarily conservative and hence would report fewer cases of thread-confinement.

Claim.
Memory synchronization (acquire and release) that occurs with access to a thread-confined synchronization variable is redundant.

Proof.
First, it is assumed that threads are *self-consistent*, i.e., a thread does not have to apply acquire and release operations to make its updates to memory visible to itself. As threads are an abstraction of a processor provided by the operating system (OS), we assume that the OS transparently applies memory synchronization if necessary (e.g., the thread is scheduled to a different processor). The memory synchronization at the OS-level is not part of this study.

According to Definition 1, all threads that aim for a consistent memory view must access the *same* synchronization variable. An access to a thread-confined synchronization variable is hence only obliged to provide memory consistency within the accessing thread. This corresponds to the self-consistency that is implied by the thread abstraction of the OS. Hence explicit acquire or release synchronization operations that occur along with the access to thread-confined synchronization variables are not necessary.   □

2.2 Thread and processor locality of synchronization
Synchronization locality is a situation where a synchronization variable is accessed in an immediate sequence by the same thread (*thread locality*) or processor (*processor locality*). The case of thread-confined synchronization variables that is discussed in the previous section is a special case of

---

[1]On the PowerPC, a light-weight variant of the `sync` instruction, i.e., `lwsync`, is sufficient to achieve the semantics of release.
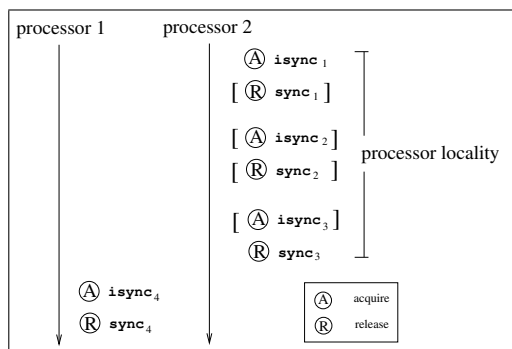
Figure 2: Locality of synchronization on the example of a lock that is subsequently acquired and released on the same processor. All accesses target the same synchronization variable.

thread locality, where the locality of access to a synchronization variable extends over the whole lifetime of the variable.

Example

A scenario of processor locality of synchronization is illustrated in Figure 2. The example is based on the PowerPC implementation of a lock given in Figure 1. Processor 2 executes an immediate sequence of lock and unlock operations on the same lock variable. The instructions $\mathtt{isync_2}$ and $\mathtt{isync_3}$ are *unnecessary*, because reads following $\mathtt{isync_1}$ can expect that data is already consistent on processor 2. The $\mathtt{sync}$ instructions on processor 2 are *proactive*, such that $\mathtt{sync_1}$ and $\mathtt{sync_2}$ turn out to be unnecessary in retrospect. In the figure, unnecessary memory synchronization is enclosed in square brackets.

Claim.

In a thread or processor local sequence of release operations, all but the last release operation is redundant. The dual holds for all but the first acquire operation.

Proof.

The proof is similar to the one given for thread-confined synchronization variables. For the period of locality, synchronization operations (acquire resp. release) can be regarded to occur on a synchronization variable that is confined to a specific environment (thread or processor). This environment is self-consistent and hence explicit memory synchronization is redundant. An exception are the first acquire operation and the last release operation that ensure memory consistency (wrt. to the synchronization variable) at the entry and exit of the access sequence. □

Thread and processor locality can be combined: In the combined model, a synchronization operation is considered as redundant if it is redundant due to thread *or* processor locality.

2.3 Eager releases and repetitive acquires

Definition 1 states that acquire and release operations that occur with different synchronization variables are independent, i.e., a release operation performed for a synchronization variable s_1 does not provide a consistent visibility of memory to an acquire operation on a synchronization variable s_2 ($s_1 \neq s_2$).

In current processor architectures such as the PowerPC, the implementation of acquire and release operations is oblivious to the synchronization variable with which these operations are associated, i.e., memory fence instructions establish consistency for the overall memory. Hence the guarantees of the implementation are actually stronger than what is required by in Definition 1.

This section defines a stronger notion of the acquire and release operation that corresponds more closely to the implementation of memory fences in current processor architectures. Then, this section explores if – given the stronger semantics of acquire and release – additional memory operations can be regarded as redundant.

Definition 2: Strong acquire-release protocol

Acquire and release operations are associated with accesses to synchronization variables. There is a partial order among acquire and release operations that is defined as follows: A release operation *precedes* an acquire operation if (1) both operations occur in the same thread and are respectively ordered in the control-flow, or (2) acquire and release occur in different threads and there are accesses to the *same* synchronization variable in the local control-flow of the threads following the release and preceding the acquire operation.

The strong acquire-release protocol imposes the following obligations on threads accessing *any* shared synchronization variable:

- A *release* operation guarantees that updates to shared memory that the current processor has issued prior to the release can be visible to other processors.

- An *acquire* operation guarantees that the current processor can correctly observe updates of shared memory that other processors have made available through a release operation (preceding the acquire).

The effect of strong acquire and release operations is independent of the synchronization variable with which they occur. □

Example

The following example illustrates the difference between normal acquire/release operations (Definition 1) and the strong variant (Definition 2). With acquire/release semantics according to Definition 1, none of the synchronization operations in Figure 3 would be redundant.

Given the strong semantics (Definition 2), some of the release operations occur over-*eagerly* (e.g, $\mathtt{sync_1}$, $\mathtt{sync_3}$) because there is a timely subsequent release with the same effect (e.g, $\mathtt{sync_2}$, $\mathtt{sync_4}$). Some acquire operations are *repetitive* (e.g, $\mathtt{isync_2}$, $\mathtt{isync_4}$), because there is some earlier acquire that already established consistency (e.g, $\mathtt{isync_1}$, $\mathtt{isync_3}$). In Figure 3, unnecessary memory synchronization is enclosed in square brackets.
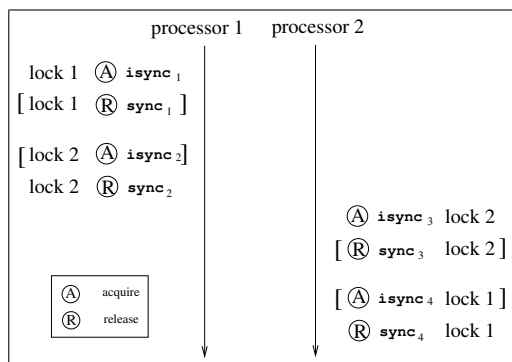
Figure 3: Eager releases and repetitive acquires in the strong acquire release model.

Note that these observations apply only to the specific execution, i.e., synchronization interleaving, that is illustrated in the figure.

Claim.
This claim assumes strong acquire and release semantics (Definition 2). Definition 2 implies that there is a partial order among memory synchronization operations that is compatible with the partial order of accesses to synchronization variables.

A release operation is *eager*, if there is an immediate successor (in the partial order) release operation that is executed on the same processor. An acquire operation is *repetitive* if there is an acquire operation issued by the same thread that is immediately preceding in the partial order.

Eager release operations and *repetitive* acquire operations are redundant.

Proof.
Follows from Definition 2.   □

3. Evaluation

The evaluation is based on a commercial Java VM with an extension that traces the occurrence of lock operations on the fly, i.e, during program execution. The experiments are done on a 4-way Power5 1.6 GHz server with symmetric multi-threading (SMT), i.e., there are 2 hardware threads per processor (overall 8 hardware threads in the multiprocessor system). For the study of processor locality, each hardware thread is treated as self-consistent logical processor.

The tracing extension slows down the program execution; the slowdown is relative to the frequency of lock operations between 5% and 1700% (average 75%) of the total execution time. There are two sources for the slowdown: First, certain optimization in the code generation are disabled; this affects in particular the efficiency of the lock enter and exit code. Second, the instrumentation and corresponding library calls add code to the execution that is not found in the original program. While the tracing extension is designed to allow multi-threaded execution, it might affect the parallelism and skew the execution of a parallel application. We

believe, however, that the potential skew would not unduly affect or support the claims made in this paper (redundancy in memory synchronization): Due to the overhead of the tracing extension, phases of locality could be shortened and hence opportunities for redundancy might even disappear.

The following set set of multi-threaded scientific and server benchmark program are investigated:

- mtrt is a multi-threaded raytracer from the JVM98 benchmark suite [19] configured with two threads.

- mold(yn), ray(tracer), and monte(carlo) are multi-threaded numeric application kernels from the Java Grande benchmark suite [10]. All benchmarks are executed with two threads in the 'size B' configuration.

- hedc is a warehouse for scientific data developed at ETH Zürich [21]. This benchmark represents an application kernel that implements a meta crawler for searching multiple Internet archives in parallel. In the benchmark configuration, four driver threads issue random queries to two archives each. The individual queries are handled by reusable worker threads.

- jigsaw is a http-server implementation [22] (version 2.2.4). The workload in the experiments is created by two http-clients that fetch a total of 2500 random web-pages of 1024 bytes.

- pseudojbb and trade6 emulate the transaction processing and WWW front-end of typical e-commerce applications. pseudojbb is a variant of SPECjbb [20] processes a fixed number of transactions (500,000, no ramp-up) instead of a fixed amount of time (original version). Configurations with 4 and 16 warehouses represent one scenario with fewer resp. more software threads than hardware threads (in the multiprocessor system that is used for this study). trade6 is configured with a 120 sec. ramp-up period followed by a 240 sec. stress test. All components of the benchmark, including the database, executed on the same machine (2-tier configuration).

Table 2 reports the rate at which the benchmarks execute memory synchronization operations. The numbers are approximate (adjusted by the slowdown due to the VM extension) and refer to the sum of operations in the 4-way multiprocessor system, i.e., the rate per processor is actually lower. Each pair of acquire and release is counted as one operation. The VM implements an aggressive compiler-based lock elimination that is enabled for all experiments (this optimization reduces the frequency of memory synchronization). We report only memory synchronization that is associated with non-reentrant lock operations – not, e.g., synchronization that occurs with accesses to volatile variables.

For monte, pseudojbb, and trade6, the frequency of lock events is relatively high, such that synchronization operations contribute noticeably to the overall execution time; we validated this for the example of specjbb, where the omission of memory synchronization in a run with one warehouse

benchmark	thousand ops / sec	CPU util
mtrt	120	170
mol	≪ 1	380
monte	1650	290
ray	≪ 1	350
hedc	30	100
jigsaw	45	130
pseudojbb (4 wh)	1530	280
pseudojbb (16 wh)	2320	240
trade6	630	290

Table 2: Frequency of synchronization operations and CPU utilization (max. 400% on a 4-way machine).

(i.e., quasi single-threaded) reduced the execution time by a few percent.

In the runs of pseudojbb, about half of the real (wall clock) time is spent for the creation of the warehouses, which is single-threaded, then follows the actual multi-threaded transaction processing. This is the reason for the relatively low CPU utilization of this benchmark. mtrt has only two threads (maximum utilization is 200%). hedc and jigsaw are I/O bound. The values for trade6 include the start-up and stop phase of WebSphere.

Thread-confined synchronization
Table 3 reports the fraction of synchronization operations that occur on thread-confined synchronization variables. The baseline (100%) is the total number of synchronization operations that occur for non-reentrant Java locks.

benchmark	thr-conf
mtrt	99.3
mol	82.7
monte	99.6
ray	81.3
hedc	89.6
jigsaw	45.5
pseudojbb (4 wh)	33.5
pseudojbb (16 wh)	18.9
trade6	30.3
average	64.5

Table 3: Fraction of synchronization events that are thread-confined in percent [%].

Thread-confined synchronization points are mainly an idiosyncrasy of Java's programming model and standard library implementation. A single-threaded application would report 100% redundancy according to this model. Despite the aggressive elimination of thread-confined locking by the compiler, yet a large fraction of thread-confined lock use and memory synchronization remains. The implementation of jigsaw and trade6, which is based on WebSphere, seems to be carefully designed to avoid and reduce the occurrence of thread-confined locking.

Thread and processor locality
Table 4 reports the fraction of synchronization operations that are redundant due to thread locality and processor locality. The baseline (100%) is again the total number of synchronization operations for non-reentrant Java locks.

Column *thr-loc* includes operations that are redundant due to thread-confinement reported in Table 3. The additional

benchmark	thr-loc	proc-loc	com-loc
mtrt	99.9	99.9	99.9
mol	98.4	98.2	98.5
monte	99.8	98.8	99.8
ray	99.0	99.0	99.2
hedc	97.2	97.6	98.1
jigsaw	84.8	91.8	91.8
pseudojbb (4 wh)	99.7	94.9	99.7
pseudojbb (16 wh)	99.7	91.7	99.7
trade6	74.5	64.5	76.7
average	94.8	92.9	95.9

Table 4: Fraction of synchronization events that are redundant due to a locality context in percent [%].

cases in Table 4, are due to immediate sequences of acquire and release operations inside the same thread but for synchronization variables that are shared, i.e., during their entire lifetime accessed by more than one thread. For all benchmarks, the case *thr-loc* covers an overwhelming majority of synchronization operations.

Column *proc-loc* reports redundancy due to processor locality. If different threads execute on the same processor – this model can include cases that are not identified as redundant according to thread locality. When a thread is migrated to a different processor, however, processor locality breaks for synchronization variables that are actually thread local. Hence, there is no clear benefit of one model over the other, assuming that thread migration occurs rarely. The redundancies reported in column *com-loc* reflects the combined model, which subsumes the cases of thread and processor locality.

Eager release or repetitive acquires
Table 5 reports a refinement of combined locality (Table 4, column com-loc). Especially for those benchmarks where synchronization locality not so pronounced (jigsaw and trade6), still a significant share of synchronization events can be deemed to be eager releases and repetitive acquires – and hence are classified as redundant. The numbers in Table 5 are rounded to a hundredth of a percent. In mtrt, e.g., for example, a mere 16 out of over 700,000 synchronization operations are not redundant.

benchmark	com-loc not eager or repetitive
mtrt	99.99
mol	99.17
monte	99.90
ray	99.56
hedc	99.66
jigsaw	98.05
pseudojbb (4 wh)	99.97
pseudojbb (16 wh)	99.99
trade6	93.65
average	98.88

Table 5: Fraction of synchronization events with combined locality that are not eager or repetitive in percent [%].

4. Related work
There are several lines of research that identify thread locality in the synchronization as optimization opportunity. In particular, there are highly efficient lock implementations that leverage the effect of thread-locality in locking: Kawachiya et. al. and Ogasawara et. al. [11, 12], e.g., avoid

atomic operations for locking up to the point where a lock is accessed by more than one thread. Tentative ownership locks (TO-lock) [16] avoid atomic operations even if locks are shared, exploiting phases of lock locality. The experimental findings on the existence and amount of lock locality is consistent with our report. Most of the lock optimization work for Java has focused on the optimization of atomic operations, not memory synchronization. This work studies the latter aspect in more detail than previous work under the presumption of the recently revised Java memory model [14].

Beyond the work on static lock elimination [1, 3, 23, 17], static analysis has also been tailored to the elimination of memory fences [4, 2, 15]. This work reports an ideal upper bound of memory fences that can be found to be redundant. As static analysis is necessarily conservative, the reductions reported in [2, 15, 4] are significantly lower than the findings reported in this paper.

Microprocessors commonly hide the effects of memory access re-ordering and optimize the execution of fence operations using the data speculation capabilities of the OOO execution pipeline [5, 7]. The effectiveness of these techniques depends on the immediate dynamically surrounding instruction context within which a fence executes, not the synchronization structure in the application. Hardware support for speculation, e.g., [7], allows to implement wait-free fences; this technique can reduce the impact of unnecessary fences on the execution time.

Memory synchronization is a performance critical aspect in distributed share memory systems. The significant overhead of eager release has been addressed in work on TreadMarks and Lazy Release Consistency [13].

5. Concluding remarks

Efficient memory synchronization remains an important topic for future multiprocessor systems. This work demonstrates on the example of multi-threaded Java benchmarks that most memory fences occurring in a program execution are unnecessary according to the memory consistency demanded by the programming language: On average more than 98% and a maximum of 99.9% of memory synchronization operations are redundant in the program executions that we observed. Acquire operations are *unnecessary* because previous synchronization in the same thread or processor have accomplished a consistent memory view earlier. Most release operations occur *proactively*, i.e., due to the uncertainty if a subsequent acquire occurs on the same or a different thread or processor. This usage model is a consequence of the fence and memory synchronization implemented in current multiprocessor architectures. We conclude that the model for memory synchronization offered by current multiprocessor systems is not well-aligned with the requirements of the widely used acquire-release software synchronization protocol.

References

[1] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proc. of Static Analysis Symposium (SAS '99)*, pages 19–38, Sept. 1999.

[2] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++: A C++ dialect for high performance parallel computing. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, pages 190–205, Mar. 1996.

[3] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19. ACM Press, Nov. 1999.

[4] X. Fang, J. Lee, and S. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the International Conference on Supercomputing (ICS'03)*, pages 285–294, June 2003.

[5] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the Intllerional Conference on Parallel Processing (ICPP'91)*, Aug. 1991.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'90)*, pages 15–26, June 1990.

[7] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP=RC? In *Proc. Int'l Symp. on Computer Architecture (ISCA'99)*, pages 162–171, May 1999.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[9] Intel Corporation. Intel architecture software developer's manual, volume3: System programming guide. http://developer.intel.com/design/PentiumIII/manuals/, Apr. 2002.

[10] JGF. Java Grande Forum multi-threaded benchmark suite, 1999.

[11] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 292–310, Nov. 2002.

[12] T. O. K. Kawachiya and A. Koseki. Lock reservation for java reconsidered. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'04)*, pages 560–584, June 2004.

[13] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[14] J. Manson, W. Pugh, and S. Adve. The java memory model. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'05)*, pages 378–391, 2005.

[15] S. Midkiff, J. Lee, and D. Padua. A compiler for multiple memory models. In *Rec. Workshop Compilers for Parallel Computers (CPC'01)*, June 2001.

[16] T. Ogasawara, H. Komatsu, and T. Nakatani. To-lock: Removing lock overhead using the owners' temporal locality. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 255–266, Oct. 2004.

[17] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 208–218, June 2000.

[18] E. Silha, C. May, and B. Frey. PowerPC User Instruction Set Architecture (Book I), 2003.

[19] SPEC. Standard Performance Evaluation Corporation - SPECjvm98, 1998.

[20] SPEC. Standard Performance Evaluation Corporation - SPECjbb2000, 2000.

[21] E. Stolte, C. von Praun, G. Alonso, and T. Gross. Scientific data repositories – designing for a moving target. In *Proceedings on the International Conference on Management of Data and Symposium on Principles of Database Systems (SIGMOD/PODS'03)*, pages 349–360, June 2003.

[22] W3C. World wide web consortium: Jigsaw - w3c's web server. `http://www.w3.org/Jigsaw`, 2003.

[23] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 187–206, Nov. 1999.