Lock Elision for Read-Only Critical Sections in Java

Takuya Nakaike
IBM Research - Tokyo
nakaike@jp.ibm.com

Maged M. Michael
IBM Thomas J. Watson Research Center
magedm@us.ibm.com

Abstract

It is not uncommon in parallel workloads to encounter shared data structures with read-mostly access patterns, where operations that update data are infrequent and most operations are read-only. Typically, data consistency is guaranteed using mutual exclusion or read-write locks. The cost of atomic update of lock variables result in high overheads and high cache coherence traffic under active sharing, thus slowing down single thread performance and limiting scalability.

In this paper, we present *SOLERO (Software Optimistic Lock Elision for Read-Only critical sections)*, a new lock implementation called for optimizing read-only critical sections in Java based on sequential locks. SOLERO is compatible with the conventional lock implementation of Java. However, unlike the conventional implementation, only critical sections that may write data or have side effects need to update lock variables, while read-only critical sections need only read lock variables without writing them. Each writing critical section changes the lock value to a new value. Hence, a read-only critical section is guaranteed to be consistent if the lock is free and its value does not change from the beginning to the end of the read-only critical section.

Using Java workloads including SPECjbb2005 and the HashMap and TreeMap Java classes, we evaluate the performance impact of applying SOLERO to read-mostly locks. Our experimental results show performance improvements across the board, often substantial, in both single thread speed and scalability over the conventional lock implementation (mutual exclusion) and read-write locks. SOLERO improves the performance of SPECjbb2005 by 3-5% on single and multiple threads. The results using the HashMap and TreeMap benchmarks show that SOLERO outperforms the conventional lock implementation and read-write locks by substantial multiples on multi-threads.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors—Optimization

General Terms Algorithms, Languages, Performance

Keywords Java, Just-In-Time compiler, synchronization, monitor, lock, optimization, lock elision

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

1. Introduction

Locking is the most common mechanism used by programmers for maintaining the consistency of shared data in multithreaded programs. The Java programming language supports the `synchronized` construct, which guarantees mutual exclusion among code blocks associated with an object's lock. A mutual exclusion [6] lock can be held by at most one thread at any time.

A common sharing pattern in multithreaded programming is the read-mostly pattern where most operations on a set of shared data are read-only, but occasionally some operations update the data or have external side effects. Read-write locks [3, 10] allow the concurrent execution of read-only operations. Read-write locks are supported in Java (5.0 and later) as part of the `java.util.concurrent` package. A read-write lock has two modes: read mode and write mode. Multiple threads may hold a read-write lock in read mode concurrently. While a thread holds such a lock in write mode, no other thread can hold the lock, whether in read mode or in write mode.

In both mutual exclusion and read-write locks, lock acquisition involves atomic instructions such as Compare-And-Swap or the pair Load-Linked and Store-Conditional. These instructions are substantially slower than regular load and store instructions. Furthermore, acquisition and release of a lock involve writing to lock variables, resulting in extra cache coherence traffic that is not inherent to the data accesses protected by the locks.

Sequential locks have been used to elide writes to lock variables for read-only critical sections in the Linux kernel. In sequential locks, the lock variable is implemented as a counter. The lock value is incremented for both lock acquisition and lock release. If the lock value is odd, then the lock is held. Otherwise, the lock is free. At the beginning of a read-only critical section, the lock value is read. At the end of the critical section, the lock value is compared with the value read at the beginning of the critical section. If the lock value is unchanged during the execution of the critical section, the execution of the critical section is completed without acquiring the lock. However, sequential locks are applicable only in limited cases, as they have many restrictions. For example, sequential locks are not re-entrant, and do not allow pointer accesses and loops in read-only critical sections that avoid writing to lock variables.

In this paper, we propose a new lock implementation called *SOLERO (Software Optimistic Lock Elision for Read-Only critical sections)* to elide writes to lock variables for read-only critical sections in Java. The idea of eliding writes to lock variables is based on the sequential lock. However, our lock implementation is capable of providing the full functionality of the conventional lock implementation used by synchronized blocks, and thus it can replace the conventional lock implementation of Java.

In SOLERO, the lock variable contains a counter value when the lock is free. At the beginning of a writing critical section which may have writes or side effects, a thread reads the counter value into

a local variable and writes its id to the lock variable with a lock bit atomically, so that other threads can detect that the lock is acquired and the thread can acquire the lock recursively, if needed. At the end of the critical section, the thread updates the lock variable with a new counter value, so that other threads can detect that the lock was acquired and released.

In a read-only critical section, a thread does not update the lock variable. If the lock is free and its value is the same at the beginning and at the end of the read-only critical section, then it can be concluded that the lock remained free throughout the duration of such critical section and no writing occurred, hence the reads performed in the critical section are guaranteed to be consistent. If the lock value is found to have changed, then the consistency of reads cannot be guaranteed and the read-only critical section is re-executed.

SOLERO includes mechanisms for recovery from faults caused by inconsistent reads. Since read-only critical sections are allowed to proceed optimistically without preventing a concurrent writing critical section from updating shared data, then it is possible for the speculatively read values to be mutually inconsistent. Read inconsistency may lead to faults such as null pointer dereference, division by zero, or infinite loops. We use Java exception handling mechanisms to catch exceptions due to faults. Our design also exploits infrequent asynchronous events, which are normally used to check garbage collection events, for read validation which detect and recover from infinite loops induced by inconsistent reads.

Our Java Just-In-Time (JIT) compiler identifies synchronized blocks as read-only if they do not include writes to heap data and events that may have side-effects, such as wait/notify and system calls. Our JIT compiler can also accept annotations to specify read-only synchronized blocks, in cases where it is difficult to identify a synchronized block as read only if the synchronized block includes virtual method invocations.

SOLERO can coexist with *bi-modal locking* [1, 11] mechanisms which are widely used by various JVMs. In a bi-modal locking mechanism, each lock has two modes: *thin* mode and *fat* mode. Our JVM employs the *tasuki lock* which is a bi-modal locking mechanism and allows the lock mode to switch bidirectionally. SOLERO supports the bidirectional switching of the lock mode the same as in the conventional lock implementation, though it can elide locks only in the thin mode.

Using Java workloads including SPECjbb2005 and the HashMap and TreeMap Java classes, we evaluate the performance impact of applying SOLERO to read-mostly locks. Our experimental results show performance improvements across the board, often substantial, in both single thread speed and scalability over the conventional lock implementation (mutual exclusion) and read-write locks. On SPECjbb2005, SOLERO improves the performance by 3-5% on a single thread and multiple threads. On the HashMap and TreeMap benchmarks, SOLERO outperforms the conventional lock implementation and read-write locks by substantial multiples in the multi-thread performance. The only case of underperformance by SOLERO was when lock elision was not applicable, and the overhead over conventional locking is less than 1% in all the benchmarks.

The contributions of this paper are:

- The design and implementation of SOLERO in a commercial JVM. It is the first JVM implementation of lock elision for read-only critical sections.
- Experimental study of the performance impact of lock elision for read-only critical sections.

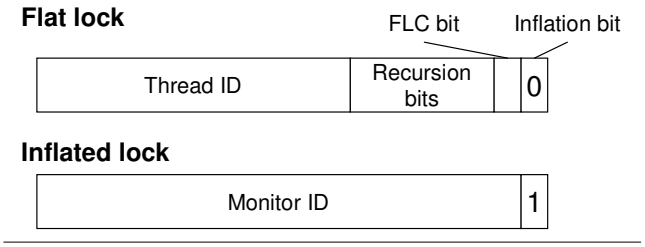


Figure 1. Conventional lock structure of Java

```
1:  while (true) {
2:    v = obj->lock;
3:    if (v == 0) {
4:      if (CAS(&obj->lock,0,thread_id)) {
5:        break; /* Fast path lock acquire */
6:      }
7:    } else {
8:      slow_enter(obj);
9:      break;
10:   }
11: }
12: ... /* Critical section */
13: if ((obj->lock & 0xff) == 0) {
14:   obj->lock = 0; /* Fast path lock release */
15: } else {
16:   slow_exit(obj);
17: }
```

Figure 2. Conventional lock operations (six recursion bits)

- Our experimental results indicate the superior performance of SOLERO over conventional lock implementations of both mutual exclusion locks and read-write locks.

The rest of the paper is organized as follows. Section 2 presents brief descriptions of typical lock implementation in Java and sequential locks. Section 3 describes SOLERO, our new lock implementation in a commercial JVM. Section 4 presents our experimental results. We describe mechanisms to extend SOLERO in Section 5. In Section 6, we discuss related work. We conclude with Section 7.

2. Background

2.1 Conventional Lock Implementation in Java

In Java, bi-modal locking mechanisms [1, 11] are commonly used to optimize lock operations. *Tasuki lock* is one of the state-of-the-art bi-modal locking mechanisms, and it is employed by our JVM. In tasuki lock, a lock has two modes: thin mode and fat mode. A lock is called *flat* in the thin mode, and it is called *inflated* in the fat mode. For a flat lock, a thread can acquire the lock at a low cost by writing its id to the lock variable atomically. If contention occurs frequently on a flat lock, the lock is transformed (inflated) into an inflated lock. For an inflated lock, a thread needs to pay a high cost to acquire the lock by using an OS monitor. If contention occurs rarely on an inflated lock, the lock can be reverted (deflated) to a flat lock.

Figure 1 shows the structure of a typical lock variable. A lock variable is initialized to zero. If the inflation bit of a lock is set, the lock is in the fat mode. Otherwise, the lock is in the thin mode. The FLC (flat-lock contention) bit is set when contention is detected on a flat lock, and it is used to inflate the lock.

```
0:  void three_tier_locking(Object *obj) {
1:    for (i = 0; i < tir3; i++) {
2:      for (j = 0; j < tier2; j++) {
3:        v = obj->lock;
4:        if (v == 0) {
5:          if (CAS(&obj->lock,0,thread_id)) {
6:            return; /* Acquired a flat lock */
7:          }
8:        } else if ((v & 0xff) != 0) {
9:          /* Recursive, inflated, or FLC */
10:         goto END_OF_SPIN;
11:       }
12:       for (k = 0; k < tier1; k++);
13:     }
14:     yield();
15:   }
16: END_OF_SPIN:
17:   ...
18:   return;
19: }
```

Figure 3. Three-tier locking scheme

Figure 2 shows pseudocode for acquiring a flat lock. If the lock value is zero, the thread tries to write its id to the lock variable atomically[1]. If the lock value is not zero, the thread calls the slow-path code (`slow_enter`) that: 1) checks if the thread already holds the lock for recursive locking, 2) resolves lock contention if the lock is held by another thread, and 3) enters the OS monitor if the lock in the fat mode.

Contention on a flat lock is resolved using a *three-tier locking* scheme with three nested loops [7]. Figure 3 shows the psuedocode for three-tier locking. In the middle-tier loop, lock acquisition (atomic updates to the lock variable) is tried repeatedly (line 5). If the lock acquisition fails, CPU cycles are wasted in the innermost loop (line 12), as a backoff mechanism. When a lock cannot be acquired after a number of attempts in the middle-tier loop, the CPU is yielded in the outermost loop (line 14).

When a flat lock cannot be acquired after a number of outermost loop iterations, the lock is inflated. The inflation can also occur when the bits of the recursion counter saturate. For inflation, a thread retrieves an OS monitor mapped to a monitor object, and then writes the pointer of the OS monitor to the lock variable after entering the OS monitor and acquiring the flat lock. An inflated lock can be deflated by writing zero to the lock variable in the slow-path code for lock release (in `slow_exit` in Figure 2).

2.2 Sequential Locks

Sequential locks (also called seqlocks) are used to elide writes to lock variables in read-only critical sections in the Linux kernel, and they are the algorithmic basis of our lock implementation. Figure 4(a) shows the pseudocode to acquire and release a sequential lock in a writing critical section. The lock variable is implemented as a counter. When a lock is acquired, the lock value is incremented. When a lock is released, the lock value is incremented again. When

[1] The atomic update of the lock variable can use the Compare-and-Swap (CAS) atomic primitive–or alternatively the pair Load-Linked (LL) and Store-Conditional (SC)–available on all mainstream processor architectures. In its simplest form, CAS takes three arguments: a memory location, an expected value, and a new value. If the memory location holds the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred. If it returns true, it is said to succeed. Otherwise, it is said to fail. LL/SC can be used to implement CAS. For generality, we present the algorithms in this paper using CAS.

```
1:  while (true) {
2:    v = lock;
3:    if ((v & 0x1) == 0) {
4:      if (CAS(&lock,v,v+1)) {
6:        break; /* Acquired lock */
7:      }
8:    }
9:  }
10: ... /* Writing critical section */
11: lock++; /* Release lock */
```

(a) Lock acquisition and release in a writing critical section

```
1:  while (true) {
2:    v = lock;
3:    if ((v & 0x1) != 0) continue;
4:    ... /* Read-only critical section */
5:    if (v == lock)
6:      break; /* Success. Lock unchanged */
7:  }
```

(b) Lock elision in a read-only critical section

Figure 4. Lock acquisition, release, and elision in sequential locks

the lowest bit of the lock value is not zero, in other words, the lock value is odd, the lock is held. Otherwise, the lock is free.

Figure 4(b) shows the pseudocode to elide writes to lock variables in read-only critical sections. At the beginning of the critical section, the lock value is stored in a local variable. If the lowest bit of the lock value is zero, the critical section is executed. At the end of the critical section, the lock value is compared with the value read at the beginning of the critical section. If the lock value is unchanged during the execution of the critical section, the execution of the critical section is considered to be completed successfully. Otherwise, the critical section is re-executed.

Although a sequential lock allows eliding writes to lock variables in read-only critical sections, it has the following limitations:

- Sequential locks are not re-entrant, because lock variables do not hold information about lock owners.
- Sequential locks do not allow pointer accesses and loops inside read-only critical sections. Inconsistent reads, which can be caused by the speculative execution of read-only critical sections, can cause faults such as invalid pointer dereference, division by zero, and infinite loops.

3. SOLERO

In this section, we describe SOLERO, our new lock implementation, which provides complete Java lock functionality and allows eliding writes to lock variables for read-only critical sections.

3.1 Lock Operations for Writing Critical Sections

The inflated lock structure for SOLERO is the same as in conventional lock implementation. The SOLERO flat lock structure is shown in Figure 5. Each SOLERO flat lock contains a lock bit. When the lock bit is set, the lock is held. Otherwise, the lock is free. When a lock is free, the higher bits of the lock variable hold a counter value like a sequential lock. When a lock is held, the higher bits hold a thread id. One bit from the recursion bits is used for the lock bit.

Figure 6 shows the pseudocode to acquire and release a lock in SOLERO. Before each acquisition of a lock, the lower three bits of the lock value are masked out. If all of the lower three bits are

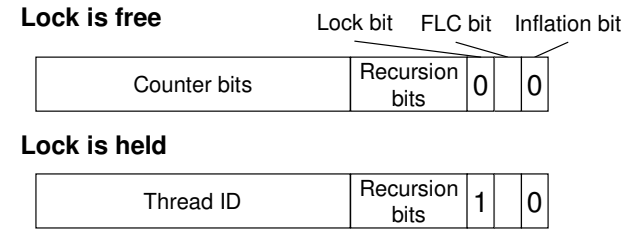


Figure 5. SOLERO lock structure

```
1:  while (true) {
2:    v1 = obj->lock;
3:    if ((v1 & 0x7) == 0) {
4:      val = thread_id+LOCK_BIT;
5:      if (CAS(&obj->lock,v1,val)) {
6:        /* Fast path for lock acquisition */
7:        break;
8:      }
9:    } else {
10:     v1 = solero_slow_enter(obj);
11:     break;
12:   }
13: }
14: ... /* Writing critical section */
15: v2 = obj->lock;
16: if ((v2 & 0xff) == LOCK_BIT) {
17:   /* Fast path for lock release */
18:   obj->lock = v1 + 0x100;
19: } else {
20:   solero_slow_exit(obj,v1);
21: }
```

Figure 6. SOLERO lock operations for writing critical sections

zero, the lock variable is atomically updated with the thread id plus the lock bit. The lock value read before the atomic update is kept in a local variable until the lock is released. We refer to such a local variable as the local lock variable in the remainder of this paper. In this figure, the local lock variable is `v1`. When the lower three bits are not zero, the slow-path code (`solero_slow_enter`) is called.

Before each release of the lock, the lower eight bits are masked out from the lock value and compared with the lock-bit value where only the lock bit is set. If only the lock bit is set among the lowest eight bits, then the flat lock is released by incrementing the value of the local lock variable and updating the lock variable with the incremented value. SOLERO needs the following operations for acquisition and release of a lock in addition to the lock operations of the conventional lock implementation:

- Masking out the lower three bits of the lock value before lock acquisition
- Adding the lock bit to the thread id as a value to be written to the lock variable for lock acquisition
- Keeping the lock value read before lock acquisition in a local variable until lock release
- Incrementing the value stored in the local lock variable for lock release

3.2 Lock Elision for Read-Only Critical Sections

Our JIT compiler identifies read-only synchronized blocks by looking for writes and side effects within synchronized blocks. For synchronized blocks without potential writes or side effects, it generates the code to elide writes to lock variables. The following operations are not allowed in read-only synchronized blocks:

- Writes to instance variables, static variables, and array elements
- Writes to the local variables that are live at the beginning of the critical section
- Invocations of methods, other than those involved in throwing runtime exceptions

We chose not to apply lock elision to synchronized blocks with writes to local variables that are live at the beginning of the critical section, because otherwise we would need to keep and restore the values that has been kept in such local variables before executing the critical section if the speculative execution of the critical section fails to complete [17]. Although we do not explicitly forbid read-only synchronized blocks from creating new objects by using the `new` construct, object creation rarely occurs in read-only synchronized blocks because they involve the invocation of constructors where writes to instance variables typically occur.

Throwing runtime exceptions such as NullPointerException and ArrayIndexOutOfBoundsException is allowed in read-only synchronized blocks as long as the references of the exception objects are not written within the read-only synchronized block. Although throwing runtime exceptions involves the creation of objects and writes to the instance variables, the exception objects are not visible until the exception are caught by the catch blocks and the references of the exception objects are written to shared area.

In order to expand the applicability of lock elision, we introduce an annotation `@SoleroReadOnly` to specify read-only synchronized blocks. This annotation can be added to a method that includes read-only synchronized blocks. Our JIT compiler recognizes the synchronized blocks included in the annotated method as read-only. The annotation is useful even when the JIT compiler uses the inter-procedure analysis to identify the synchronized blocks that include the method invocations as read-only, because it is difficult to identify the invocation targets in Java where invocation targets are determined dynamically by dynamic class loading and virtual methods.

Figure 7 shows the pseudocode to elide writes to lock variables for a read-only critical section. At the beginning of the critical section, the lower three bits of the lock value are masked out. If the lower three bits are zero, a thread proceeds to the inside of the critical section after keeping the lock value in a local lock variable. At the end of the critical section, the value of the local lock variable is compared with the latest lock value. If the lock value is unchanged throughout the duration of the critical section, the critical section is considered to have completed successfully. In the fast path, the read-only critical section does not write to lock variables, and it has fewer operations than the conventional lock implementation.

If the lower three bits of the lock value are not zero at the beginning of the read-only critical section, the thread calls the slow-path code (`solero_slow_read_enter`) shown in Figure 8. If a thread already holds the flat lock, it increments the recursion count. If the lock is already inflated (the inflation bit is set), the inflated lock is acquired by using the OS monitor. If the lock is flat and is held by another thread, then the thread waits for the lock to be released in three-nested loops like the three-tier locking scheme. If at some point in the three-nested loops, the lowest three bits of lock value are found to be zero, the thread exits the loops, stores the lock value in the local lock variable, and proceeds to execute the read-only critical section. However, if the lowest three bits of

```
1:  v = obj->lock;
2:  if ((v & 0x7) != 0)
3:    v = solero_slow_read_enter(obj);
4:  while (true) {
5:    ... /* Read-only critical section */
6:    if (v == obj->lock) {
7:      /* Lock elision succeeded */
8:      break;
9:    } else if (solero_slow_read_exit(obj,v)) {
10:     break;
11:   } else {
12:     /* Re-execute the critical section */
13:     v = solero_slow_enter(obj);
14:   }
15: }
```

Figure 7. Lock elision in SOLERO for a read-only critical section

the lock value are not zero throughout the three-nested loops, the thread inflates the lock.

Before inflation, a thread must enter the OS monitor that is mapped to the monitor object and acquire the flat lock. This is the same as the conventional lock implementation. Additionally, SOLERO needs to increment the counter value that is read before the acquisition of the flat lock and to store the incremented value in the OS monitor. When a lock is deflated, the incremented value in the OS monitor is written back to the lock variable. As a result, concurrent threads executing speculative read-only critical sections while the lock is inflated and deflated can detect the change in the lock value. After the inflated lock is acquired, zero is returned from the slow-path code (`solero_slow_read_enter`) and stored in a local lock variable to force the thread to call the slow-path code (`solero_slow_read_exit`) at the end of the critical section. The lock value never matches with zero because the inflation bit of the lock value is set.

Figure 9 shows the pseudocode of the slow-path code called at the end of the critical section. If a thread already holds the flat lock and the recursion count is not zero, it decrements the recursion count. If a thread holds the flat lock and the recursion count is zero, it releases the flat lock by incrementing the value of the local lock variable and updating the lock variable with the incremented value. If a thread holds the inflated lock, it releases the inflated lock. In these three cases, a non-zero value is returned and the critical section is completed. If the lock value does not match these three cases, then it must be the case that the lock value was changed by another thread during the execution of the read-only critical section. As a result, a zero value is returned, and the critical section is re-executed.

As shown in Figure 7, `solero_slow_enter` is called to acquire the lock when `solero_slow_read_exit` returns zero. This is the fallback mechanism to avoid starvation. In our implementation, the fallback occurs after one failure to execute the read-only critical section. This can be expanded so that the fallback occurs after a larger number of failures.

3.3 Recovery from Inconsistent Reads

Since read-only critical sections are allowed to proceed optimistically without preventing a concurrent writing critical section from starting, then it is possible for read values to be inconsistent. Read inconsistency can lead to faults such as null pointer dereference, division by zero, and infinite loops. If not handled, these faults can occur to correct programs. Therefore, SOLERO must prevent or detect and recover from such faults.

```
1:  int solero_slow_read_enter(Object *obj) {
2:    if (test_recursion(obj)) {
3:      obj->lock += 0x8;
4:      return 0;
5:    }
6:    for (i = 0; i < tier3; i++) {
7:      for (j = 0; j < tier2; j++) {
8:        v = obj->lock;
9:        if ((v & 0x7) == 0) {
10:         return v;
11:       } else if ((v & 0x3) != 0) {
12:         goto INFLATION;
13:       }
14:       for (k = 0; k < tier1; k++);
15:     }
16:   yield();
17:   }
18: INFLATION:
19:   ...
20:   return 0;
21: }
```

Figure 8. Slow-path entry code for SOLERO read-only critical sections

```
1:  int solero_slow_read_exit(Object *obj, int v) {
2:    if (test_recursion(obj)) {
3:      obj->lock -= 0x8;
4:      return 1;
5:    } else if (hold_flat_lock(obj)) {
6:      obj->lock = v + 0x100;
7:      check_flc(obj);
8:      return 1;
9:    } else if (hold_inflated_lock(obj)) {
10:     release_inflated_lock(obj);
11:     return 1;
12:   } else {
13:     return 0;
14:   }
15: }
```

Figure 9. Slow-path exit code for SOLERO read-only critical sections

To detect and recover from read inconsistency, our JIT compiler generates a catch block for each synchronized block so that any exception raised in the synchronized block can be caught before leaving the synchronized block. Note that this mechanism is not specific to SOLERO and it is required for the conventional lock implementation to force a lock to be released before leaving the synchronized block. The difference between SOLERO and the conventional lock implementation is that the latter releases the lock and throws the exception to the next higher catch block, while in SOLERO, read consistency is validated by comparing the lock variable with the local lock variable. If they are equal, then the reads are consistent and the exception is genuine, i.e., inherent to the program and not merely due to SOLERO allowing reads to be inconsistent, and the exception is thrown to the higher level. However, if the values are not equal, then the consistency of reads is not guaranteed, and hence it is possible that the exception is not genuine, then the exception is ignored and the execution of the read-only critical is retried.

In order to avoid infinite loops due to inconsistent reads, our JVM sends occasionally asynchronous events to threads. Each

thread checks the existence of the event at an *asynchronous check point* which is inserted by the JIT compiler at method entries and before backward branches of loops. When a thread receives the event, it validates the consistency of reads by comparing the lock value with the local lock value. If a thread finds that reads may be inconsistent, it throws an exception to go back to the beginning of the critical section and retry executing the read-only critical section.

For read consistency validation, our JIT compiler prepares the information about the local lock variables and the local variables that hold the references to monitor objects. Note that storing a local reference to a held lock is not specific to SOLERO. Conventional lock implementations use such local references to held locks to release a lock when an exception occurs inside a synchronized block and escapes outside the synchronized block. When a thread receives an event to validate read consistency, it walks the call stack and checks whether or not it is in a read-only critical section. If a thread is in a read-only critical section, it compares the value of the local lock variable with the lock value read from the monitor object. If the lock value is changed, an exception is thrown (which is then caught as described above and then causes the retry of the critical section).

3.4 Memory Ordering

Compiler optimizations and multiprocessor architectures with weak memory models often reorder independent memory accesses or cause them to appear out of program order to other threads. However, the correctness of the algorithms for SOLERO lock operations depends on enforcing certain orderings among memory accesses within these operations. In this subsection we describe required ordering in the fast paths of SOLERO operations.

In the fast path of a writing critical section (Figure 6), the CAS of the lock variable in line 5 must be ordered before loads and stores inside the critical section (line 14), and these loads and stores must be ordered before the store to the lock variable in line 18. In the fast path of a read-only critical section (Figure 7), the load from the lock variable in line 1 must be ordered before loads in the read-only critical section (line 5), and these loads must be ordered before the load of the lock variable in line 6. The above ordering requirements are needed for guaranteeing that mutual atomicity of critical sections protected by the same lock.

The Java Memory Model [9] implies certain ordering requirements related to locks. Relevant to SOLERO are the requirements that (a) loads and stores that precede a critical section in program order must not be reordered after the corresponding exit section, (b) loads and stores that follow a critical section in program order must not be reordered before the corresponding entry section, and (c) all critical sections for the same lock must have a total serial order.

In order to conform to the above mentioned Java lock ordering semantics, loads and stores that precede read-only critical sections must be ordered before loads inside the critical section, and the latter loads must be ordered before loads and stores that follow the critical section.

On the PowerPC architecture, its weak memory model requires memory-fence instructions to enforce the above orderings. On the X86 architecture and on SPARC architectures with TSO (Total Store Order) memory model, no explicit fences are needed, except for ordering stores that precede a read-only critical sections before the loads inside the critical sections, if the lock acquisition is elided.

4. Experimental Results

4.1 Experimental Environment

We used a 16-way 4.7GHz Power6 multiprocessor system for performance evaluation. All of the performance measurements were done on a fixed hardware configuration where sixteen processors are active. When we see the scalability or the single-thread performance, we changed the number of the software (Java) threads. We implemented SOLERO in a 64-bit commercial JVM where each object has a 64-bit lock variable. In the conventional lock implementation, a flat-lock variable consists of: one inflation bit, one FLC bit, six recursion bits, and 56-bit thread id field. In SOLERO, a flat-lock variable consists of: one inflation bit, one FLC bit, one lock bit, five recursion bits, and the 56-bit counter or thread id field. We did not take into account for the wrap around of the counter because the 56-bit counter is impossible to wrap around in less than 68 years, assuming the fastest rate for acquiring and releasing the lock.

As mentioned in Section 3.4, we need to choose the appropriate PowerPC memory-fence instructions for SOLERO. At the beginning of a writing critical section, we inserted the `lwsync` instruction [16] immediately after the atomic update to the lock variable (line 6 in Figure 6). At the beginning of a read-only critical section, we inserted the `sync` instruction immediately after the store of the lock value to the local lock variable. Since both the `lwsync` and `sync` instructions are more expensive than the `isync` instruction which is inserted at the beginning of a critical section in the conventional lock implementation, SOLERO on the PowerPC architecture has a larger overhead of memory ordering than the conventional lock implementation. However, read-only critical sections avoid the overheads of atomic instructions.

We used SPECjbb2005 [15], the da-capo benchmarks [2], and the following three micro-benchmarks for performance evaluation:

- **Empty**: This executes an empty synchronized block iteratively.
- **HashMap**: This accesses a single `java.util.HashMap` object in a synchronized block. The number of entries is 1K.
- **TreeMap**: This accesses a single `java.util.TreeMap` object in a synchronized block. The number of entries is 1K.

We used the version 9.10 of the da-capo benchmarks to evaluate the performance of SOLERO with the four multi-thread applications: 1) h2, 2) tomcat, 3) tradebeans, and 4) tradesoap. We used the annotation to specify a read-only synchronized block for the methods which call the `get` methods of the HashMap and TreeMap classes. In SPECjbb2005 and the da-capo benchmarks, we relied on the JIT compiler to find read-only synchronized blocks without using the annotation. Table 1 shows the lock statistics of each benchmark.

We compared the following three lock implementations.

- **Lock**: The conventional lock implementation used by synchronized blocks
- **RWLock**: The read-write lock in `java.util.concurrent` package
- **SOLERO**: Our lock implementation

We manually replaced synchronized blocks with read-write locks in three micro-benchmarks. The read mode is applied for read-only synchronized blocks such as the synchronized block to get an object from a `java.util.HashMap` object.

We ran each benchmark five times on each lock implementation and calculated the average throughput. On each run, the throughput is continuously measured five times and the best score is used to calculate the average throughput. The reason why we used the best

Table 1. Lock statistics

Benchmarks	Lock frequency (millions locks per sec)	Ratio of read-only locks (%)
Empty	12.8	100.0
HashMap (0% writes)	5.4	100.0
HashMap (5% writes)	5.3	95.0
TreeMap (0% writes)	1.7	100.0
TreeMap (5% writes)	1.6	95.0
SPECjbb2005	6.2	53.6
h2	2.0	0.0
tomcat	7.3	3.7
tradebeans	1.7	0.3
trradesoap	3.4	11.4

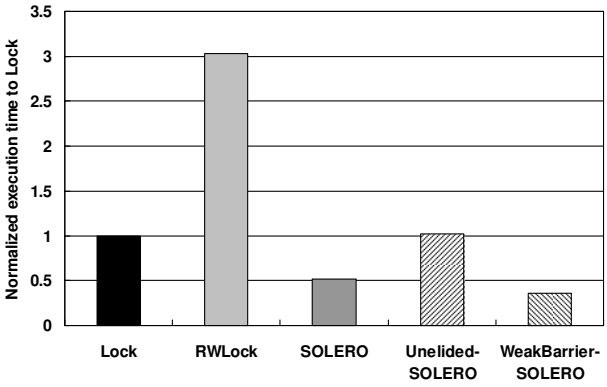


Figure 10. Lock overhead (execution time when executing an empty synchronized block on a single thread)

score is to exclude the impact of the JIT compilation time from the performance evaluation.

4.2 Experimental Results

Figure 10 shows the lock overhead of each lock implementation on Empty. Of course, the empty synchronized block is treated as read-only. For SOLERO, we measured the execution time when we did not elide locks for read-only critical sections (Unelided-SOLERO), and when we used the same memory-fence instructions as the conventional lock implementation, in other words, we used the incorrect memory-fence instructions (WeakBarrier-SOLERO). As shown in this figure, SOLERO reduces the lock overhead by 50% compared to the conventional lock implementation though it pays a significant overhead for memory ordering. SOLERO degrades the performance by 1.4% compared to the conventional lock when it does not elide locks in the empty synchronized block. Note that 1.4% is the upper bound of the overhead, and the overhead of SOLERO is much smaller in more realistic cases as shown below.

Figure 11 shows the single-thread performance of each lock implementation. Under the 0% writes (i.e. 100% read-only) configuration of HashMap, SOLERO improves the performance by 7.8% compared to the conventional lock implementation. Under the 5% writes configuration of HashMap, SOLERO improves the performance by 6.4%. In SPECjbb2005, SOLERO improves the performance by 4.2%. In contrast, SOLERO has little performance improvement in TreeMap. This is because TreeMap has a lower lock frequency than the other two benchmarks as shown in Table 1. In all of the micro-benchmarks, read-write locks substantially underperform the conventional lock implementation, as they are not inlined and involve a level of indirection in accessing lock variables, unlike SOLERO and the conventional Java lock implementation.

Similarly with the Empty benchmark, we measured the performance of SOLERO in the HashMap, TreeMap, and SPECjbb2005 benchmarks without eliding locks and with the incorrect memory-fence instructions though we do not plot the results in Figure 11. In the three benchmarks, the performance degradation of SOLERO compared to the conventional lock implementation is less than 1% when locks are not elided. This is the only disadvantage of SOLERO to the conventional lock implementation. In the HashMap, TreeMap, and SPECjbb2005 benchmarks, SOLERO has 20%, 7%, and 5% overhead for memory ordering respectively.

Figure 12 shows the multi-thread performance of each lock implementation in HashMap. Under the 0% writes configuration of HashMap, SOLERO shows almost linear scalability though the other two lock implementations degrade the performance as the number of threads increases. Under the 5% writes configuration of HashMap, SOLERO shows higher performance than the other two lock implementations though it drops the performance when the number of thread is more than two. The reason for the performance degradation is lock contentions and failures in the speculative execution of read-only critical sections. Figure 15 shows the ratio of the failures in the speculative execution. The 5% writes configuration of HashMap shows 23% failures on sixteen threads.

Figure 12(c) shows the performance of a fine-grained version of HashMap. In the fine-grained version, multiple hash-map objects are prepared, and each hash-map object is accessed through its own lock. In this experiment, the number of the hash-map objects equals to the number of threads. In the fine-grained version, SOLERO improves the performance until eight threads. Although the other two lock implementations also improve the performance as the number of threads increases, SOLERO has higher performance than the other two lock implementations on any number of threads. The fine-grained lock version of HashMap shows 3% failures on sixteen threads.

Figure 13 shows the multi-thread performance of TreeMap. Under the 0% writes configuration of TreeMap, SOLERO shows almost linear scalability and higher performance than the other two lock implementations. Under the 5% writes configuration of TreeMap, SOLERO improves the throughput until eight threads and shows higher performance than the other two lock implementations on any number of threads. The failure ratio in TreeMap is 35% on sixteen threads.

Figure 14 shows the multi-thread performance of SPECjbb2005. SPECjbb2005 is known to be highly scalable with minimal lock contention, as such it offered little opportunity for SOLERO to show its potential for higher scalability than the other lock implementations. However, the single-thread advantage of SOLERO over the other lock implementations carried over proportionally to higher number of threads, yielding the highest throughput across the various numbers of threads. SPECjbb2005 shows almost 0% failures on any number of threads.

Finally, we show the performance of the da-capo benchmarks in Figure 16. Unfortunately, we could not see a major difference between the conventional lock implementation and SOLERO. This is because the ratio of the read-only synchronized blocks is low compared to the other benchmarks as shown in Table 1. The read-only ratios of the da-capo benchmarks do not exceed 11% while SPECjbb2005 has 53% read-only ratio. While SOLERO has little performance improvement to the conventional lock implementa-

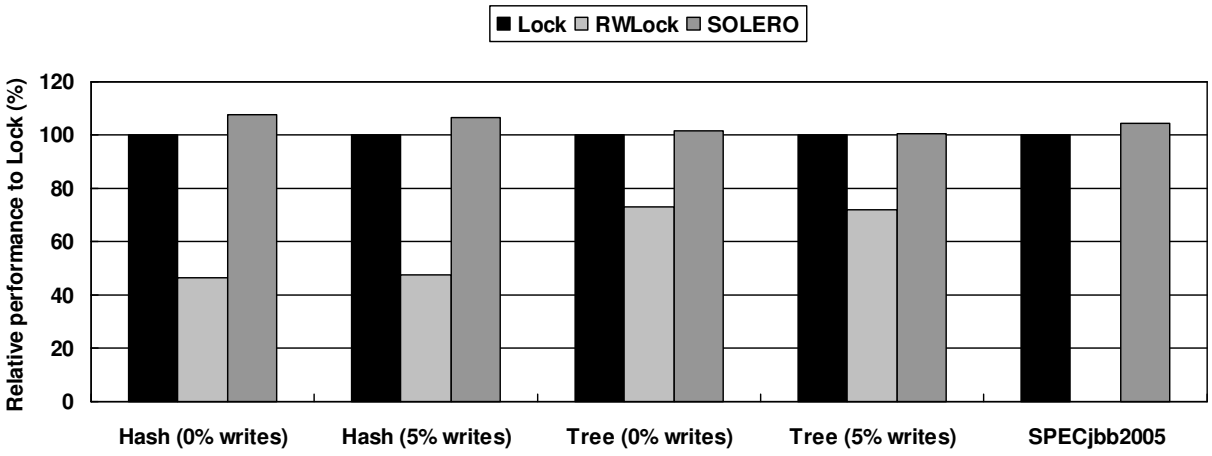


Figure 11. Single-thread performance of HashMap, TreeMap, and SPECjbb2005 (the performance of read-write locks is not measured in SPECjbb2005)

tion, its performance degradation is negligible (less than 1%). This indicates that SOLERO has negligible overheads even in the applications where there are few read-only critical sections.

5. SOLERO Read-Mostly Extension

In this section, we present an extension to apply SOLERO to critical sections that are likely to be read-only but there is some chance that they may involve writes or side effects. In this extension of SOLERO, the JIT compiler identifies a critical section that contains writes or side effects as read-mostly if the execution of those writes or side effects is rare. The JIT compiler generates the code to elide locks for read-mostly critical sections the same as for read-only critical sections. In addition, it inserts the code to acquire the lock before each write or side effect within read-mostly critical sections.

Figure 17 shows the pseudocode for a read-mostly critical section with the capability to acquire the lock if a write or a side effect is encountered. Before a thread executes writes or actions with side effects, it first tries to acquire the lock using the local lock value read at the beginning of the critical section. The CAS for lock acquisition uses the local lock value in order to guarantee that all the prior reads in the so-far read-only critical section are consistent. If the lock acquisition succeeds or the thread already holds the lock, then the thread can continue the critical section without returning to its beginning. If the thread fails to acquire the lock using the local lock value and it does not already hold the lock, then it is unsafe to proceed with the remainder of the critical section. The thread calls `solero_slow_enter` to acquire the lock. It then re-executes the critical section, but this time while holding the lock.

6. Related Work

The seqlock mechanism used in the Linux kernel is the algorithmic basis of our SOLERO implementation. It is applied in limited cases where a single variable is read. The mechanism is attributed to Stephen Hemminger. We are not aware of prior work in the literature that studies the design or performance of seqlocks.

In this paper, we expand the core seqlock mechanism by developing it to be applicable to general critical section that access multiple locations. We describe our design for a Java implementation that we prototyped in a commercial JVM. We evaluate the performance of that design using standard benchmarks and data structures.

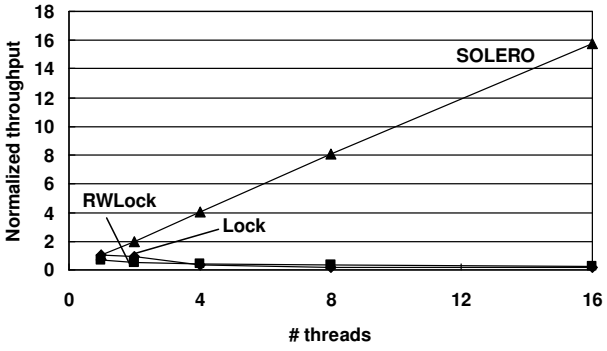
David Dice [4] discusses using sequential locks in Java at the programmer level, using Java constructs and concurrency library. The SOLERO implementation is at the JVM/JIT level, with no need for Java program changes.

Many optimizations have been proposed to reduce the locking overhead for Java. One of the major optimizations is bi-modal locking which reduces the overhead by avoiding the use of OS monitors [1, 11]. A three-tier locking scheme can be used with bi-modal locking to further reduce the locking overhead [7]. As described in this paper, these lock optimizations can co-exist with SOLERO.

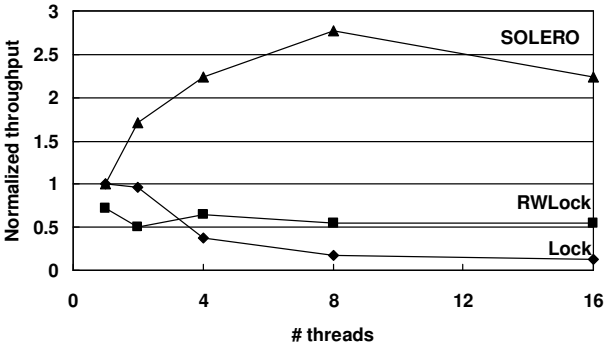
Eliminating atomic operations such as Compare-And-Swap or the pair Load-Linked and Store-Conditional on objects that turn out to be thread private is also popular to reduce the locking overhead [8, 14]. This kind of technique is complementary to the SOLERO mechanism. The former mechanisms are useful when locks turn out to be used by a single thread, and thus are unnecessary. While SOLERO is useful when locks are indeed shared by multiple threads, but the critical sections under these locks are mostly read-only. In the absence of employing lock elimination mechanisms for private locks, the SOLERO mechanism can be useful in partially reducing the unnecessary lock overheads if some of the critical sections are read-only.

There are some techniques to elide locks based on transactional memory (TM) runtime systems or TM-like runtime systems which are implemented in hardware or software [5, 12, 13, 18]. These lock elision techniques allow a thread to enter a critical section speculatively without acquiring the lock. The runtime system forces a thread to rollback to the beginning of the critical section when the thread causes data conflicts or encounters an operation that cannot be executed speculatively, such as I/O. After rollback, the thread can fallback to conventional locking and execute the critical section non-speculatively.

The existing lock elision techniques require memory access tracking within critical sections for data conflict detection. Although the overhead of memory access tracking is very small when the runtime system is implemented in hardware, currently, no such hardware is available. If the runtime system is implemented in software, the large overhead is caused by memory access tracking. Since SOLERO does not need memory access tracking within critical sections, it does not require any extra hardware support and its overhead is very small though it has a smaller coverage of spec-



(a) 0% writes



(b) 5% writes

Figure 13. Multi-thread performance of TreeMap

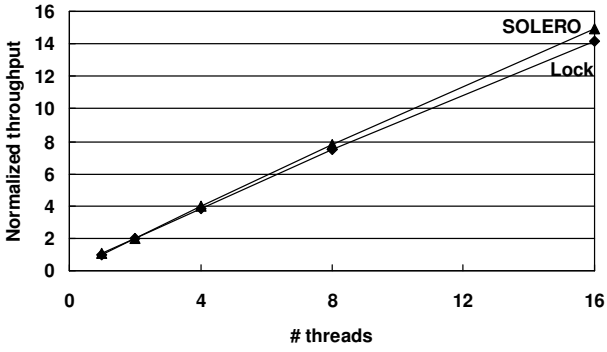


Figure 14. Multi-thread performance of SPECjbb2005

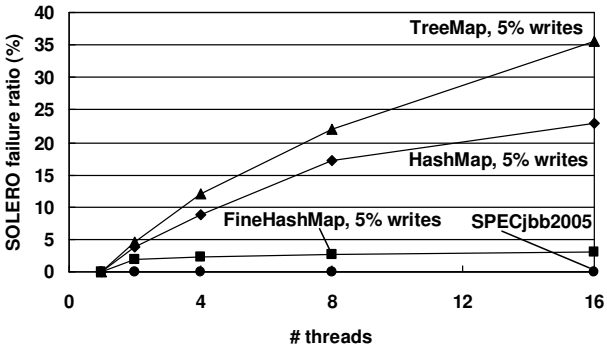


Figure 15. Ratio of failures in the speculative execution of read-only synchronized blocks in SOLERO

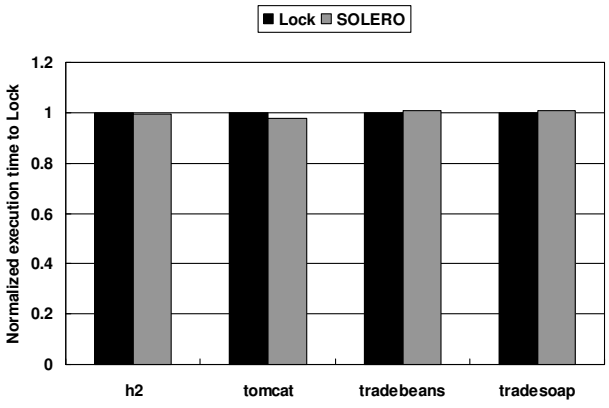
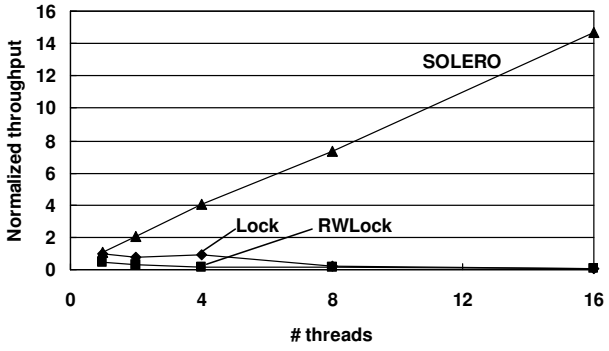
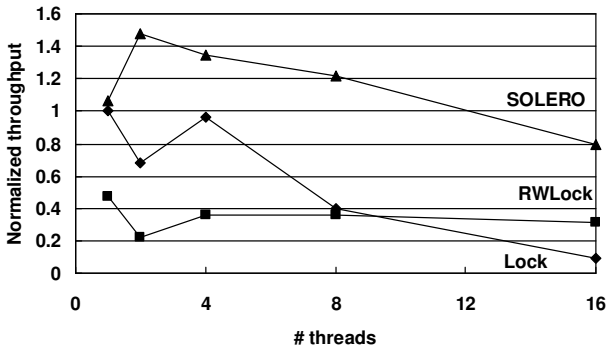


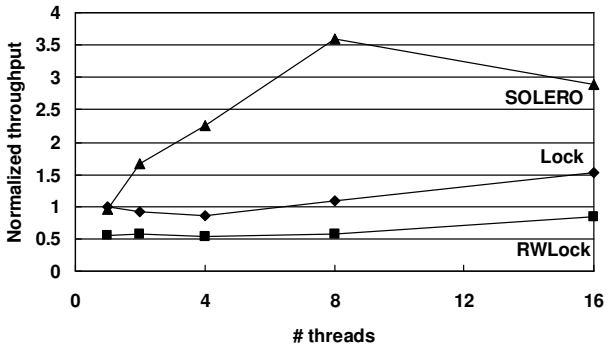
Figure 16. Performance of da-capo benchmarks



(a) 0% writes



(b) 5% writes



(c) 5% writes (fine-grained version)

Figure 12. Multi-thread performance of HashMap (the fine-grained version uses multiple hash-map objects to reduce lock contentions)

```
1:  v = obj->lock;
2:  if ((v & 0x7) != 0)
3:    v = solero_slow_read_enter(obj);
4:  while (true) {
5:    /* Read-mostly critical section */
6:    ...
7:    /* Write encountered. Must hold lock */
8:    if (CAS(&obj->lock,v,thread_id+LOCK_BIT) ||
9:        hold_lock(obj)) {
10:     /* Now holding lock - Safe to write */
11:   } else {
12:     /* Not holding lock - Slow acquire */
13:     v = solero_slow_enter(obj);
14:     continue; /* Re-execute critical section */
15:   }
16:   /* Rest of critical section and exit code */
17:    ...
18: }
```

Figure 17. Lock acquisition within a read-mostly critical section

ulatively executed critical sections than transactional lock elision techniques.

7. Conclusion

In this paper we presented SOLERO a Java implementation of a lock elision mechanism that avoids writing to lock variables when used for read-only critical sections. SOLERO provides full lock functionality, including reentrance, bi-modality, and multi-tier contention management, and hence it is suitable for replacement of conventional locks, without requiring source code modification. We implemented SOLERO in a commercial JVM, and evaluated its performance in comparison to the conventional lock implementation of synchronized blocks (i.e., mutual exclusion locks) and read-write locks.

Our performance results indicate that SOLERO consistently outperforms conventional lock implementation and read-write locks on a single thread and multiple threads, and under high read concurrency. On the SPECjbb2005 benchmark, SOLERO outperforms the conventional lock implementation by 3-5% on single and multiple threads. The results using the HashMap and TreeMap benchmarks show that SOLERO outperforms the conventional lock implementation and read-write locks by substantial multiples on multi-threads. However, SOLERO by itself does not reduce write contention. Therefore, under high write contention, fine-grained designs may be useful. If the fine-grained designs are lock-based, there is no inherent impediment to applying SOLERO to fine-grain locks. The only disadvantage of SOLERO is the overhead of acquiring and releasing locks. However, the overhead is less than 1% in all of the benchmarks.

Acknowledgments

We are grateful to the JIT compiler development team at IBM Toronto for advising us on our paper and for participating in helpful discussions. We also thank the members of our group at the IBM Research - Tokyo and the IBM T. J. Watson Research Center for advising us about the paper.

References

[1] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *PLDI'98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, June 1998.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[3] P.-J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, Oct. 1971.

[4] D. Dice. Seqlocks in java - readers shouldn't write synchronization metadata. David Dice's Weblog, Sept. 2006.

[5] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS'09: Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168, Mar. 2009.

[6] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, Sept. 1965.

[7] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable jvms. *IBM Systems Journal*, 39(1):151–174, 2000.

[8] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA'02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–141, 2002.

[9] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGCAT symposium on Principles of programming languages*, pages 378–391, 2005.

[10] J. M. Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPoPP'91: Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113, Apr. 1991.

[11] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In *OOPSLA'99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 223–237, 1999.

[12] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Nov. 2001.

[13] A. Roy, S. Hand, and T. Harris. A runtime system for software lock elision. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 261–274, Apr. 2009.

[14] K. Russel and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk locking. In *OOPSLA'06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 263–272, 2006.

[15] *SPECjbb2005*. `http://www.spec.org/jbb2005/`.

[16] J. Wetzel, E. Siha, C. May, B. Frey, J. Furukawa, and G. Fraizier. *BookII: PowerPC Virtual Environment Architecture*.

[17] P. Wu, M. M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. F. Mergen, X. Shen, M. F. Spear, H. Wang, and K. Wang. Compiler and runtime techniques for software transactional memory optimization. *Concurrency and Computation: Practice and Experience*, 21(1):7–23, Jan. 2009.

[18] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for java. In *ECOOP'08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 129–154, July 2008.