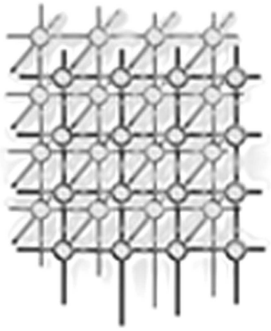
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **00**:1–41 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Revocation techniques for Java concurrency



Adam Welc[†], Suresh Jagannathan[‡], Antony L. Hosking[§]

Department of Computer Sciences
Purdue University
250 N. University Street
West Lafayette, IN 47907-2066, U.S.A.

SUMMARY

This paper proposes two approaches to managing concurrency in Java using a *guarded region* abstraction. Both approaches use *revocation* of such regions – the ability to *undo* their effects automatically and transparently. These new techniques alleviate many of the constraints that inhibit construction of transparently scalable and robust concurrent applications. The first solution, *revocable monitors*, augments existing mutual exclusion monitors with the ability to resolve priority inversion and deadlock dynamically, by reverting program execution to a consistent state when such situations are detected, while preserving Java semantics. The second technique, *transactional monitors*, extends the functionality of revocable monitors by implementing guarded regions as lightweight transactions that can be executed concurrently (or in parallel on multiprocessor platforms). The presentation includes discussion of design and implementation issues for both schemes, as well as a detailed performance study to compare their behavior with the traditional, state-of-the-art implementation of Java monitors based on mutual exclusion.

KEY WORDS: isolation, atomicity, concurrency, synchronization, Java, speculation

1. Introduction

Managing complexity is a major challenge in constructing robust large-scale server applications (such as database management systems, application servers, airline reservation systems, *etc*). In a typical environment, large numbers of clients may access a server application concurrently. To provide satisfactory response time and throughput, applications are often made concurrent. Thus, many programming languages (*eg*, Smalltalk, C++, ML, Modula-3, Java) provide mechanisms that enable concurrent programming via a thread abstraction, with threads being the smallest unit of concurrent

[†]E-mail: welc@cs.purdue.edu
[‡]E-mail: suresh@cs.purdue.edu
[§]E-mail: hosking@cs.purdue.edu
Contract/grant sponsor: National Science Foundation; contract/grant number: IIS-9988637, CCR-0085792, STI-0034141

execution. Another key mechanism offered by these languages is the notion of *guarded* code regions in which accesses to shared data performed by one thread are *isolated* from accesses performed by other threads, and all updates performed by a thread within a guarded region become visible to the other threads *atomically*, once the executing thread exits the region. Guarded regions (*eg*, Java synchronized methods and blocks, Modula-3 `LOCK` statements) are usually implemented using mutual-exclusion locks.

In this paper, we explore two new approaches to concurrent programming, comparing their performance against use of a state-of-the-art mutual exclusion implementation that uses *thin locks* to minimize the overhead of locking [4]. Our discussion is grounded in the context of the Java programming language, but is applicable to any language that offers the following mechanisms:

- Multithreading: concurrent threads of control executing over objects in a shared address space.
- Synchronized blocks: lexically-delimited blocks of code, guarded by dynamically-scoped monitors (locks). Threads synchronize on a given monitor, acquiring it on entry to the block and releasing it on exit. Only one thread may be perceived to execute within a synchronized block at any time, ensuring exclusive access to all monitor-protected blocks.
- Exception scopes: blocks of code in which an error condition can change the normal flow of control of the active thread, by exiting active scopes, and transferring control to a handler associated with each block.

Difficulties arising in the use of mutual exclusion locking with multiple threads are widely-recognized, such as *race conditions*, *priority inversion* and *deadlock*.

Race conditions are a serious issue for non-trivial concurrent programs. A race exists when two threads can access the same object, and one of the accesses is a write. To avoid races, programmers must carefully construct their application to trade off performance and throughput (by maximizing concurrent access to shared data) for correctness (by limiting concurrent access when it could lead to incorrect behavior), or rely on race detector tools that identify when races occur [7, 8, 18]. Recent work has advocated higher-level safety properties such as atomicity for concurrent applications [19].

In languages with priority scheduling of threads, a low-priority thread may hold a lock even while other threads, which may have higher priority, are waiting to acquire it. *Priority inversion* results when a low-priority thread T_l holds a lock required by some high-priority thread T_h, forcing the high-priority T_h to wait until T_l releases the lock. Even worse, an unbounded number of runnable *medium*-priority threads $\mathcal{T}_m$ may exist, thus preventing T_l from running, making unbounded the time that T_l (and hence T_h) must wait. Such situations can cause havoc in applications where high-priority threads demand some level of guaranteed throughput.

Deadlock results when two or more threads are unable to proceed because each is waiting on a lock held by another. Such a situation is easily constructed for two threads, T_1 and T_2: T_1 first acquires lock L_1 while T_2 acquires L_2, then T_1 tries to acquire L_2 while T_2 tries to acquire L_1, resulting in deadlock. Deadlocks may also result from a far more complex interaction among multiple threads and may stay undetected until and beyond application deployment. The ability to resolve a deadlock dynamically is much more attractive than permanently stalling some subset of concurrent threads.

For real-world concurrent programs with complex module and dependency structures, it is difficult to perform an exhaustive exploration of the space of possible interleavings to determine statically when races, deadlocks, or priority inversions may arise. For such applications, the ability to redress undesirable interactions transparently among scheduling decisions and lock management is very useful.

These observations inspire the first solution we propose: *revocable monitors*. Our technique augments existing mutual exclusion monitors with the ability to resolve priority inversion dynamically (and automatically). Some instances of deadlock may be resolved by revocation. However, we note that deadlocks inherent to a program that are independent of scheduling decisions will manifest themselves as *livelock* when revocation is used.

A second difficulty with using mutual exclusion to mediate data accesses among threads is ensuring adequate performance when running on multi-processor platforms. To manipulate a complex shared data structure like a tree or heap, applications must either impose a global locking scheme on the roots, or employ locks at lower-level nodes in the structure. The former strategy is simple, but reduces realizable concurrency and may induce false exclusion: threads wishing to access a distinct piece of the structure may nonetheless block while waiting for another thread that is accessing an unrelated piece of the structure. The latter approach permits multiple threads to access the structure simultaneously, but incurs implementation complexity, and requires more memory to hold the necessary lock state.

Our solution to this problem is an alternative to lock-based mutual exclusion: *transactional monitors*. These extend the functionality of revocable monitors by implementing guarded regions as lightweight transactions that can be executed concurrently (or in parallel on multiprocessor platforms). Transactional monitors define the following data visibility property that preserves isolation and atomicity invariants on shared data protected by the monitor: all updates to objects guarded by a transactional monitor become visible to other threads only on successful completion of the monitor transaction.* Because transactional monitors impose serializability invariants on the regions they protect (*ie*, preserve the appearance of serial execution), they can help reduce race conditions by allowing programmers to more aggressively guard code regions that may access shared data *without* paying a significant performance penalty. Since the system dynamically records and redresses state violations (by revoking the effects of the transaction when a serializability violation is detected), programmers are relieved from the burden of having to determine when mutual exclusion can safely be relaxed. Thus, programmers can afford to over-specify code regions that must be guarded, provided the implementation can relax such over-specification safely and efficiently whenever possible.

While revocable monitors and transactional monitors rely on similar mechanisms, and can exist side-by-side in the same virtual machine, their semantics and intended utility are quite different. We expect revocable monitors to be used primarily to resolve deadlock as well as to improve throughput for high-priority threads by transparently averting priority inversion. In contrast, we envision transactional monitors as an entirely new synchronization framework that addresses the performance impact of classical mutual exclusion while simplifying concurrent programming.

We examine the performance and scalability of these different approaches in the context of a state-of-the-art Java compiler and virtual machine, namely the Jikes Research Virtual Machine (RVM) [3] from IBM. Jikes RVM is an ideal platform to compare our solutions with pure lock-based mutual exclusion, since it already uses sophisticated strategies to minimize the overhead of traditional mutual-exclusion locks [4]. A detailed evaluation in this context provides an accurate depiction of the tradeoffs embodied and benefits obtained using the solutions we propose.

*A slightly weaker visibility property is present in Java for updates performed within a synchronized block (or method); these are *guaranteed* to be visible to other threads only upon exit from the block.

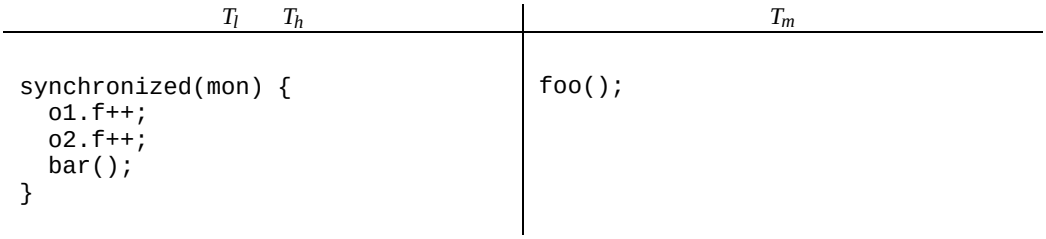


Figure 1. Priority inversion

2. Revocable monitors: Overview

There are several ways to remedy erroneous or undesirable behavior in concurrent programs. Static techniques can sometimes identify erroneous conditions, allowing programmers to restructure their application appropriately. When static techniques are infeasible, dynamic techniques can be used both to identify problems and remedy them when possible. Solutions to priority inversion such as the *priority ceiling* and *priority inheritance* protocols [40] are good examples of such dynamic solutions.

Priority ceiling and priority inheritance solve an *unbounded priority inversion* problem, illustrated using the code fragment in Figure 1 (both T_l and T_h execute the same code and methods `foo()` and `bar()` contain an arbitrary sequence of operations). Let us assume that thread T_l (low priority) is first to acquire the monitor `mon`, modifies objects o_1 and o_2, and is then preempted by thread T_m (medium priority). Note that thread T_h (high priority) is not permitted to enter monitor `mon` until it has been released by T_l, but since method `foo()` executed by T_m may contain arbitrary sequence of actions (*eg*, synchronous communication with another thread), it may take arbitrary time before T_l is allowed to run again (and exit the monitor). Thus thread T_h may be forced to wait for an unbounded amount of time before it is allowed to complete its actions.

The priority ceiling technique raises the priority of any locking thread to the highest priority of any thread that ever uses that lock (*ie*, its priority ceiling). This requires the programmer to supply the priority ceiling for each lock used throughout the execution of a program. In contrast, priority inheritance will raise the priority of a thread only when holding a lock causes it to block a higher priority thread. When this happens, the low priority thread inherits the priority of the higher priority thread it is blocking. Both of these solutions prevent a medium priority thread from blocking the execution of the low priority thread (and thus also the high priority thread) indefinitely. However, even in the absence of the medium priority thread, the high priority thread is forced to wait until the low priority thread releases its lock. In the example given, the time to execute method `bar()` is potentially unbounded, thus high priority thread T_h may still be delayed indefinitely until low priority thread T_l finishes executing `bar()` and releases the monitor. Neither priority ceiling nor priority inheritance offer a solution to this problem.

Besides priority inversion, deadlock is another potentially unwanted consequence of using mutual-exclusion abstractions. A typical deadlock situation is illustrated with the code fragment in Figure 2. Let us assume the following sequence of actions: thread T_1 acquires monitor `mon1` and updates object

T_1	T_2

`synchronized(mon1) {`
`  o1.f++;`
`  synchronized(mon2) {`
`  bar();`
`  }`
`}`
	`synchronized(mon2) {`
	`  o2.f++;`
	`  synchronized(mon1) {`
	`  bar();`
	`  }`
	`}`

Figure 2. Deadlock

o_1, thread T_2 acquires monitor `mon2` and updates object o_2, thread T_1 attempts to acquire monitor `mon2` (T_1 blocks since `mon2` is already held by thread T_2) and thread T_2 attempts to acquire monitor `mon1` (T_2 blocks as well since `mon1` is already held by T_1). The result is that both threads are deadlocked – they will remain blocked indefinitely and method `bar()` will never get executed by any of the threads.

In both of the scenarios illustrated by Figures 1 and 2, one can identify a single *offending* thread that must be revoked in order to resolve either the priority inversion or the deadlock. For priority inversion the offending thread is the low-priority thread currently executing the monitor. For deadlock, it is either of the threads engaged in deadlock – there exist various techniques for preventing or detecting deadlock [21], but all require that the actions of one of the threads leading to deadlock be revoked.

Revocable monitors can alleviate both these issues. Our approach to revocation combines compiler techniques with run-time detection and resolution. When the need for revocation is encountered, the run-time system selectively revokes the offending thread executing the monitor (*ie*, synchronized block) and its effects. All updates to shared data performed within the monitor are *logged*. Upon detecting priority inversion or deadlock (either at lock acquisition, or in the background), the run-time system interrupts the offending thread, uses the logged updates to undo that thread's shared updates, and transfers control of the thread back to the beginning of the block for retry. Externally, the effect of the roll-back is to make it appear that the offending thread never entered the block.

The process of revoking the effects performed by a low priority thread within a monitor is illustrated in Figure 3 where wavy lines represent threads T_l and T_h, circles represent objects o_1 and o_2, updated objects are marked grey, and the box represents the dynamic scope of a common monitor guarding a synchronized block executed by the threads. This scenario is based on the code from Figure 1 (data access operations performed within method `bar()` have been omitted for brevity). In Figure 3(a) low-priority thread T_l is about to enter the synchronized block, which it does in Figure 3(b), modifying object o_1. High-priority thread T_h tries to acquire the same monitor, but is blocked by low-priority T_l (Figure 3(c)). Here, a priority inheritance approach [40] would raise the priority of thread T_l to that of T_h, but T_h would still have to wait for T_l to release the lock. If a priority ceiling protocol was used, the priority of T_l would be raised to the ceiling upon its entry to the synchronized block, but the problem of T_h being forced to wait for T_l to release the lock would remain. Instead, our approach preempts T_l, undoing any updates to o_1, and transfers control in T_l back to the point of entry to the synchronized block. Here T_l must wait while T_h enters the monitor, and updates objects o_1 (Figure 3(e))

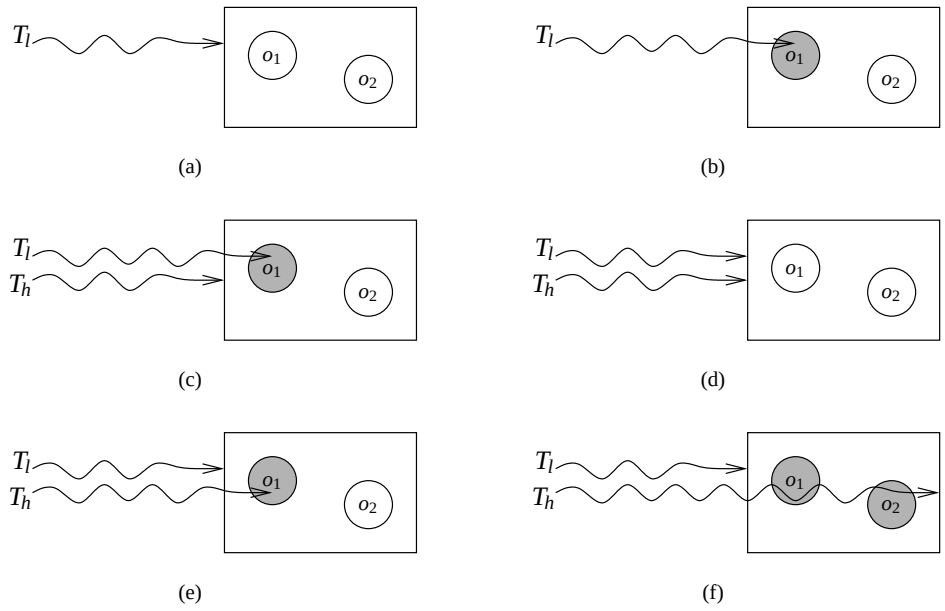


Figure 3. Revoking the effects of a synchronized block's execution – priority inversion

and o_2 (Figure 3(f)), before leaving. At this point the monitor is released and T_l will re-gain entry. This example reveals why roll-backs are useful in dealing with priority inversion issues. Note, however, that the correctness of the solution relies critically on the assumption that threads see updates performed within synchronized blocks only after the lock on the block is released, permitting them entry. If T_h were allowed to see updates to o_1 while T_l still held the lock on the synchronized block, the effect of the roll-back would be moot.

The process of revoking a thread in the case of deadlock is illustrated in Figure 4. The wavy lines represent threads T_1 and T_2, circles represent objects o_1 and o_2, updated objects are marked grey, and the boxes represent the dynamic scopes of monitors `mon1` and `mon2`. This scenario is based on the code from Figure 2. In Figure 4(a) thread T_1 is about to enter monitor `mon1`. In Figure 4(b) T_1 enters `mon1`, updates object o_1 and attempts to enter monitor `mon2`. In Figure 4(c) thread T_2 is about to enter monitor `mon2`. In Figure 4(d) the same thread enters `mon2`, updates object o_2 and attempts to enter monitor `mon1`. We assume that thread T_1 is selected for revocation – its updates to object o_1 are rolled back and its execution of monitor `mon1` retried (Figure 4(e)). Thread T_2 may then enter monitor `mon1`, proceed to execute method `bar()` (data access operations performed within method `bar()` have again been omitted for brevity) and exit both monitor `mon1` and monitor `mon2` (Figure 4(f)).

Some instances of deadlock cannot be resolved using revocation. If deadlock is guaranteed to arise in the way locks have been programmed (independently of scheduling) when using traditional non-

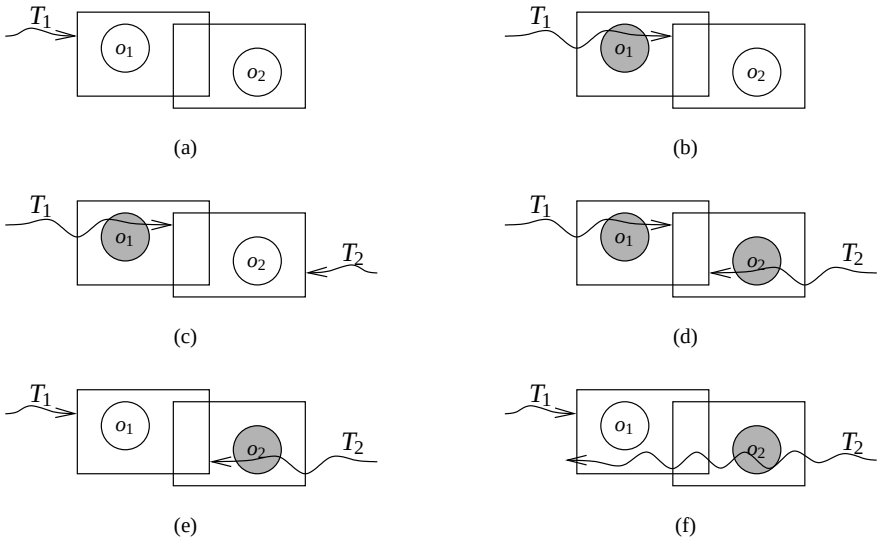


Figure 4. Revoking the effects of a synchronized block's execution – deadlock

T_1	T_2

`synchronized(mon1) {`
`  while (!o1.f) {`
`  synchronized(mon2) {`
`  bar();`
`  }`
`  }`
`  o2.f = true;`
`}`
	`synchronized(mon2) {`
	`  while (!o2.f) {`
	`  synchronized(mon1) {`
	`  bar();`
	`  }`
	`  }`
	`  o1.f = true;`
	`}`

Figure 5. Schedule-independent deadlock

revocable monitors, the deadlock still cannot be resolved by revocable monitors. Consider the code fragment in Figure 5. Because of control-flow dependencies, *all* executions of this program under traditional mutual exclusion will eventually lead to deadlock. When executing this program using revocable monitors, the run-time system will attempt to resolve deadlock by revoking one of the threads. Let's assume that thread T_1 is selected for revocation. However, in order for thread T_2 to make progress it must be able to observe updates performed by thread T_1. Because T_2 is unable to proceed, it will maintain ownership of the monitor(s) it has already acquired, which will eventually lead to another deadlock once execution of thread T_1 is resumed. Note however, that while revocable monitors are unable to assist in resolving schedule-independent deadlocks, the final observable effect of the resulting *livelock* (*ie*, repeated attempts to resolve the deadlock situation via revocation) is the same for deadlock – none of the threads will make progress.

3. Revocable monitors: Design

One of the main principles underlying the design of revocable monitors is a *compliance requirement*: programmers must perceive all programs executing in our system to behave exactly the same as on all other platforms implemented according to the specification of a given language. In order to achieve this goal we must adhere to the execution semantics of the language and follow the memory access rules specified by those semantics.

We fulfill the compliance requirement by *logging* all updates to shared data performed by a thread executing within a monitor. We use the information from the log to *roll back* updates whenever the monitor is revoked. In effect, synchronized sections execute speculatively, and their updates may be revoked at any time before the block is exited.

Our approach is inspired by optimistic concurrency control protocols [29]. Traditionally, optimistic techniques distinguish three execution phases: a *read phase*, a *validation phase* and a *write phase* [29]. In the read phase all updates are redirected to the log, the validation phase verifies the integrity of all data accessed during the entire execution, and the write phase atomically installs all updates into the shared space. However, in the case of revocable monitors data integrity is guaranteed by the presence of mutual exclusion. Thus, updates can be performed in place and the validation phase can be omitted. It is only when a monitor is revoked that the information from the log is used to roll back changes performed by a thread executing that monitor. The space overhead of maintaining logs is not excessive since a log (associated with each thread object) needs to be maintained only when the thread is executing within a revocable monitor, and can be discarded upon exit from the monitor.

The introduction of revocable monitors requires a careful consideration of the interaction between revocation and the Java Memory Model (JMM) [32]. We elaborate on these issues in the following sections.

3.1. The Java memory model (JMM)

The JMM defines a *happens-before* relation (written $\xrightarrow{hb}$) among the actions performed by threads in a given execution of a program. For single-threaded execution the happens-before relation is defined by program order. For multi-threaded execution a happens-before relation is induced between an unlock

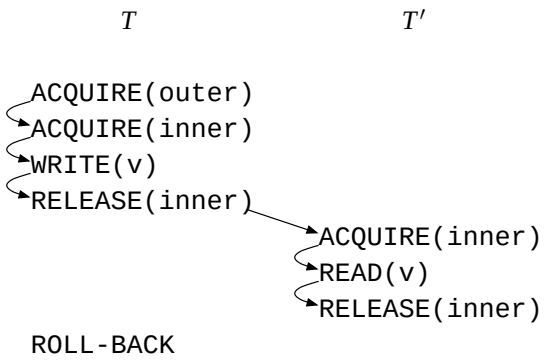


Figure 6. Erroneous revocation sequence due to monitor nesting and Java visibility semantics

$\mathtt{u}_M$ (release) and a subsequent lock $\mathtt{l}_M$ (acquire) operation on a given monitor M ($\mathtt{u}_M \overset{hb}{\rightarrow} \mathtt{l}_M$). The happens-before relation is transitive: $\mathtt{x} \overset{hb}{\rightarrow} \mathtt{y}$ and $\mathtt{y} \overset{hb}{\rightarrow} \mathtt{z}$ imply $\mathtt{x} \overset{hb}{\rightarrow} \mathtt{z}$. The JMM shared data visibility rule is defined using the happens-before relation: a read $\mathtt{r}_v$ is *allowed* to observe a write $\mathtt{w}_v$ to a given variable variable v if $\mathtt{r}_v$ does not happen before $\mathtt{w}_v$ and there is no intervening write $\mathtt{w}'_v$ such that $\mathtt{r}_v \overset{hb}{\rightarrow} \mathtt{w}'_v \overset{hb}{\rightarrow} \mathtt{w}_v$ (we say that a read becomes *read-write dependent* on the write that it is *allowed* to see). As a consequence, it is possible that partial results computed by some thread T executing within monitor M become visible to (and are used by) another thread T' even before thread T releases M if accesses to those updated objects performed by T' are not mediated by first acquiring M. However, a subsequent revocation of monitor M would undo the update and remove the happens-before relation, making a value seen by T' appear "out of thin air" and thus the execution of T' inconsistent with the JMM.

An example of such an execution appears in Figure 6: thread T acquires monitor `outer` and subsequently monitor `inner`, writes to a shared variable `v` and releases monitor `inner`. Then thread T' acquires monitor `inner`, reads variable `v` and releases monitor `inner`. The execution is JMM-consistent up to the roll-back point: the read performed by T' is *allowed* but the subsequent roll-back of T would violate consistency.

A similar problem occurs when *volatile* variables are used. The Java Language Specification (JLS) [20] states that updates to volatile variables immediately become visible to all program threads. Thus, there also exists a happens-before relation between a volatile write and all subsequent volatile reads of the same (volatile) variable. For the execution presented in Figure 7 `vol` is a volatile variable and edges depict a happens-before relation. As in the previous example, the execution is JMM-consistent up to the roll-back point because a read performed by T' is *allowed*, but the roll-back would violate consistency. We now discuss possible solutions to these JMM-consistency preservation problems.

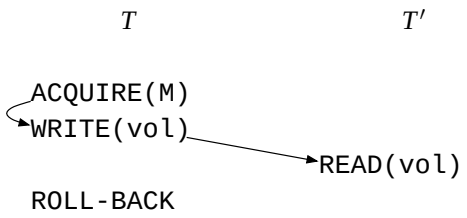


Figure 7. Erroneous revocation sequence due to volatile variable access

```
static boolean v=false;
```

T	T'

`synchronized(outer) {`
`  synchronized(inner) {`
`  v=true;`
`  }`
 	
`  // ROLL-BACK`
`}`
	`while (true) {`
	`  synchronized(inner) {`
	`  if (v) break;`
	`  }`
	`}`

Figure 8. Rescheduling thread execution in the presence of rollback may not always be correct

3.2. Preserving JMM-consistency

Several solutions to the problem of partial results of a monitored computation being exposed to other threads can be considered. We might trace read-write dependencies among all threads and upon roll-back of a monitor trigger a cascade of roll-backs for threads whose read-write dependencies are violated. An obvious disadvantage of this approach is the need to consider *all* operations (including non-monitored ones) for a potential roll-back. In the execution of Figure 7 the volatile read performed by T' would have to be rolled back even though it is not guarded by any monitor. Furthermore, to apply this solution, the full execution context of each thread (*ie*, its instruction pointer, registers, thread stack *etc*) would have to be logged in addition to its shared data operations. Consider a situation based on the example of Figure 6 where thread T' *returns* (from the current method) after releasing monitor `inner` but before thread T is asked to roll back the execution of monitor `outer`. Without the ability to restore the full execution context of T', the subsequent roll-back of monitor `inner` by that thread becomes infeasible.

Another possible solution is to re-schedule the execution of threads in problematic cases. In the examples of Figures 6 and 7, if thread T' executes fully before thread T, the execution will still be JMM-consistent. The roll-back of T does not violate consistency since none of the updates performed by T are visible to T'. Besides the obvious question about the practicality of re-scheduling as a solution

(some knowledge about the future actions performed by threads would be required), there also remains the issue of correctness. While re-scheduling may be correct in some cases, it is not necessarily correct in others. Consider the Java program of Figure 8. Completion of thread T' is dependent upon it seeing the effect of T executing the statement `v=true`. If we choose to reschedule T' to run before T, knowing that T will be revoked, then T' will never complete. Of course, if we make the "right" choice to reschedule T' after T, things will work. There are however, similar cases where rescheduling never works.

The solution that does seem flexible enough to handle all possible problematic cases, and simple enough to avoid using complex analyses and/or maintaining significant additional meta-data, is to disable the revocability of monitors whose roll-back could create inconsistencies with respect to the JMM. As a consequence, not all instances of priority inversion can be resolved. We mark a monitor M *non-revocable* when a read-write dependency is created between a write it has performed within M^* and a read performed by another thread. Detecting the possibility for this is relatively straightforward, without needing to track every read, so long as we track monitor acquire/release dependencies. This can be achieved as follows. When a thread holding an outer monitor enters some inner monitor, it becomes *associated* with the inner monitor. This association is cleared when the thread exits the outer monitor, or when the thread is made non-revocable, as follows. Any other thread arriving at the monitor will simply make non-revocable any thread associated with that monitor, clearing the association. If the arriving thread itself holds an outer monitor then it now becomes associated with the monitor. We believe this solution does not severely penalize the effectiveness of our technique. Intuitively, programmers guard accesses to the same subset of shared data using the same set of monitors; in such cases, there is no need to force non-revocability of any of the monitors (even if they are nested) since mutual-exclusion induced by monitor acquisition prevents generation of problematic dependencies among these threads.

There are other Java constructs that affect revocability of the monitors. Calling a native method within a monitor also forces non-revocability of the monitor (and all of its enclosing monitors if it is nested), since the effects of a native method cannot generally be revoked (*eg*, printing a message to the console is irrevocable, even if benign). The same applies to executions where a `wait` method is invoked within a nested monitor.[†] Revocation of the `wait` call would result in a situation where the matching `notify` call (that "woke up" the waiting thread) "disappears" (*ie*, does not get delivered to any thread) which would violate Java execution semantics. A call to `notify` does not force irrevocability of enclosing monitors: Java VM implementations are permitted [32] to perform "spurious wake-ups" so a rolled back notification can be considered as such.

4. Revocable monitors: Implementation

To demonstrate the validity of our approach, we base our implementation on a well-known Java execution environment with a high-quality compiler. We use IBM's Jikes RVM [3], a state-of-the-art

[*] The write may additionally be guarded by other monitors nested within M.

[†] A monitor object associated with the receiver object is released upon a call to `wait` and re-acquired after returning from the call. In the case of a non-nested monitor a potential roll-back will therefore not reach beyond the point when `wait` was called.

research virtual machine (VM) for Java with performance comparable to many production VMs. Java bytecodes in Jikes RVM are compiled directly to machine code using either a low-cost non-optimizing "baseline" compiler or an aggressive optimizing compiler.

When discussing the details of our approach, we concentrate on the necessary compiler and run-time capabilities that allow the VM to interrupt execution of synchronized blocks (monitors) at arbitrary points without inducing any observable effects on an application's execution behavior. For our case study we chose the priority inversion problem, rather than deadlock resolution, as an excellent vehicle to measure the trade-offs inherent in speculative execution.

4.1. Monitor roll-back

Our implementation uses bytecode rewriting* to save program state (values of local variables and method parameters) for re-execution and the existing exception mechanisms to return control to the beginning of the synchronized block. We modify the compiler and run-time system to suppress generation (and invocation) of undesirable exception handlers during a roll-back operation, to insert access "barriers"† for logging and to revert updates performed up to revocation of a synchronized block.

4.1.1. Bytecode transformation

There exist two different synchronization constructs in Java: synchronized methods and synchronized blocks. We treat them uniformly, by transforming synchronized methods into non-synchronized equivalents whose entire body is enclosed in a synchronized block. For each synchronized method we create a non-synchronized wrapper with a signature identical to the original method. We fill the body of the wrapper method with a synchronized block enclosing an invocation of the original (non-synchronized) method, which has been appropriately renamed to avoid name clashes. We also instruct the VM to inline the original method within the wrapper to avoid performance penalties related to the delegating method invocation. This approach greatly simplifies our implementation,‡ is extremely simple and robust, and also efficient because of inlining.

Each synchronized block (bracketed at the bytecode level by *monitorenter* and *monitorexit* operations) is wrapped within an exception scope that catches a special `Rollback` exception. The roll-back exception is thrown internally by the VM (see below), but the code to catch it is injected into the bytecode. Since a roll-back may involve a nested synchronized block, each roll-back exception catch handler invokes an internal VM method to check if it corresponds to the synchronized block that is to be re-executed. If it does, then the handler releases the lock associated with its synchronized block, and returns control to the beginning of the block. Otherwise, the handler re-throws the `Rollback` exception to the enclosing synchronized block.

*We use the Bytecode Engineering Library (BCEL) from Apache for this purpose. Note that our solution does not preclude the use of languages that do not have a similar intermediate representation – we could use source-code rewriting instead.

†Code snippets inserted by the compiler into the code stream and directly preceding (or substituting) the implementation of actual data access operations (*ie*, loads and stores).

‡We need only handle explicit *monitorenter* and *monitorexit* bytecodes, without worrying about implicit monitor operations for synchronized methods.

There is an additional complication related to the return of control to the beginning of the block. The contents of the VM's operand stack before executing a *monitorenter* operation must be the same as at first invocation in subsequent re-invocations resulting from revocation. However, in accordance with the Java virtual machine specification [30], the run-time system erases the operand stack of the method activation that will catch the exception. To handle this, we inject bytecode to save the values on the operand stack just before each roll-back scope's *monitorenter* opcode, and to restore the stack state in the handler before transferring control back to the *monitorenter*.

4.1.2. Compiler and run-time modifications

The roll-back operation is initiated by throwing a `Rollback` exception. However, we cannot rely on the standard exception handling mechanism to propagate the `Rollback` exception up the activation stack to the synchronized block being revoked, since it will also run "default" exception handlers in nested exception scopes as it unwinds the stack. Such "default" handlers include both `finally` blocks, and `catch` blocks for exceptions of type `Throwable`, of which all exceptions (including `Rollback`) are instances. Running these intervening handlers would violate our semantics that an aborted synchronized block produces no side-effects.

To handle this, we augment exception handling to ignore all handlers (including `finally` blocks) that do not explicitly catch the `Rollback` exception, when it is thrown. The default behavior still applies for all other exceptions, to preserve the standard semantics. We are careful to release monitors as necessary wherever the Jikes RVM optimizing compiler releases them explicitly in its implementation of synchronized blocks.

Roll-back relies on information collected within the compiler inserted write barriers. Both compilers (baseline and optimizing) have been modified to inject barriers before every store operation (represented by the bytecodes: `putfield` for object stores, `putstatic` for static variable stores, and *X*`astore` for array stores). The barrier records in the log every modification performed by a thread executing a synchronized block. We implemented the log as a sequential buffer. For object and array stores, three values are recorded: the target object or array, the offset of the modified field or array slot, and the previous (old) value in that field/slot. For stores to static variable two values are recorded: the offset of the static variable in the global symbol table and the previous value of that variable. Upon monitor revocation information stored in the log is used to undo updates to shared data performed by the thread executing this monitor.

4.1.3. Discussion

Instead of using bytecode transformations, we note that an alternative strategy might be to implement re-execution entirely at the VM level (*ie*, all the code modifications necessary to support roll-backs would only involve the compiler and the Java run-time system). This approach simply requires that the current state of the thread (*ie*, contents of local variables, non-volatile registers, stack pointer, *etc*) be remembered upon entry to a synchronized block, and restored when a roll-back is required. Unfortunately, this strategy has the significant drawback that it introduces implicit control-flow edges in the program's control-flow graph that are not visible to the compiler. Consequently, liveness information necessary for the garbage collector may be computed incorrectly, since a roll-back action

may require stack slots to remain live that would ordinarily be marked dead. Resolving these issues would entail substantial changes to the compiler and run-time system.

A second alternative we considered (and discarded) was a fully portable user-level implementation that would not require any modifications to the VM or the compiler. Instead, this solution would take advantage of language-level exceptions and use bytecode rewriting techniques exclusively to provide all the support necessary to perform a roll-back operation. Unfortunately, in the absence of any compiler modifications, the built-in exception handling mechanism may execute an arbitrary number of other user-defined exception handlers and finalizers, violating the transparency of our design. Moreover, inserting write-barriers at the bytecode level to log changes would require optimizations to remove them to be re-implemented at bytecode level as well. For example an escape analysis could be used to eliminate barriers for accesses to thread-local objects. However, if implemented inside the compiler, barriers inserted at bytecode level appear as snippets of ordinary code, so the compiler cannot optimize them effectively.

4.2. Priority inversion avoidance

Detecting priority inversion is reasonably simple. A thread acquiring a monitor deposits its priority in the header of the monitor object. Before another thread can acquire the monitor, the scheduler checks whether its own priority is higher than the priority of the thread currently executing within the synchronized block. If it is, then the it triggers roll-back of the low priority thread. After the low-priority thread rolls back its changes and releases the monitor, the high-priority thread acquires control of the synchronized block. If the incoming thread's priority is lower, it blocks on the monitor and waits for the other thread to complete execution of the synchronized block.

The Jikes RVM does not include a priority scheduler; threads are scheduled in a round-robin order. This does not affect the generality of our solution nor does it invalidate the results obtained, since the problems solved by our mechanisms cannot be solved simply by using a priority scheduler. However, in order to make the measurements independent of the random order in which threads arrive at a monitor, we augmented the monitor queues to take priority into account. A thread can have either high or low priority. When a thread releases a monitor, another thread is scheduled from the queue. If it is a high-priority thread, it is allowed to acquire the monitor. If it is a low-priority thread, it is allowed to run only if there are no other high-priority threads waiting in the queue.

5. Revocable monitors: Experiments

We quantify the overhead of the revocable monitors mechanism using a detailed micro-benchmark. We measure executions that exhibit priority inversion to verify if the increased overheads induced by our implementation are mitigated by higher overall throughput of high-priority threads. The experiments are performed for a uni-processor system, since revocable monitors do nothing to increase concurrency in applications, so applications will exhibit no more parallelism using revocable monitors on multi-processors than they would using non-revocable monitors. In our results, it is to be expected that revocable monitors used to address priority inversion will sacrifice throughput of low-priority threads to improve throughput of high-priority threads. As a result, total throughput will suffer. Our results

quantify this sacrifice of total throughput to be approximately 30%, while throughput for high-priority threads improves by 25% to 100%.

5.1. Benchmark program

The micro-benchmark executes several low and high-priority threads contending on the same lock. Regardless of their priority, all threads are compiled identically, with write barriers inserted to log updates, and special exception handlers injected to restart synchronized blocks. Though our benchmark is structured so that only low-priority threads will actually be revoked, updates of both low-priority and high-priority threads are logged for fairness, even though high-priority threads are never rolled back. Every thread executes 100 synchronized synchronized blocks. Each synchronized block contains an inner loop containing an interleaved sequence of read and write operations. We emphasize that our micro-benchmark has been constructed to gauge the overheads inherent in our techniques (the costs of re-execution, logging, *etc*) and not necessarily to simulate any particular real-life application. We do not bias the benchmark structure in favor of our mechanisms by artificially extending the execution time using benign (with respect to logging) operations (*eg*, method calls). Therefore, we decided to make the execution time of a synchronized block directly proportional to the number of shared data operations performed within that block. We fixed the number of iterations of the inner loop for low-priority threads at 500K, and varied it for the high-priority threads (100K and 500K). The remaining parameters for our benchmark include:

- The ratio of high-priority threads to low-priority threads – we used three configurations: $2+8$, $5+5$, and $8+2$, high-priority plus low-priority threads, respectively.
- The ratio of write to read operations performed within a synchronized block – we used six different configurations ranging from 0% writes (*ie*, 100% reads) to 100% writes (*ie*, 0% reads).

Our benchmark also includes a short random pause time (on average approximately a single thread quantum in Jikes RVM) right before an entry to the synchronized block, to ensure random arrival of threads at the monitors guarding the blocks.

Our thesis is that the total elapsed time of high-priority threads can be improved using the roll-back scheme, at the expense of longer elapsed time for low-priority threads. Improvement is measured against a priority scheduling implementation that provides no remedy for priority inversion. Thus, for every run of the micro-benchmark we compare the total time it takes for all high-priority threads to complete their execution for the following two settings:

- An *unmodified* VM that does not allow execution of a synchronized block to be interrupted and revoked: when a high-priority thread wants to acquire a lock already held by a low-priority thread, it waits until the low-priority thread exits the synchronized block. The benchmark code executed on this VM is compiled using the Jikes RVM optimizing compiler without any modification.
- A *modified VM* equipped with the compiler and run-time changes to interrupt and revoke execution of synchronized blocks by low-priority threads: when a high-priority thread wants to acquire a lock held by a low-priority thread it signals its intent, resulting in the low-priority thread exiting the synchronized block at the next yield point, rolling back any changes to shared data made from the time it began executing inside the block. The benchmark code executed on

this VM is compiled using the modified version of the Jikes RVM optimizing compiler described in Section 4.

To measure the total elapsed time of high-priority threads we take two time-stamps for each high-priority thread: one when it begins its `run()` method and one at the end of its `run()` method. We compute the total elapsed time for all high-priority threads by subtracting the latest end time-stamp of all high-priority threads from the earliest begin time-stamp of all the high-priority threads. We also record the impact that our solution has on the overall elapsed time of the entire micro-benchmark, including low-priority elapsed times: this is simply the difference between the end time-stamp of the last thread to finish and the begin time-stamp of the first thread to start, regardless of priority.

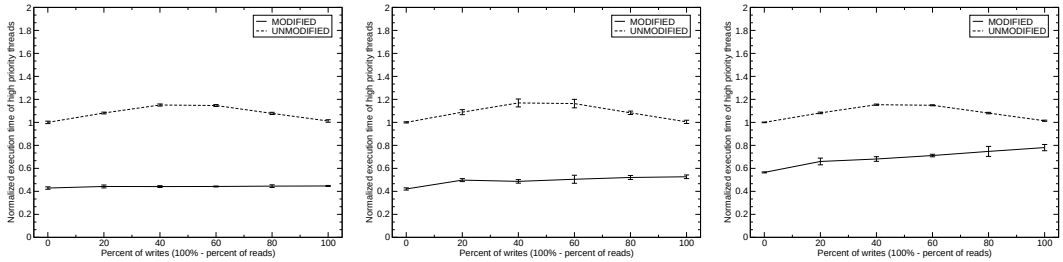
The measurements were taken on an 800MHz Intel Pentium III (Coppermine) with 1GB of RAM running Linux kernel version 2.4.20-13.7 (RedHat 7.0) in single-user mode. A benchmark run consists of one invocation of the VM in which the benchmark is repeated six times. We discard the results of the first iteration, in which the benchmark classes are loaded and compiled, to eliminate the overheads of compilation. We report the average elapsed time for the five subsequent iterations, and show 90% confidence intervals in our results. Our system is based on Jikes RVM 2.2.1 and we use a configuration where both the Jikes RVM (which is itself implemented and bootstrapped in Java) and dynamically loaded classes are compiled using the optimizing compiler by default. Even in this configuration there remain some methods (*eg*, class initializers) that override this setting and are compiled without optimization.

5.2. Results

Figures 9 and 10 plot elapsed times for high priority threads executed on both the modified VM (indicated by a solid line) and unmodified VM (indicated by a dotted line), normalized with respect to the configuration executing 100% reads on an unmodified VM using standard non-revocable monitors. We normalize with respect to the 100% reads benchmark configuration so as to obtain a standard baseline for illustrating performance trends as the read/write mix changes. In Figure 9 every high priority thread executes 100K internal iterations; in Figure 10 the iteration count is 500K. In each figure: the graph labeled (a) reflects a workload consisting of two high-priority threads, and eight low-priority threads; the graph labeled (b) reflects a workload consisting of five high-priority and five low-priority threads; and, the graph labeled (c) reflects a workload consisting of eight high-priority threads and two low-priority ones.

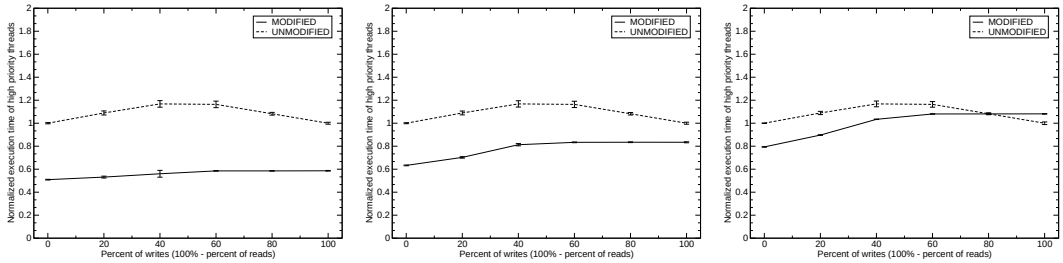
If the ratio of high-priority threads to low-priority threads is relatively low (Figures 9-10 (a)(b)), our hybrid implementation improves throughput for high-priority threads by 25% to 100% over the unmodified implementation. Average elapsed-time gain across all the configurations, including those where the number of high-priority threads is greater than the number of low-priority threads, is 78%. If we discard the configuration where there are eight high-priority threads competing with only two low-priority ones, the average elapsed time of a high-priority thread is half that of the reference implementation.

Note that the influence of different read-write ratios on overall performance is small; recall that all threads, regardless of their priority, log all updates within a synchronized block. This implies that the cost of operations related to log maintenance and roll-back of partial results is also small, compared to the elapsed time of the entire benchmark. Indeed, the actual “workload” (the contents



(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 9. Total time for high-priority threads, 100K iterations



(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

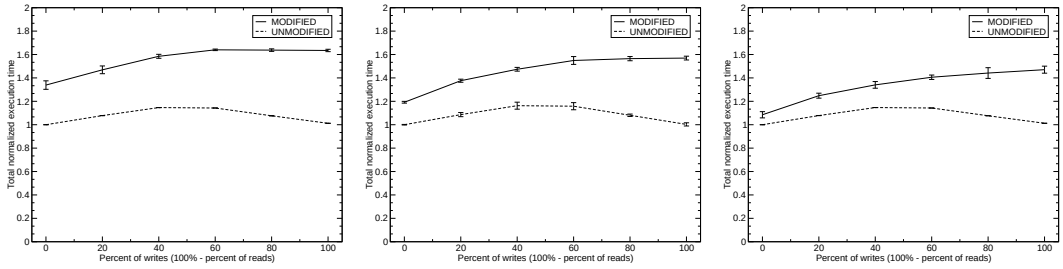
Figure 10. Total time for high-priority threads, 500K iterations

of the synchronized block) in the benchmark consists entirely of data access operations – no delays (method calls, empty loops, *etc*) are inserted in order to artificially extend its execution time. Since realistic programs are likely to have a more diverse mix of operations, the overheads would be even smaller in practice.

As expected, if the number of write operations within a synchronized block is sufficiently large, the overhead of logging and roll-backs may start outweighing potential benefit. For example, in Figure 10(c), under a 100% write configuration, every high priority thread writes, and thus logs, approximately 500K words of data in every execution of a synchronized block. We believe that synchronized blocks that consist entirely of write operations of this magnitude are relatively rare.

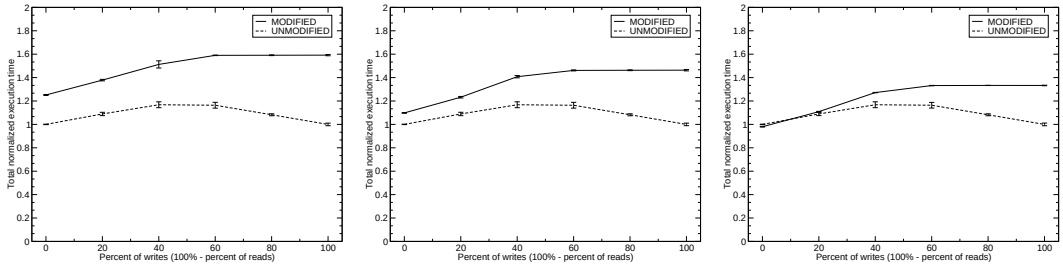
As the ratio of high-priority threads to low-priority threads increases, the benefit of our strategy diminishes (see Figures 9(c) and 10(c)). This is expected: since there are relatively fewer low-priority threads in the system, there is less opportunity to "steal" cycles from them to improve throughput of higher priority ones. We note, however, that even when the roll-back-enabled VM has weaker performance than the unmodified implementation, the average difference in execution time is only a few percent.

Figures 11 and 12 plot overall elapsed times for the entire application executed on both modified (solid line) and unmodified (dotted line) VMs. These graphs are also normalized with respect to a configuration executing 100% reads on the unmodified VM. Note that the overall elapsed time for the modified VM must always be longer than for the unmodified VM. If we disallowed revocability



(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 11. Overall time, 100K iterations



(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 12. Overall time, 500K iterations

of synchronized blocks, threads executing on both VMs would need exactly the same amount of time to execute their workloads (modulo costs related to the implementation of our mechanisms for the modified VM such as barriers, log maintenance, *etc*). However, if the execution of synchronized blocks can be interrupted and revoked, low-priority threads executing on the modified VM will re-execute parts of their synchronized blocks, thus lengthening overall elapsed time. Since our focus is on lowering elapsed times for high priority threads, we consider the impact on overall elapsed time (on average 30% higher on the modified VM) to be acceptable. If our mechanism is used to resolve deadlocks then these overheads may be an even more acceptable price to pay to obtain progress by breaking deadlocks.

6. Transactional monitors: Overview

Revocable monitors solve one class of problems related to writing concurrent programs, but because of the compliance requirement with respect to Java's execution semantics and memory model they are still required to use mutual exclusion as an underlying synchronization mechanism. As a result, the degree of concurrency achievable in the concurrent (parallel) setting is still severely limited. Transactional monitors are an attempt to ease that restriction.

T	T'

`monitored(account_monitor) {`
`  b1 = checking.getBalance();`
`  b2 = savings.getBalance();`
`  print(b1 + b2);`
`}`
	`monitored(account_monitor) {`
	`  checking.withdraw(amount);`
	`  savings.deposit(amount);`
	`}`

Figure 13. Bank account example

Unlike Java monitors implemented using mutual exclusion locks, which make threads acquire a given monitor serially, transactional monitors require only that threads *appear* to acquire the monitor serially. Transactional monitors permit concurrent execution within the monitor so long as the effects of the resulting schedule are *serializable*: the effects of concurrent execution of the monitor are equivalent to *some* legal serial schedule that would arise if no interleaving of the actions of different threads occurred within the guarded region. The executions are equivalent if they produce the same observable behavior; that is, all threads at any point during their execution observe the same state of the shared data. Thus, while transactional monitors and mutual-exclusion monitors have very similar execution semantics, transactional monitors permit a higher degree of concurrency.

Consider the code sample shown in Figure 13. Thread T computes the total balance of both checking and savings accounts. Thread T' transfers money between these accounts. Both account operations (balance and transfer) are guarded by the same `account_monitor` – the code region guarded by the monitor is delimited by curly braces following the `monitored` statement. If the account operations were unguarded then concurrent execution of these operations could potentially yield an incorrect result: the total balance computed after the withdrawal but before the deposit would not include the amount withdrawn from the checking account. If `account_monitor` were a traditional mutual-exclusion monitor, either thread T or T' would win a race to acquire the monitor and would execute fully before releasing the monitor; regardless of the order in which they execute, the total balance computed by thread T would be correct (it would in fact be the same in both cases).

If `account_monitor` is a transactional monitor, two scenarios are possible, depending on the interleaving of the statements implementing the account operations. The interleaving presented in Figure 14 results in both threads successfully completing their executions – it preserves serializability since T''s withdrawal from the checking account does not compromise T's read from the savings account. This interleaving is equivalent to a serial execution in which T executes before T'.

The interleaving presented in Figure 15 results in an incorrect execution (with respect to the serializability requirement): thread T reads an inconsistent state. Serializability is enforced by automatically re-executing the guarded region of thread T.

These examples illustrate several issues in formulating an implementation of transactional monitors. Threads executing within a transactional monitor must execute in *isolation*, and their view of shared data on exit from the monitor must be *consistent* with their view upon entry. Isolation and consistency imply that shared state appears unchanged by other threads. A thread executing in a monitor cannot

	T	T'
(1)	checking.getBalance	
(2)		checking.withdraw
(3)	savings.getBalance	
(4)		savings.deposit

Figure 14. Serializable execution

	T	T'
(1)		checking.withdraw
(2)	checking.getBalance	
(3)	savings.getBalance	
(4)		savings.deposit

Figure 15. Non-serializable execution

see the updates to shared state by other threads. Transactional monitor implementations must permit threads to detect state changes that violate isolation and to roll back automatically (and transparently), restarting their execution in response to such violations.

In Figure 15, the execution of thread T is *not* isolated from the execution of thread T' since thread T sees the effects of the withdrawal but does not see the effects of the deposit. Thus, T is obliged to re-execute its operations. In general, a thread may end up being re-executed at any time within a transactional monitor. To ensure that partial results of a computation performed by a thread do not affect the execution of other threads, the execution of any monitored region must be *atomic*: either the effects of all operations performed within the monitor become visible to other threads upon successful commit or they are all discarded upon abort. The semantics of transactional monitors thus comprise the ACI (*atomicity*, *consistency*, and *isolation*) properties of a classical ACID transaction model [21], and their realization may be viewed as adapting optimistic concurrency control protocols [29] to concurrent object-oriented languages.

The properties of transactional monitors described here are enforced only between threads executing within the same monitor; no guarantees are provided for threads executing within different transactional monitors, nor for threads executing outside of any transactional monitor. These properties result in semantics similar to those of Java's mutual-exclusion monitors. Accesses to data shared by different threads are synchronized only if they acquire the same monitor.

7. Transactional monitors: Design

Our approach to managing concurrency in the case of transactional monitors is even closer in spirit to that of traditional optimistic concurrency techniques [29] than in the case of revocable monitors. Transactional monitors maintain serializability by tracking all accesses to shared data performed during the read phase within a thread-specific *log* . When a thread attempts to release a monitor on exit from a guarded region, an attempt is made to *commit* the log. The commit operation has the effect of verifying the consistency of shared data with respect to the information recorded in the log (*ie*, the validate phase), and *atomically* performing all logged operations at once with respect to any other commit operation (*ie*, the write phase). If the shared data changes in such a way as to invalidate the log, the monitored code region is re-executed, and the commit retried. A log is invalidated if committing its changes would violate serializability of actions performed in the monitored region.

An alternative design might consider performing updates in-place (directly on shared data), and reverting them using information from the log upon monitor revocation. However, using this technique can lead to *cascading revocations* – since all threads that have seen updates of a thread being aborted must be revoked as well – which may severely impact overall performance, or require a global per-access locking protocol (*eg*, two-phase locking [21]) to prevent conflicting data accesses by different threads. A significant disadvantage of using per-access locking is a potential for deadlocks to arise between concurrently executing threads.

There are a number of important issues that arise in a formulating a semantics for transactional monitors:

1. *Transparency*: The degree of programmer control and visibility of internal transaction machinery influences the degree of flexibility provided by the abstraction, and the complexity of using it. For example, if a programmer is given control over how shared data accesses are tracked, objects known to be immutable need not be logged when accessed.
2. *Serializability Violation Detection*: A thread executing within a guarded region may try to detect serializability violation whenever a barrier is executed, or may defer detecting such violation until a commit point (*eg*, monitor exit).
3. *Nesting*: Transaction models often allow transactions to nest freely [34], permitting division of any transaction into some number of sub-transactions. In the presence of nesting, a transactional monitor semantics must define rules on visibility of updates made by sub-transactions.

We motivate our design decisions with respect to the issues above. One of the most important principles underlying our design is transparency of the transactional monitors mechanism: an application programmer should not be concerned with how monitors are represented, nor with details of the logging mechanism, abort or commit operations. After marking a region of code at source level as guarded by a given transactional monitor, a programmer can simply rely on the underlying compiler and run-time system to ensure transactional execution of the region (satisfying the properties of atomicity, consistency, and isolation).

The decision about *when* a thread should attempt to detect serializability violations is strongly dependent on the cost of detection and may vary from one implementation of transactional monitors to another. When choosing the most appropriate point for detecting serializability violations, we must consider the trade-off between reducing the overall cost of checking any serializability invariant (once

if performed on exit from the monitor, or potentially multiple times if performed within access barriers), and reducing the amount of computation performed by a thread that may eventually abort.

Modularity principles dictate that our design support nested transactional monitors. A given monitor region may contain a number of child monitors. Because monitors are released from the bottom up, child monitors must always release before their parent. Thus, a child monitor will re-execute (as needed) until it can be (successfully) released. The updates of child monitors are visible upon release only within the scope of their parent (and, upon release of the outermost monitor, are propagated to the shared space). Updates performed by a parent monitor are always visible to the child.

8. Transactional monitors: Implementation

An implementation that directly reflects the concept behind transactional monitors would redirect all shared data accesses performed by a thread within a transactional monitor to a thread-local log. When an object is first accessed, the accessing thread records its current value in the log and refers to it for all subsequent operations. Serializability violations would be detected by traversing the log and comparing values of objects recorded in the log with those of the original. The effectiveness of this scheme depends on a number of different parameters all of which are influenced by the data access patterns that occur within the application:

- expected contention (or concurrency) at monitor entry points;
- the number of shared objects (both read and written) accessed per-thread;
- the percentage of operations that occur within a transactional monitor that are benign with respect to shared data accesses (method calls, local variable computation, type casts *etc*)

Because the generic implementation is not biased toward any of these parameters, it is not clear how effectively it would perform under widely varying application conditions. Therefore, we consider implementations of transactional monitors optimized towards different shared data access patterns, informally described as low-contention and high-contention.

Both optimized implementations must provide a solution to logging, commit, and abort actions. These actions can be broadly classified under the following categories:

1. *Initialization*: When a transactional monitor is entered, actions to initialize logs, *etc*, may have to be taken by threads before they are allowed to enter the monitor.
2. *Read and Write Operations*: Barriers define the actions to be taken when a thread performs a read or write to an object when executing within a transactional monitor.
3. *Conflict Detection*: Conflict detection determines whether the execution of a region guarded by a given monitor is serializable with respect to the concurrent execution of other regions guarded by the same monitor and it is safe to commit changes to shared data made by a thread.
4. *Commitment*: If there are no conflicts, changes to the original objects must be committed atomically; otherwise the guarded region must be re-executed.

Our current implementation does not yet include support for nested transactions. While nesting adds complications, there are no inherent difficulties in supporting them [33]. Chiefly, the optimistic techniques described below require distinct versions (where used) to be maintained for each

transactional monitor, and for those versions to be applied on commit of each (nested) transactional monitor.

The low-contention scheme and the high-contention scheme both use the same mechanism to perform automatic re-execution. It is very similar to the roll-back mechanism for revocable monitors, except for the following differences:

- The `Rollback` exception scope wraps a transactional monitor instead of a Java-style synchronized block.
- Implementation of the data access barriers and of the log maintenance algorithm depend on the particular scheme (and are different than for revocable monitors).
- Implementation of the re-execution mechanism is simplified since the current implementation of transactional monitors does not support nesting.

As for revocable monitors, our implementation of transactional monitors is based on IBM's Jikes Research Virtual Machine.

8.1. Low-contention concurrency

Conceptually, transactional monitors use thread-local logs to record updates and install these updates into the original (shared) objects when a thread commits. However, if the contention on shared data accesses is low, the log is superfluous. If the number of objects concurrently written by different threads executing within the same monitor is small and the number of threads performing concurrent writes is also small,* then reads and writes can operate directly over the original data. To preserve correctness, an implementation must still prevent multiple non-serializable writes to objects and must disallow readers from seeing partial or inconsistent updates to objects performed by the writers.

To address these concerns, we devise a low-contention implementation that stores the following information in each transactional monitor object:

- *writer*: the thread currently executing within the monitor that has performed writes to objects guarded by the monitor;
- *thread count*: the number of threads concurrently executing within the monitor.

In this scheme, we permit only one thread executing within the monitor to perform writes to objects. Before entering a monitor, a thread must check that no writer thread is present in the monitor. Before writing to an object, a thread must ensure it is the exclusive writer.

8.1.1. Initialization

A thread attempting to enter the monitor must first check whether there is any active writer within the monitor. If there is no active writer, the thread can freely proceed after incrementing the thread count. Otherwise, shared data is not guaranteed to be in a consistent state, and the entering thread must wait until the writing thread exits the monitor. This guarantees serializability of guarded execution.

*An example of a low-contention scenario might be multiple mostly read-only threads traversing a tree-like structure or accessing a hash-table.

8.1.2. *Read and write barriers*

Because there are no object copies or logs, there are no read barriers; threads read values from the original shared objects. Write barriers are necessary to ensure that no other thread has performed writes within the monitor. A write to a shared object can occur if any one of the following conditions holds:

- The writer field in the monitor object is nil, indicating no other writers are executing within the monitor. In this case, the current thread atomically sets the writer field and executes the write.
- The writer field in the monitor points to the current thread. This implies that the current thread has previously written to an object within the monitor. The current write can proceed.

If either condition does not hold then the thread must roll back and re-execute the monitor.

8.1.3. *Conflict detection*

In order for the shared data operations of a thread exiting the monitor to be consistent and serializable with respect to other threads, there must have been no other writers within the monitor besides the exiting thread. This is guaranteed by exclusion of other threads from entering monitors in which a writer exists, and by the write barrier which revokes threads that try to write when a writer already exists. So long as there has been no concurrent writer within the monitor, actions of read-only threads are trivially serializable. Thus, read-only threads simply check this condition on monitor exit.

8.1.4. *Monitor exit*

All threads decrement the monitor thread count on exit from the monitor. The last thread to leave the monitor (*ie*, when the thread count reaches zero) clears the monitor writer field. Read-only threads successfully exit the monitor only when the writer field is nil. A writer thread always succeeds in exiting the monitor, since its writes have been validated by the write barrier at the time they occurred. Since there are no copies or logs, all updates are immediately visible in the original object.

The actions performed by the low-contention scheme executing the account example from Figure 13 are illustrated in Figure 16, where wavy lines represent threads T and T', circles represent objects *c* (checking account) and *s* (savings account), and updated objects are marked grey. The large box represents the dynamic scope of a common transactional monitor *account_monitor* guarding code regions executed by the threads and small boxes represent the additional information associated with the monitor: the writer field (initially nil) and the thread count (initially 0). In Figure 16(a) thread T' is about to enter the monitor, which it does in Figure 16(b) incrementing the thread count. In Figure 16(c) thread T also enters the monitor and increments the thread count. In Figure 16(d) thread T' updates object *c* and sets the writer to itself. Subsequently thread T reads object *c* (Figure 16(e)), thread T' updates object *s* and exits the monitor (Figure 16(f)) (no conflicts are detected since there were no intervening writes on behalf of other threads executing within the monitor). The thread count is decremented but the writer cannot be reset since thread T is still executing within the monitor. In Figure 16(g) thread T reads object *s* and attempts to exit the monitor, but the writer field still points to

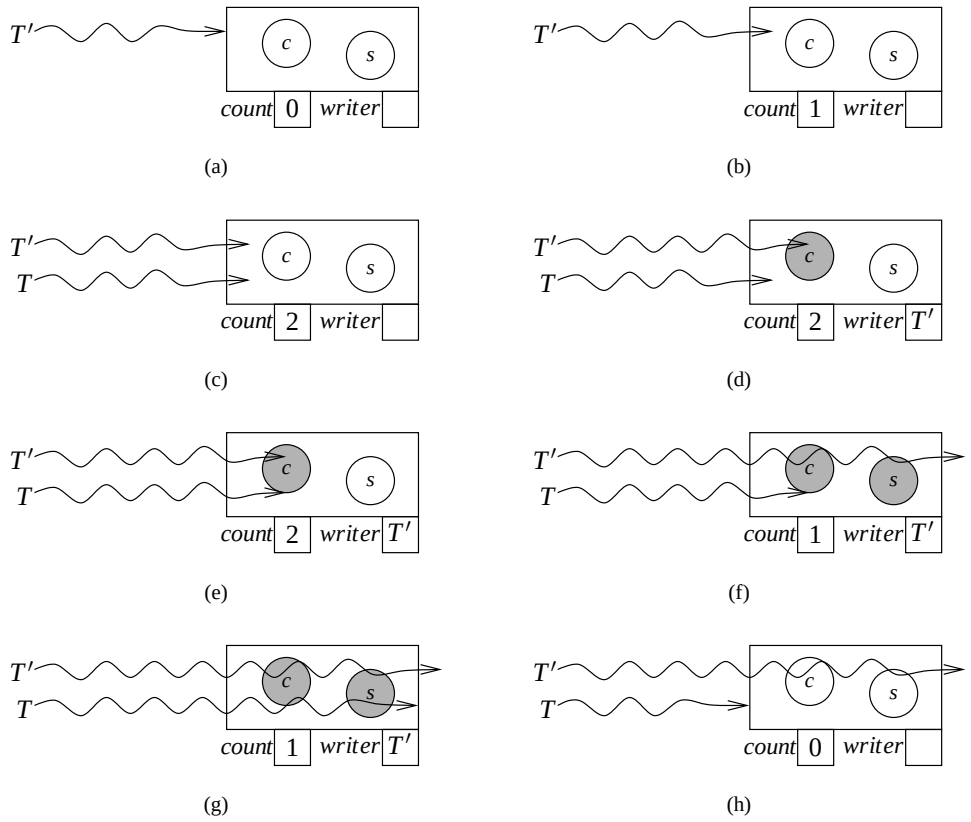


Figure 16. Low contention scheme example

thread T' indicating a potential conflict* – the guarded region of thread T must be re-executed. Since thread T is the last one to exit the monitor, in addition to decrementing the thread count it also resets the writer field (Figure 16(h)).

8.2. High-contention concurrency

When there is even moderate contention for shared data, the previous strategy is unlikely to perform well because attempts to execute multiple writes, even to distinct objects, may result in conflicts and

*This example is based on an interleaving of operations where the conflict really exists (*ie*, serializability is violated).

aborting all but one of the writers. A realistic example scenario in which contention becomes apparent is multiple threads traversing disjoint sub-trees of a tree-like structure or accessing different buckets in a hash-table. We can avoid being penalized for contention by permitting threads to manipulate *copies* of shared objects, committing their changes only when they do not conflict with the actions of other threads. This implementation is closer to the conceptual idea underlying transactional monitors: updates and accesses performed by a thread are tracked within a log, and committed only when the actions of one thread with respect to actions of other threads executing under the same guard respect serializability. Since applications tend to perform a lot more reads than writes, we decided to use a copy-on-write strategy (instead of creating copies on both reads and writes) to reduce the cost of read operations (trading this lower overhead for potential loss of precision in detecting serializability violations).

The high-contention scheme maintains the following information in each monitor:

- *global write map*: identifies the objects written by all threads executing within the monitor. This map is implemented as a bit-map with a bit being set for every modified object. The mapping is many-to-one with multiple objects possibly hashing to the same bit;
- *thread count*: the number of threads concurrently executing within the monitor.

The monitor object also contains information about whether any thread executing within a monitor has already managed to commit its updates. The global write map and thread count can be combined into one data structure to simplify access to it.

In addition to the data stored in the monitor object, the header of every object holds the following information:

- *copies*: a circular list of the object's copies, created by the different threads executing within transactional monitors (the original object is the head of the list).
- *writer*: each copy holds a reference to the thread T that created it.

Each thread also holds the following (thread-local) information:

- *local writes*: a list of object copies created by the thread when executing within the current transactional monitor;
- *local read map*: a local bit-map, implemented similarly to the global write map, which identifies those objects read by the thread within the current monitor.
- *local write map*: identifies the objects written by a given thread when executing within the current monitor.

8.2.1. Initialization

The first thread attempting to enter a monitor initializes the monitor by clearing the global write map and setting the thread counter to one. Any subsequent thread entering the monitor simply increments the thread counter, and is immediately allowed to enter the monitor provided that no thread has yet committed its updates. If updates have already been installed, the remaining threads still executing within the monitor are allowed to continue their execution, but no further threads are allowed to enter the monitor. We do this so as to avoid accumulating spurious conflicts due to threads that have successfully exited the monitor after having performed writes. Otherwise out-dated global write map

information about updates performed within the monitor might be retained for an indefinite time, and newly-entering threads would abort without reason. By preventing threads from entering the monitor once updates have been installed we allow the remaining threads to complete and the last one out will clear the global write map. Each thread entering a monitor must also clear its local data structures.

8.2.2. Read and write barriers

The barriers implement a copy-on-write semantics. The following actions are taken before writing to an object:

- If the bit representing the object in the local write map is clear (*ie*, the current thread has not yet written to this object), then a copy of the original object is created and threaded onto its per-object copy chain, and onto the list of copies for this thread. The object's bit in the local write map is set, and the write is redirected to the copy.
- If the bit representing the object in the local write map is set, then the current thread may already have a copy of this object (the mapping is imprecise). The copy is located by traversing the list of copies to find the one created by the current thread; if a copy is not found, one is created. The write is redirected to the copy and the local write map is set.

The following actions are taken before reading an object:

- If the bit representing the object in the local write map is clear (*ie*, the current thread has not yet written to this object), the local read map is first set, before the original object is read.
- If the bit representing the object in the local write map is set, then the corresponding copy is located (as above). If a copy exists, the read is performed against the copy, otherwise the original object is read; in both cases the local read map is set.

8.2.3. Conflict detection

Before a thread can exit a monitor, conflict detection checks if the global write map and the thread's local read map are disjoint. If they are disjoint then no reads by the current thread could have been interleaved with committed writes of other threads within the monitor, so the thread proceeds to exit the monitor. If the maps intersect then a potentially harmful interleaving may have occurred that may violate serializability; in this case, the exiting thread must abort and re-execute the monitored region. Only if the thread passes the test for conflicts can it proceed to exit the monitor, as follows.

8.2.4. Monitor exit

Having passed the test for possible conflicts, the thread proceeds to commit its updates atomically before exiting the monitor. The updated contents of each copy are installed in the original object, and the local write map is merged into the global write map to reflect the writes performed by the exiting thread. The copies are discarded from their circular copy list. The monitor thread count is decremented, and the per-monitor state is cleared if the counter reaches zero (there are no longer threads active within the monitor).

The actions performed in this scheme executing the account example from Figure 13 are illustrated in Figure 17, where wavy lines represent threads T and T', circles represent objects c (checking account)

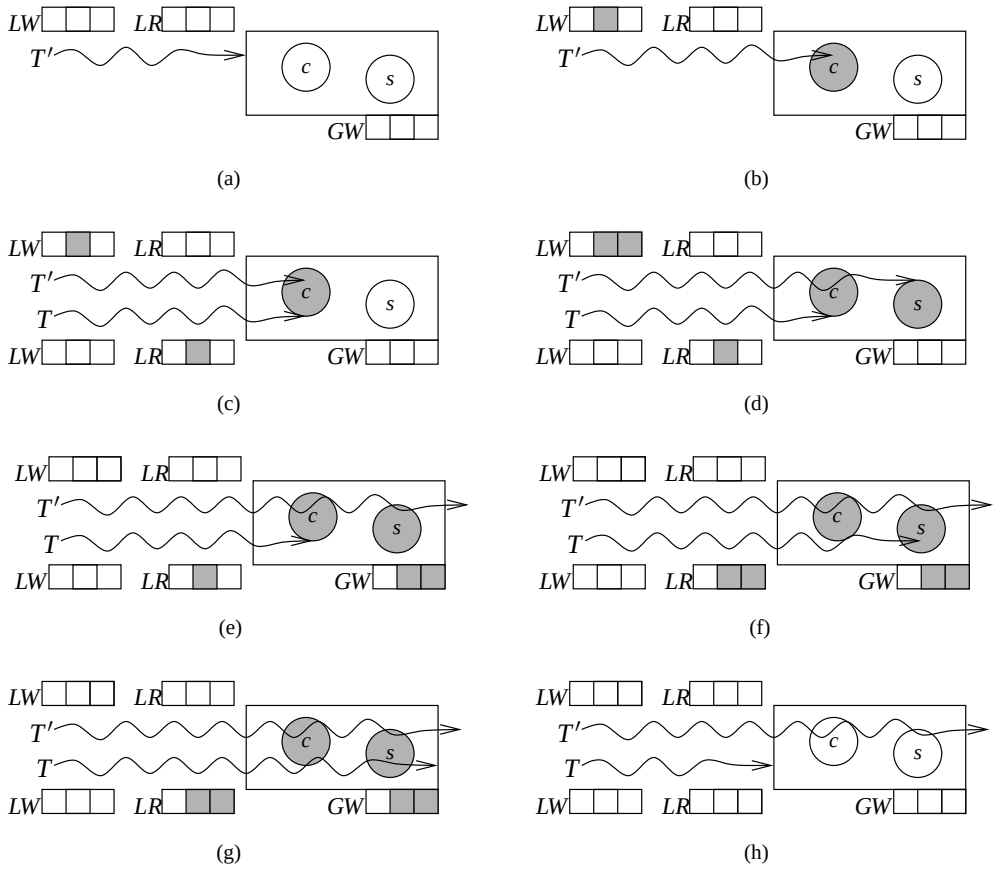


Figure 17. High contention scheme example

and *s* (saving account), and updated objects are marked grey. The box represents the dynamic scope of a common transactional monitor *account_monitor* guarding code regions executed by the threads. Both the global write map (*GW*) associated with the monitor and the local maps (write map *LW* and read map *LR*) associated with each thread have three slots. Local maps above the wavy line representing thread T' belong to T' and local maps below the wavy line representing thread T belong to T. In Figure 17(a) thread T' is about to enter the monitor, which it does in Figure 17(b), modifying object *c*. Object *c* is greyed and information about the update is reflected in the local write map of T' (we assume that object *c* hashes into the second slot of the map). In Figure 17(c) thread T enters the same monitor and reads object *c* (the read operation gets reflected in the local read map of T). In Figure 17(d) thread T' modifies object *s*, object *s* is greyed and the update also is reflected in T''s local write map (we assume

that object *s* hashes into the third slot of the map). In Figure 17(e) thread T' exits the monitor. Since no conflicts are detected (there were no intervening writes on behalf of other threads executing within the monitor), T' installs its updates, modifies the global write map to reflect updates performed within the guarded region and clears its local maps. Thread T subsequently reads object *s* marking its local read map (Fig 17(f)) and attempts to exit the monitor (Figure 17(g)). In the case of thread T however its local read map and the global write map overlap indicating a potential conflict* – the guarded region of thread T must be re-executed (Figure 17(h)). Since thread T is the last thread to exit the monitor, in addition to clearing its local maps, it also cleans up the monitor by clearing the global write map.

8.3. Java-specific issues

Realizing transactional monitors for Java requires reconciling their implementation with Java-specific features such as native method calls and existing thread synchronization mechanisms (including the `wait`/`notify` primitives). We now elaborate on these issues.

8.3.1. Native methods

In general, the effects of executing a native method cannot be undone. Thus, we disallow execution of native methods within regions guarded by transactional monitors. However, it is possible to relax this restriction in certain cases. For example, if the effects of executing a native method do not affect the shared state (*eg*, a call to obtain the current system time), it can safely be performed within a guarded region. It may also be possible to provide compensation code to be invoked when a transaction aborts that will revert the effects of native method calls executed within the aborting transaction. However, our current implementation does not provide such functionality. Instead, when a native method call occurs inside the dynamic context protected by a transactional monitor, a commit operation is attempted for the updates performed up to that point. If the commit fails, then the monitor re-executes, discarding all its updates. If the commit succeeds, the updates are retained, and execution reverts to mutual-exclusion semantics: a conventional mutual-exclusion lock is acquired for the remainder of the monitor. Any other thread that attempts to commit its changes while the lock is held must abort. Any thread that attempts to enter the monitor while the lock is held must wait.

8.3.2. Existing synchronization mechanisms

Double guarding a code fragment with both a transactional monitor and a mutual-exclusion monitor (the latter expressed using Java's `synchronized` keyword) does not strengthen existing serializability guarantees. Indeed, code protected in such a manner will behave correctly. However, the visibility rule for mutual-exclusion monitors embedded within a transactional monitor will change with respect to the original Java memory model: all updates performed within a region guarded by a mutual-exclusion monitor become visible only upon commit of the transactional monitor guarding that region.

*This example is also based on an interleaving of operations where the conflict really exists (*ie*, serializability invariants are violated).

8.3.3. Wait-notify

We allow invocation of `wait` and `notify` methods inside of a region guarded by a transactional monitor, provided that they are also guarded by a mutual-exclusion monitor (and invoked on the object representing that mutual-exclusion monitor). This requirement is identical to the original Java execution semantics – a thread invoking wait or notify must hold the corresponding monitor. Invoking `wait` releases the corresponding mutual-exclusion monitor and the current thread waits for notification, but updates performed so far do not become visible until the thread resumes and exits the transactional monitor. Invoking `notify` postpones the effects of notification until exit from the transactional monitor. That is, notification modifies the shared state of a program and is therefore subject to the same visibility rules as other shared updates.

9. Transactional monitors: Experiments

To evaluate the performance of the prototype implementation, we use and extend the multi-threaded version of the OO7 object operations benchmark [14], originally developed in the database community. Our incarnation of OO7 uses modified traversal routines to allow parameterization of synchronization and concurrency behavior. We have selected this benchmark because it provides a great deal of flexibility in the choice of run-time parameters (*eg*, percentage of reads and writes to shared data performed by the application) and extended it to allow control over placement of synchronization primitives and the amount of contention on data access. When choosing OO7 for our measurements, our goal was to accurately gauge various trade-offs inherent with different implementations of transactional monitors, rather than emulating workloads of selected potential applications. Thus, we believe the benchmark captures essential features of scalable concurrent programs that can be used to quantify the impact of the design decisions underlying a transactional monitor implementation.

9.1. The OO7 benchmark

The OO7 benchmark suite [14] provides a great deal of flexibility for benchmark parameters (*eg*, database structure, fractions of reads/writes to shared/private data). The multi-user OO7 benchmark [13] allows control over the degree of contention for access to shared data. By varying these parameters we are able to characterize the performance of transactional monitors over a mixed range of workloads.

The OO7 benchmarks operate on a synthetic design database, consisting of a set of *composite parts*. Each composite part comprises a graph of *atomic parts*, and a `document` object containing a small amount of text. Each atomic part has a set of attributes (*ie*, fields), and is connected via a bi-directional association to several other atomic parts. The connections are implemented by interposing a separate connection object between each pair of connected atomic parts. Composite parts are arranged in an *assembly* hierarchy; each assembly is either made up of composite parts (a *base* assembly) or other assemblies (a *complex* assembly). Each assembly hierarchy is called a *module*, and has an associated *manual* object consisting of a large amount of text. Our results are all obtained with an OO7 database configured as in Table I.

Our implementation of OO7 conforms to the standard OO7 database specification. Our traversals are a modified version of the multi-user OO7 traversals. A traversal chooses a single path through the

Table I. Component organization of the OO7 benchmark

Component	Number
Modules	1
Assembly levels	7
Subassemblies per complex assembly	3
Composite parts per assembly	3
Composite parts per module	500
Atomic parts per composite part	20
Connections per atomic part	3
Document size (bytes)	2000
Manual size (bytes)	100000

assembly hierarchy and at the composite part level randomly chooses a fixed number of composite parts to visit (the number of composite parts to be visited during a single traversal is a configurable parameter). When the traversal reaches the composite part, it has two choices:

1. Do a *read-only* depth-first traversal of the atomic part subgraph associated with that composite part; or
2. Do a *read-write* depth-first traversal of the associated atomic part subgraph, swapping the x and y coordinates of each atomic part as it is visited.

Each traversal can be done beginning with either a *private* module or a *shared* module. The parameter's of the workload control the mix of these four basic operations: read/write and private/shared. To foster some degree of interesting interleaving and contention in the case of concurrent execution, our traversals also take a parameter that allows extra overhead to be added to read operations to increase the time spent performing traversals.

Our experiments here use traversals that always operate on the *shared* module, since we are interested in the effects of contention on performance of transactional monitors. Our implementation of OO7 conforms to the standard OO7 database specification. Our traversals differ from the original OO7 traversals in allowing multiple composite parts to be visited during a single traversal rather than just one as in the original specification, and adding a parameter that controls entry to monitors at varying levels of the database hierarchy.

9.2. Measurements

Our measurements were obtained on an eight-way 700MHz Intel Pentium III with 2GB of RAM running Linux kernel version 2.4.20-20.9 (RedHat 9.0) in single-user mode. We ran each benchmark configuration in its own invocation of RVM, repeating the benchmark six times in each invocation, and discarding the results of the first iteration, in which the benchmark classes are loaded and compiled, to eliminate the overheads of compilation.

When running the benchmarks we varied the following parameters:

- number of threads competing for shared data access along with the number of processors executing the threads: we ran $P*8$ threads (where P is the number of processors) for $P = 1, 2, 4, 8$.

- ratio of shared reads to shared writes: from 10% shared reads and 90% shared writes (mostly read-only guarded regions) to 90% shared reads and 10% shared writes (mostly write-only guarded regions).
- level of the assembly hierarchy at which monitors were entered: level one (module level), level three (second layer of composite parts) and level six (fifth layer of composite parts). Varying the level at which monitors are entered models different locking granularities from coarse-grained (*ie*, module) through to fine-grained (*ie*, composite part).

Every thread performs 1000 traversals (enters 1000 guarded regions) and visits 2M atomic parts during each iteration.

9.3. Results

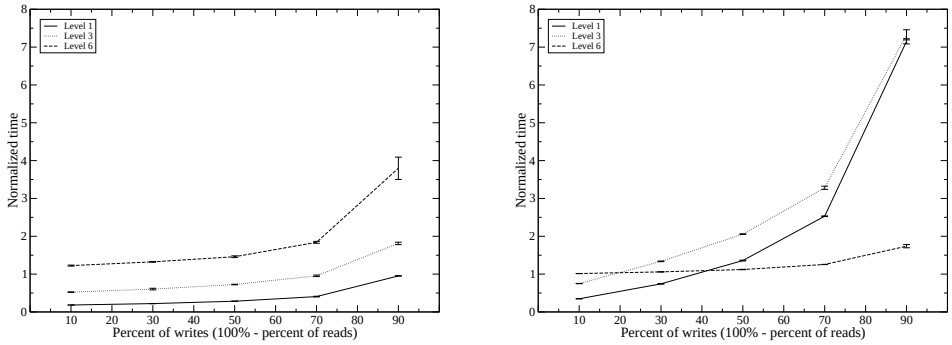
The expected behavior for transactional monitor implementations optimized for low-contention applications is one in which performance is maximized when contention on guarded shared data accesses is low, for example, if most operations in guarded regions are reads. The expected behavior for transactional monitor implementations optimized for high-contention applications is one in which performance is maximized when contention on guarded shared data accesses is moderate, the operations protected by the monitor contain a mix of reads and writes, and concurrently executing threads do not often attempt concurrent updates of the *same* object. Potential performance improvements over a mutual-exclusion implementation arise from the improved scalability that should be observable when executing on multi-processor platforms.

Our experimental results confirm these hypotheses. Contention on shared data accesses depends on the number of updates performed within guarded regions combined with the amount of contention on entering monitors.* Figure 18 plots execution time for 64 threads running on 8 processors for the high-contention scheme (Figure 18(a)) and low-contention scheme (Figure 18(b) normalized to the execution time for standard mutual-exclusion monitors,† while varying the ratio of shared reads and writes and the level at which monitors are entered. It is important to note that only monitor entries at levels one and three create any reasonable contention on shared data accesses – at level six the probability of two threads concurrently entering the same monitor is very low (thus no performance benefit can be expected). In Figure 18(a) we see the high-contention scheme outperforming mutual-exclusion monitors for *all* configurations when monitors are entered at level one. When monitors are entered at level three, the high-contention scheme outperforms mutual-exclusion monitors for the configurations where write operations constitute 70% of all data operations. For larger write ratios, the number of aborts and the number of copies created during guarded execution overcome any potential benefit from increased concurrency.

The low-contention scheme's performance is illustrated in Figure 18(b): it outperforms mutual-exclusion monitors for configurations where write operations constitute 30% of all data operations (low contention on shared data accesses). The total number of aborts across all iterations for both

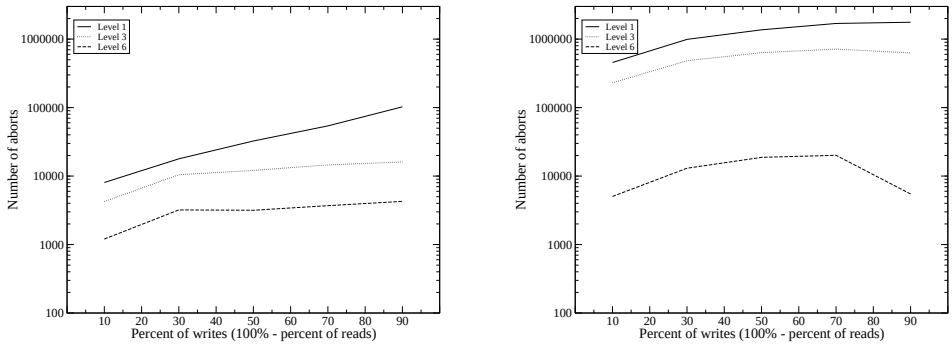
*Threads contend on entering a monitor only if they enter the *same* monitor.

†To obtain results for the mutual-exclusion case we used an unmodified version of Jikes RVM (no compiler or run-time modifications). Figures reporting execution times show 90% confidence intervals in our results.



(a) high-contention (b) low-contention

Figure 18. Normalized execution time for 64 threads running on 8 processors



(a) high-contention (b) low-contention

Figure 19. Total aborts for 64 threads running on 8 processors

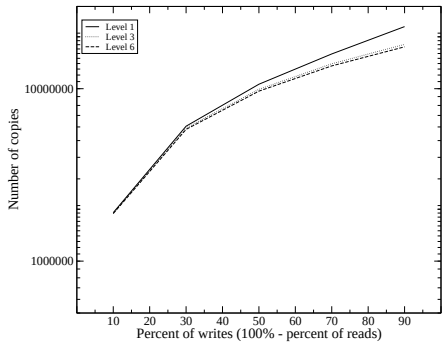
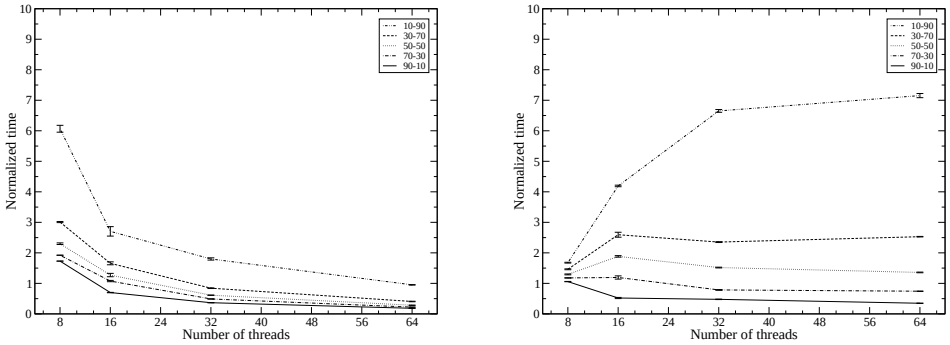
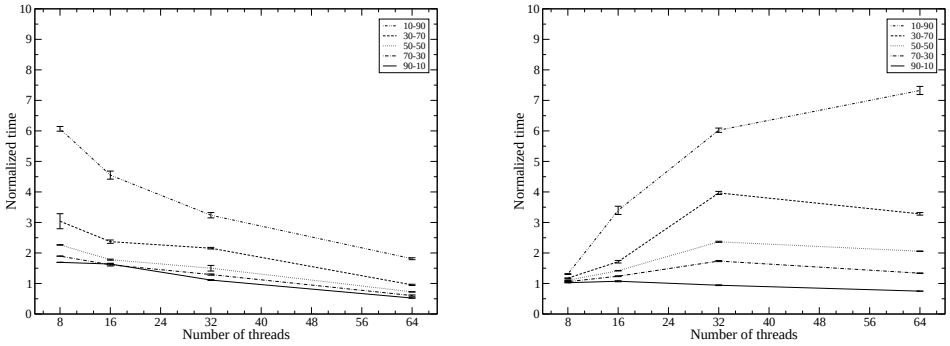


Figure 20. Total copies created for 64 threads running on 8 processors



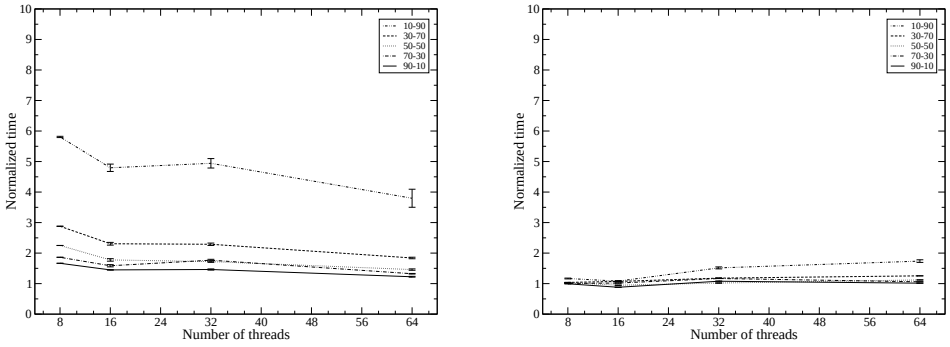
(a) high-contention (b) high-contention

Figure 21. Normalized execution times – monitor entries at level 1



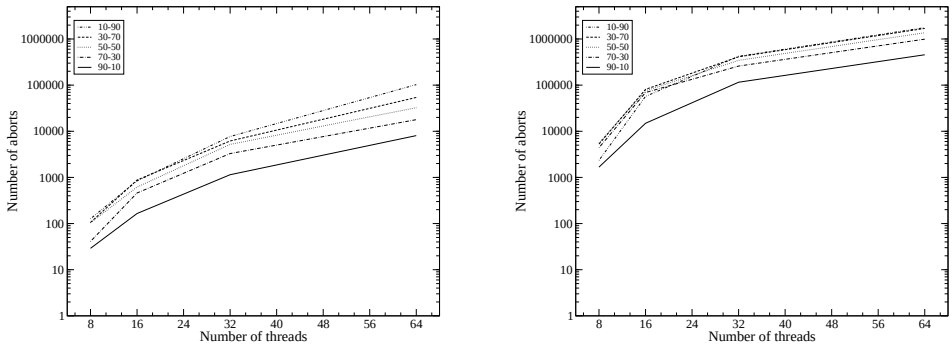
(a) high-contention (b) high-contention

Figure 22. Normalized execution times – monitor entries at level 3



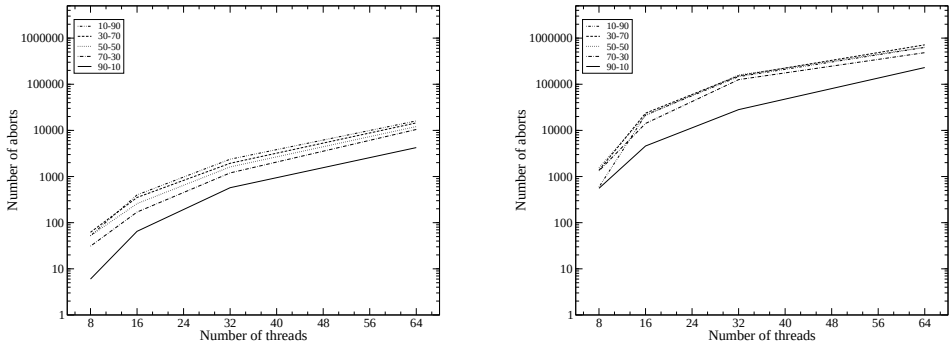
(a) high-contention (b) high-contention

Figure 23. Normalized execution times – monitor entries at level 6



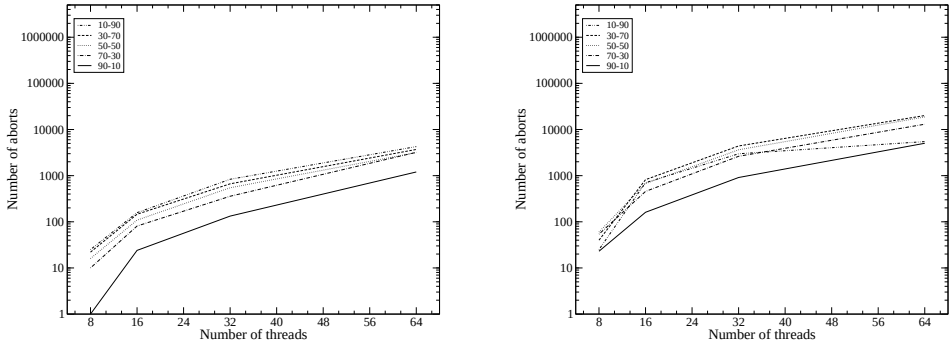
(a) high-contention (b) low-contention

Figure 24. Total aborts – monitor entries at level 1



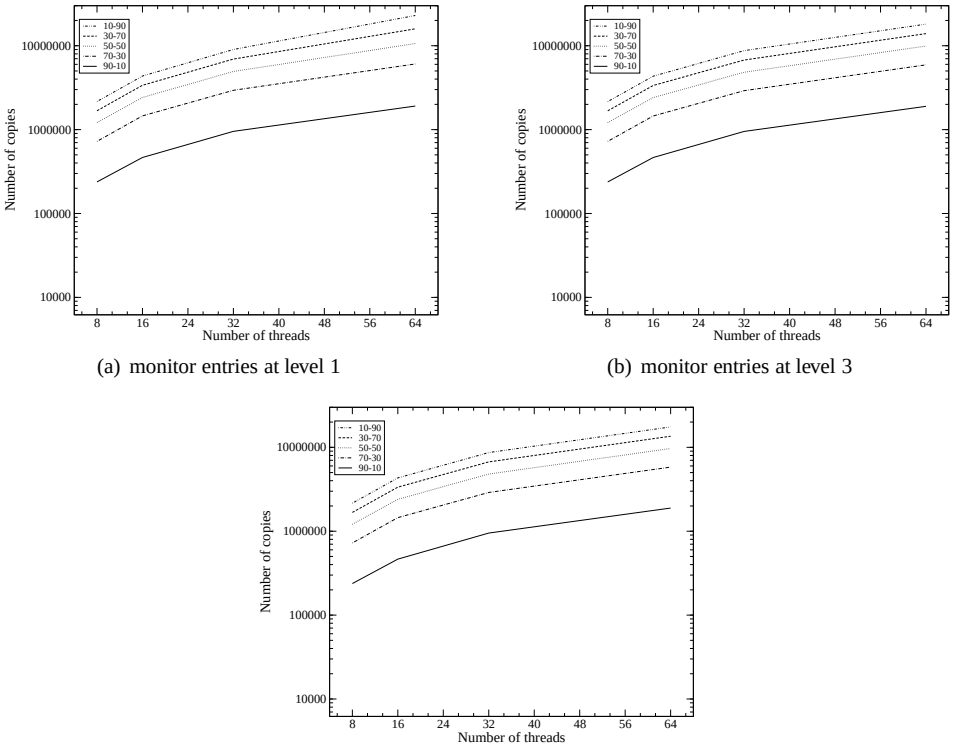
(a) high-contention (b) low-contention

Figure 25. Total aborts – monitor entries at level 3



(a) high-contention (b) low-contention

Figure 26. Total aborts – monitor entries at level 6



(a) monitor entries at level 1

(b) monitor entries at level 3

(c) monitor entries at level 6

Figure 27. Total copies

high-contention scheme and low-contention scheme appears in Figure 19. The total number of copies created across all iterations for the high-contention scheme appears in Figure 20. The remaining graphs illustrate the scalability of both schemes by plotting normalized execution times for the high-contention scheme (Figures 21(a)-23(a)) and low-contention scheme (Figures 21(b)-23(b)) when varying the number of threads (and processors) for monitor entries placed at levels one, three, and six (Figures 21-23, respectively), along with the information concerning number of aborts and copies created (Figures 24-26 and Figure 27(a)-27(c), respectively).

We observe that the reduced performance of the low contention scheme for higher shared write ratios is caused almost exclusively* by an increase in the number of aborts (graphs reporting number of aborts

*The run-time overheads of the low contention scheme are low: no logging and no read barriers. As a result performance of this scheme in the case when there is almost no contention on entering monitors is only slightly worse than that of the mutual-exclusion monitors (Figure 23(b)).

are plotted on a logarithmic scale)(Figures 24(b) and (Figures 25(b)). It is a direct result of a degree of imprecision in detecting serializability violations. Note however, that the low contention scheme has been specifically designed to perform well only when executing workloads when concurrent writes are infrequent and as such has exactly met our expectations. Conversely, the high contention scheme has higher run-time costs and therefore its performance is reduced when there is not enough opportunity for increased concurrency (Figures 21(a)-23(a) – 8 threads executing on 1 CPU), even though the number of aborts is in these cases relatively low (Figures 24-26(a)).

10. Related work

Our use of roll-backs to redo computation inside monitored regions is reminiscent of optimistic concurrency protocols first introduced in the 1980's [29] to improve database performance. Given a collection of transactions, the goal in an optimistic concurrency implementation is to ensure that only a serializable schedule results [1, 24, 42]. Devising fast and efficient techniques to confirm that a schedule is correct remains an important topic of study.

Several recent efforts explore alternatives to lock-based concurrent programming. Harris *et al* [22] introduce a new synchronization construct to Java called *atomic* that is superficially similar to our transactional monitors. The idea behind the atomic construct is that logically only one thread appears to execute *any* atomic block at a time. However, it is unclear how to translate their abstract semantic definition into a practical implementation. For example, a complex data structure enclosed within *atomic* is subject to a costly *validation* check, even though operations on the structure may occur on separate disjoint parts. We regard our work as a significant extension and refinement of their approach, especially with respect to understanding implementation issues related to the effectiveness of new concurrency abstractions on realistic multi-threaded applications. Thus, we focus on a detailed quantitative study to measure the cost of logging, commits, aborts, *etc*; we regard such an exercise as critical to validate the utility of these higher-level abstractions on scalable platforms.

Lock-free data structures [38, 28] and transactional memory [26, 41] are also closely related to transactional monitors. Herlihy *et al* [25] present a solution closest in spirit to transactional monitors. They introduce a form of software transactional memory that allows for the implementation of *obstruction-free* (a weaker incarnation of lock-free) data structures. However, because shared data accesses performed in a transactional context are limited to statically pre-defined *transactional objects*, their solution is less general than the dynamic protection afforded by transactional monitors. Moreover, the overheads of their implementation are also unclear. They compare the performance of operations on an obstruction-free red-black tree only with respect to other lock-free implementations of the same data structure, disregarding potential competition from a carefully crafted implementation using mutual-exclusion locks. The notion of transactional lock removal proposed by Rajwar and Goodman [38] also shares similar goals with our work, but their implementation relies on hardware support.

Recently, Pizlo *et al* [37] proposed a transactions-based solution to resolving priority inversion in the context of the Real-Time Specification for Java (RTSJ). They extended RTSJ with a notion of transactional lock-free (TLF) objects whose methods can be designated as atomic and run under the protection of lightweight transactions (currently only one thread is allowed to execute a given method at a time). The basic idea underlying their solution is similar to that of revocable monitors – a low priority thread executing an atomic method can be interrupted and revoked by a higher priority thread.

However, their major focus is on providing real-time guarantees for transactional execution, such as bounding the space consumed by transaction logs, while our emphasis is improving throughput of high-priority threads in a semantically transparent way.

Rinard [39] describes experimental results using low-level optimistic concurrency primitives in the context of an optimizing parallelizing compiler that generates parallel C++ programs from unannotated serial C++ source. Unlike a general transaction facility of the kind described here, his optimistic concurrency implementation does not ensure atomic commitment of multiple variables. Moreover in contrast to a low-level facility, the code protected by transactional monitors may span an arbitrary dynamic context. In similar vein, Harris and Fraser [23] propose another low-level mechanism called revocable locks. A revocable lock is associated with a single heap location and, once acquired by a thread, can be revoked by another thread attempting to access the same location by gaining ownership of the same lock. The intended use of revocable locks is quite different from that of higher level synchronization mechanisms such as revocable monitors because they are meant to be used as a building block *within* implementations of such high-level abstractions.

There has been much recent interest in data race detection for Java. Some approaches [7, 8] present new type systems using, for example, ownership types [17] to verify the absence of data races and deadlock. Recent work on generalizing type systems allows reasoning about higher-level atomicity properties of concurrent programs that subsumes data race detection [19, 18]. Other techniques [44] employ static analyses such as escape analysis along with run-time instrumentation that meters accesses to synchronized data. Transactional monitors share similar goals with these efforts but differ in some important respects. In particular, our approach does not rely on global analysis, programmer annotations, or alternative type systems. While it replaces lock-based implementations of synchronized blocks, the set of schedules it allows is not identical to that supported by lock-based schemes. Indeed, transactional monitors ensure preservation of atomicity and serializability properties in guarded regions without enforcing a rigid schedule that prohibits benign concurrent access to shared data. In this respect, they can be viewed as a starting point for an implementation that supports higher-level atomic operations.

There have been several attempts to reduce locking overhead in Java. Agesen *et al* [2] and Bacon *et al* [4] describe locking implementations for Java that attempt to optimize lock acquisition overhead when there is no contention on a shared object. Transactional monitors obviate the need for a multi-tiered locking algorithm by allowing multiple threads to execute simultaneously within guarded regions provided that updates are serializable.

Finally, the formal specification of various flavors of transactions has received much attention [31, 16, 21]. Black *et al* [6] present a theory of transactions that specifies atomicity, isolation and durability properties in the form of an equivalence relation on processes. Choithia and Duggan [15] present the pik-calculus and pike-calculus as extensions of the pi-calculus that support abstractions for distributed transactions and optimistic concurrency. Their work is related to other efforts [9] that encode transaction-style semantics into the pi-calculus and its variants. The work of Busi, Gorrieri and Zavattaro [10] and Busi and Zavattaro [12] formalize the semantics of JavaSpaces, a transactional coordination language for Linda, and discuss the semantics of important extensions such as leasing [11]. Berger and Honda [5] examine extensions to the pi-calculus to handle various forms of distributed computation include aspects of transactional processing such as two-phase commit protocols for handling commit actions in the presence of node failures. We have recently applied the ideas presented here to define an optimistic concurrency (transaction-like) semantics for a Linda-

like coordination language that addresses scalability limitations in these other approaches [27]. A formalization of a general transaction semantics for programming languages expressive enough to capture the behavior of transactional monitors is presented in [43].

11. Conclusions

We have presented a revocation-based priority inversion avoidance technique and demonstrated its utility in improving throughput of high priority threads in a priority scheduling environment. The solution proposed is relatively simple to implement, portable, and can be adopted to solve other types of problems (*eg*, deadlocks). We have also introduced transactional monitors, a new synchronization mechanism, alternative to mutual-exclusion. Transactional monitors preserve the semblance of serial execution within monitored regions and are implemented as lightweight transactions that can be executed concurrently. We have presented two different schemes tailored to different concurrent access patterns and examined their performance and scalability. All the techniques we described use compiler support to insert barriers to monitor accesses to shared data, and run-time modifications to implement revocation.

REFERENCES

1. ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record 24*, 2 (June 1995), 23–34.
2. AGESEN, O., DETLEFS, D., GARTHWAITE, A., KNIPPEL, R., RAMAKRISHNA, Y. S., AND WHITE, D. An efficient meta-lock for implementing ubiquitous synchronization. In OOPSLA'99 [36], pp. 207–222.
3. ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., NGO, T., MERGEN, M., SHEPHERD, J. C., AND SMITH, S. Implementing Jalapeño in Java. In OOPSLA'99 [36], pp. 314–324.
4. BACON, D., KONURU, R., MURTHY, C., AND SERRANO, M. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Montréal, Canada, June). *ACM SIGPLAN Notices 33*, 5 (May 1998), pp. 258–268.
5. BERGER, M., AND HONDA, K. The Two-Phase Commitment Protocol in an Extended pi-Calculus. In *Satellite workshops from CONCUR 2000* (San Francisco, California, Aug.). *Electronic Notes in Theoretical Computer Science 39*, 1 (2000).
6. BLACK, A., CREMET, V., GUERRAOUI, R., AND ODERSKY, M. An equational theory for transactions. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science* (Mumbai, India, Dec.). vol. 2914 of *Lecture Notes in Computer Science*. 2003, pp. 38–49.
7. BOYAPATI, C., LEE, R., AND RINARD, M. C. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Seattle, Washington, Nov.). *ACM SIGPLAN Notices 37*, 11 (Nov. 2002), pp. 211–230.
8. BOYAPATI, C., AND RINARD, M. A parameterized type system for race-free Java programs. In OOPSLA'01 [35], pp. 56–69.
9. BRUNI, R., LANEVE, C., AND MONTANARI, U. Orchestrating transactions in the join calculus. In *Proceedings of the International Conference on Concurrency Theory* (Brno, Czech Republic, Aug.), L. Brim, M. K. Petr Jancar, and A. Kucera, Eds. vol. 2421 of *Lecture Notes in Computer Science*. 2002, pp. 321–337.
10. BUSI, N., GORRIERI, R., AND ZAVATTARO, G. On the semantics of JavaSpaces. In *Formal Methods for Open Object-Based Distributed Systems IV* (Stanford, California, Sept.), S. F. Smith and C. L. Talcott, Eds. vol. 177 of *IFIP Conference Proceedings*. Kluwer, 2000.
11. BUSI, N., GORRIERI, R., AND ZAVATTARO, G. Temporary data in shared dataspace coordination languages. In *Proceedings of the International Conference on Foundations of Software Science and Computation Structures* (Genova, Italy, Apr.), F. Honsell and M. Miculan, Eds. vol. 2030 of *Lecture Notes in Computer Science*. 2001, pp. 121–136.
12. BUSI, N., AND ZAVATTARO, G. On the serializability of transactions in shared dataspaces with temporary data. In *Proceedings of the ACM Symposium on Applied Computing* (Madrid, Spain, Mar.). 2002, pp. 359–366.

13. CAREY, M. J., DEWITT, D. J., KANT, C., AND NAUGHTON, J. F. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, Oct.). *ACM SIGPLAN Notices 29*, 10 (Oct. 1994), pp. 414–426.
14. CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. The OO7 benchmark. In *Proceedings of the ACM International Conference on Management of Data* (Washington, DC, May). *ACM SIGMOD Record 22*, 2 (June 1993), pp. 12–21.
15. CHOITHIA, T., AND DUGGAN, D. Abstractions for fault-tolerant computing. Tech. Rep. 2003-3, Department of Computer Science, Stevens Institute of Technology, 2003.
16. CHRYSANTHIS, P., AND RAMAMRITHAM, K. Synthesis of extended transaction models using ACTA. *ACM Trans. Database Syst. 19*, 3 (1994), 450–491.
17. CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership types for flexible alias protection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, Oct.). *ACM SIGPLAN Notices 33*, 10 (Oct. 1998), pp. 48–64.
18. FLANAGAN, C., AND FREUND, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Venice, Italy, Jan.). 2004, pp. 256–267.
19. FLANAGAN, C., AND QADEER, S. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation* (New Orleans, Louisiana, Jan.). 2003, pp. 1–12.
20. GOSLING, J., JOY, B., STEELE, JR., G., AND BRACHA, G. *The Java Language Specification*, second ed. Addison-Wesley, 2000.
21. GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Data Management Systems. Morgan Kaufmann, 1993.
22. HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Anaheim, California, Nov.). *ACM SIGPLAN Notices 38*, 11 (Nov. 2003), pp. 388–402.
23. HARRIS, T., AND FRASER, K. Revocable locks for non-blocking programming. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, Illinois, June). 2005.
24. HERLIHY, M. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. Database Syst. 15*, 1 (1990), 96–124.
25. HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing* (Boston, Massachusetts, July). 2003, pp. 92–101.
26. HOSKING, A. L., AND MOSS, J. E. B. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Washington, DC, Sept.). *ACM SIGPLAN Notices 28*, 10 (Oct. 1993), pp. 288–303.
27. JAGANNATHAN, S., AND VITEK, J. Optimistic concurrency semantics for transactions in coordination languages. In *Coordination Models and Languages, 6th International Conference COORDINATION 2004* (Pisa, Italy, Feb.), R. D. Nicola, G. L. Ferrari, and G. Meredith, Eds. vol. 2949 of *Lecture Notes in Computer Science*. 2004, pp. 183–198.
28. JENSEN, E. H., HAGENSEN, G. W., AND BROUGHTON, J. M. A new approach to exclusive data access in shared memory multiprocessors. Tech. rep., Lawrence Livermore National Laboratories, 1987.
29. KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 9*, 4 (June 1981), 213–226.
30. LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.
31. LYNCH, N., MERRITT, M., WEIHL, W., AND FEKETE, A. *Atomic Transactions*. Morgan Kaufmann, 1994.
32. MANSON, J., AND PUGH, W. JSR133: Java memory model and thread specification, 2004.
33. MOSS, J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Apr. 1981. Also published as MIT Laboratory for Computer Science Technical Report 260.
34. MOSS, J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.
35. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Tampa, Florida, Oct.). *ACM SIGPLAN Notices 36*, 11 (Nov. 2001).
36. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Nov.). *ACM SIGPLAN Notices 34*, 10 (Oct. 1999).
37. PIZLO, F., PROCHAZKA, M., JAGANNATHAN, S., AND VITEK, J. Transactional lock-free objects for real-time Java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs* (jul). 2004, pp. 54–62.
38. RAJWAR, R., AND GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, Oct.). *ACM SIGPLAN Notices 37*, 10 (Oct. 2002), pp. 5–17.
39. RINARD, M. Effective fine-grained synchronization for automatically parallelized programs using optimistic

synchronization primitives. *ACM Trans. Comput. Syst. 17*, 4 (Nov. 1999), 337–371.
40. SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 29*, 9 (Sept. 1990), 1175–1185.
41. SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Canada, Aug.). 1995, pp. 204–213.
42. STONEBRAKER, M., AND HELLERSTEIN, J., Eds. *Readings in Database Systems*, third ed. Morgan Kaufmann, 1998.
43. VITEK, J., JAGANNATHAN, S., WELC, A., AND HOSKING, A. L. A semantic framework for designer transactions. In *Proceedings of the European Symposium on Programming* (Barcelona, Spain, Mar./Apr), D. E. Schmidt, Ed. vol. 2986 of *Lecture Notes in Computer Science*. 2004, pp. 249–263.
44. VON PRAUN, C., AND GROSS, T. R. Object race detection. In OOPSLA'01 [35], pp. 70–82.