Preemption-Based Avoidance of Priority Inversion for Java

Adam Welc welc@cs.purdue.edu
Antony L. Hosking hosking@cs.purdue.edu
Suresh Jagannathan suresh@cs.purdue.edu
Department of Computer Sciences
Purdue University
West Lafayette, IN 47906

Abstract

Priority inversion occurs in concurrent programs when low-priority threads hold shared resources needed by some high-priority thread, causing them to block indefinitely. Shared resources are usually guarded by low-level synchronization primitives such as mutual-exclusion locks, semaphores, or monitors. There are two existing solutions to priority inversion. The first, establishing high-level scheduling invariants over synchronization primitives to eliminate priority inversion a priori*, is difficult in practice and undecidable in general. Alternatively, run-time avoidance mechanisms such as priority inheritance still force high-priority threads to wait until desired resources are released.*

We describe a novel compiler and run-time solution to the problem of priority inversion, along with experimental evaluation of its effectiveness. Our approach allows preemption of any thread holding a resource needed by higher-priority threads, forcing it to release its claim on the resource, roll back its execution to the point at which the shared resource was first acquired, and discard any updates made in the interim.

The compiler inserts code at synchronization points, permitting rollback of thread execution, and efficient revocation of interim updates. Our design and implementation are realized in the context of IBM's Jikes RVM, a high-quality compiler and runtime system for Java. Our performance results show that throughput of high-priority threads using our scheme can be improved by 30% to 100% when compared with a classical scheduler that does not address priority inversion.

1 Introduction

Modern programming languages (*eg*, ML, Java, C++, and Modula-3) support concurrent programming, either through built-in language primitives, or via a set of external libraries. In general, the basic units of concurrent execution are *threads*. Threads can interact by accessing and modifying objects in their shared address space, synchronizing their actions via mutual exclusion *locks*.

The resulting programming model is reasonably simple, but unfortunately unwieldy for large-scale applications. A significant problem with using low-level, lock-based synchronization primitives is priority inversion. We propose a new solution for priority inversion that exploits close cooperation between the compiler and the run-time system. Our approach is applicable to any language that offers the following mechanisms:

- Multithreading: concurrent threads of control executing over objects in a shared address space.
- Synchronized sections: lexically-delimited blocks of code, guarded by dynamically-scoped monitors. Threads synchronize on a given monitor, acquiring it on entry to the block and releasing it on exit. Only one thread may execute within a synchronized section at any time, ensuring exclusive access to all monitor-protected blocks. Monitors are usually implemented using locking, with acquisition of a mutual exclusion lock on entry, and release of the lock on exit.
- 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.

The advantages of multithreading, for I/O-bound applications, human interfaces, distributed and parallel systems are established and well-understood. However, the difficulties in using locking with multiple threads are also widely-recognized.

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 and making unbounded the time that T_l (and hence T_h) must wait. Such situations can play havoc in real-time systems where high-priority threads demand some level of guaranteed throughput.

The ease with which even experienced programmers can write programs that exhibit priority inversion makes it worthwhile to explore transparent solutions that dynamically resolve priority inversion by reverting programs to consistent states when it occurs, while preserving source program semantics. 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 statically determine when priority inversion may arise. For such applications, the ability to transparently redress undesirable interactions between scheduling decisions and lock management is very useful.

In this paper, we propose a scheme to deal with priority inversion for monitor-based prorgrams that differ from existing solutions such as *priority ceiling* and *priority inheritance*. These latter techniques, although useful for certain applications, suffer from several significant drawbacks. Priority ceiling requires a programmer to manually specify the ceiling (the highest priority of any thread that uses the lock) for every lock, and is therefore not transparent to applications. Although priority inheritance does not exhibit this particular problem, it has several other notable disadvantages [25] that encourage us to explore alternative approaches: (a) It is non-trivial to implement; (b) Because it is a transitive operation, it may lead to unpredictable performance degradation when nested regions are protected by priority inheritance locks; (c) The existence of non-inheriting blocking operations (*eg*, synchronous inter-thread communication) may lead to unbounded inversion delay.

1.1 Our contribution: Monitor rollback

Our approach is to combine compiler techniques with run-time detection and resolution of priority inversion. Having detected priority inversion, the run-time system selectively revokes the effects of the offending thread within a synchronized section in order to resolve the problem. The compiler provides support by injecting *write barriers* to log updates to shared state performed by threads active in synchronized sections and generating code that allows interruption and revocation of the active thread. Note that this means *all* compiled code needs at least a fast-path test on every non-local update to check if the thread is executing within a synchronized section, with the slow path logging the update if it is. Compiler analyses and optimization may elide these run-time checks when the update can be shown statically never to occur within a synchronized section.

Detection of priority inversion (either at lock acquisition, or periodically in the background) by the run-time system triggers revocation of the relevant critical section for its associated thread. The run-time system interrupts the target thread, revokes the updates for the section, and transfers control for the thread back to the beginning of the section for retry. Externally, the end effect of the rollback is as if the low-priority thread never executed the section.

The process of revoking effects performed by a thread within a synchronized section is illustrated in Figure 1, 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 some (set of) synchronized section(s) executed by the threads. In Figure 1(a) low-priority thread T_l is about to enter the synchronized section, which it does in Figure 1(b), modifying object o_1. High-priority thread T_h tries to acquire the same monitor, but is blocked by low-priority T_l (1(c)). Here, a priority inheritance [21] approach 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. 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 section (1(d). Here T_l must wait while T_h now enters the monitor, and updates objects o_1 (1(e)) and o_2, before leaving (1(f)). At this point the monitor is released and T_l will gain reentry.

Note that the same technique can also be used to detect and resolve *deadlock*. 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. Generally, deadlocks may occur among more than two threads, and deadlocking programs are often difficult to diagnose and fix. As a result, many deployed applications may execute under schedules in which deadlock may occur. Using our techniques, such deadlocks can be detected and resolved automatically, permitting the application to make progress. Of course, applications that deadlock are intrinsically incorrect; our approach is not intended to mask such errors. However, for mission-critical applications in which running programs cannot be summarily terminated, our approach provides an opportunity for corrective action to be undertaken gracefully. Note that while deadlocks can be handled using our technique, without taking additional precautions a sequence of deadlock revocations may result in livelock.

The remainder of this paper details design choices and modifications made to IBM's Jikes Research Virtual Machine (Jikes RVM) to implement our approach. Section 2 outlines the design. Section 3 details several implementation choices for efficiently implementing rollbacks. Performance results are given in Section 4. Related work is discussed in Section 5.

2 Design

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

Our work is couched in terms of the Java language. In Java, every object can act as a *monitor*. A thread holding a

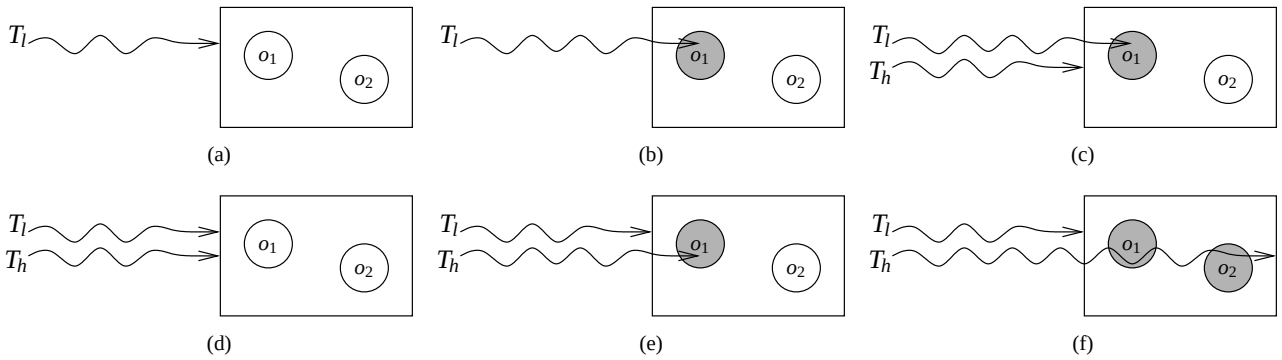


Figure 1. Revoking synchronized sections

monitor may enter another synchronized section guarded by the same or a completely different monitor. This action may be repeated an arbitrary number of times thus permitting monitors to be arbitrarily nested dynamically.

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 to permit a higher priority thread to run. In effect, synchronized sections execute speculatively, and their updates may be revoked at any time until the section exits. However, the introduction of revocable synchronized sections requires a careful consideration of the interaction between revocation and the Java Memory Model (JMM) [16].

2.1 The Java memory model

The JMM defines a *happens-before* relation (written $\stackrel{hb}{\rightarrow}$) 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 $\mathtt{u}_M$ (release) and a subsequent lock $\mathtt{l}_M$ (acquire) operation on a given monitor M ($\mathtt{u}_M \stackrel{hb}{\rightarrow} \mathtt{l}_M$). The happens-before relation is transitive: $\mathtt{x} \stackrel{hb}{\rightarrow} \mathtt{y}$ and $\mathtt{y} \stackrel{hb}{\rightarrow} \mathtt{z}$ imply $\mathtt{x} \stackrel{hb}{\rightarrow} \mathtt{z}$. 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 $\mathtt{r}_v \stackrel{hb}{\rightarrow} \mathtt{w}'_v \stackrel{hb}{\rightarrow} \mathtt{w}_v$ (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.

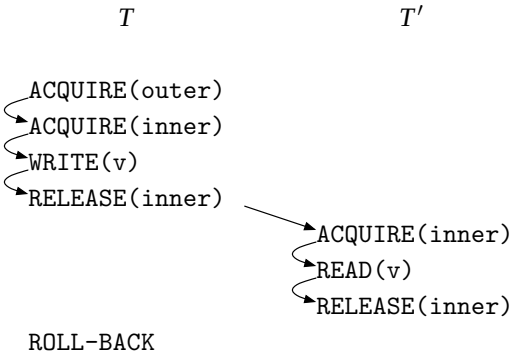


Figure 2. Bad revocation: nesting

An example of such an execution appears in Figure 2: 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 rollback point: the read performed by T' is *allowed* but the subsequent rollback of T would violate consistency.

A similar problem occurs when *volatile* variables are used. The Java Language Specification (JLS) [8] 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 3 `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 rollback point because a read performed by T' is *allowed*, but the rollback would violate consistency. We now discuss possible solutions to these JMM-consistency preservation problems.

2.2 Preserving JMM-consistency

Several solutions to the problem of partial results of a monitored computation being exposed to other threads can

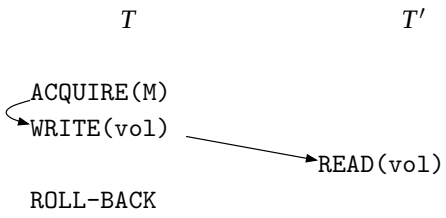


Figure 3. Bad revocation: volatile access

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

T	T'

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

Figure 4. Impossible re-schedule

be considered. We might trace read-write dependencies among all threads and upon rollback of a monitor trigger a cascade of rollbacks 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 rollback. In the execution of Figure 3 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 program threads would have to be logged in addition to shared data operations performed by the threads. Consider a situation based on the example of Figure 2 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 rollback of monitor `inner` by that thread becomes infeasible.

Another potential solution is to re-schedule the execution of threads in problematic cases. In both of the examples presented above, if thread T' executes fully before thread T, the execution will still be JMM-consistent. The rollback 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 this solution (some knowledge about future actions performed by threads would be required), there also remains the issue of correctness. While correct in some cases, this solution is not necessarily correct in others. Consider the Java program presented in Figure 4. We cannot re-schedule thread T' to execute fully before thread T because of semantic constraints: termination of T' depends on thread T performing the operation `v=true`.

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 rollback 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 performed within M[1] and a read performed by another thread. 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. Determining the precise impact of this design choice on the effectiveness of our technique is an integral part of future research.

There exist 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). The same applies to executions where a *wait* method is invoked within a nested monitor.[2] A revocation of the wait call would result in a situation where a respective *notify* call (that "woke up" the waiting thread) "disappears" (*ie*, does not get delivered to any thread) which would violate the Java execution semantics. A call to notify does not enforce the irrevocability of the enclosing monitors: Java VM implementations are permitted [16] to perform "spurious wake-ups" so a rolled back notification can be considered as such.

3 Implementation

In order 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 [2], a state-of-the-art 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 what we believe is the main contribution of this paper, namely *how* to resolve a priority inversion problem once detected (though later parts of the paper also briefly discuss how the VM determines *when* a priority inversion situation occurs and needs to be resolved). In other words

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

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

we concentrate on the description of necessary compiler and run-time support that allows the VM to interrupt execution of synchronized sections at arbitrary points without inducing any observable effects on an application's execution behavior. In subsequent sections, we describe how to implement the re-execution procedure itself: transparently return control from an arbitrary point during a synchronized section's execution to that section's starting point, restore the state of the VM, and re-execute the synchronized section's code.

3.1 Our strategy

Our implementation uses bytecode rewriting[3] to save program state (values of local variables and method parameters) for re-execution and to return control to the beginning of the synchronized section. We modify the compiler and run-time system to suppress generation (and invocation) of undesirable exception handlers during a rollback operation, to insert write barriers for logging, to revert updates performed during the unfinished execution of a synchronized section, and to augment context-switch code invoked at yield points[4] to check also whether a rollback action must be initiated.

3.1.1 Bytecode transformation

There exist two different synchronization constructs in Java: synchronized methods and synchronized blocks. To treat them uniformly, we transform 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 invocation of the original (non-synchronized) method, which has been appropriately renamed to avoid name clash. We also instruct the VM to inline the original method within the wrapper to avoid performance penalties related to the additional method invocation. This approach greatly simplifies the implementation,[5] is extremely simple, robust, and also efficient, because of inlining directives.

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

There is an additional complication related to the return of control to the beginning of the section. The contents of the VM's operand stack before executing a *monitorenter* operation must be the same at the first invocation and at all subsequent invocations resulting from that section's re-execution. However, according to the Java VM specification [14], 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 rollback-scope's *monitorenter* opcode, and to restore the stack state in the handler before transferring control back to the *monitorenter*.

3.1.2 Compiler and run-time modifications

The rollback operation is initiated by throwing the rollback exception (as described in the previous section). However, we cannot rely on the default exception handling mechanism to propagate the rollback exception up the activation stack to the synchronized section being rolled back, since it will also run "default" exception handlers in nested exception scopes as it unwinds the stack up to the rollback scope. 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, the augmented exception handling routine ignores all handlers (including `finally` blocks) that do not explicitly catch the *rollback* exception, when one 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.

We also modified both compilers to inject write barriers before every store operation (represented by the bytecodes: `putfield` for object stores, `putstatic` for static variable stores, and `Xastore` for array stores). The barrier records in the log every modification performed by a thread executing a synchronized section. We implemented the log as a sequential buffer. For object and array stores, three values are recorded: object or array reference, value offset and the (old) value itself. For static variable stores two values are recorded: the offset of the static variable in the global symbol table and the old value of the static variable.

If the execution of a synchronized section is interrupted and needs to be re-executed then the log is processed in reverse to restore modified locations to their original values. The procedure to do this is invoked before a thread that has been interrupted releases any of its locks. Since partial re-

[3] 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.

[4] Threads in the Jikes RVM are pseudo-preemptive: thread context-switches can happen only at pre-specified yield points inserted by the compiler.

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

sults of a computation performed by a thread executing the interrupted synchronized section are reverted before any of the locks are released, they do not become visible to any other thread, in accordance with Java execution semantics.

3.2 Discussion

Rather than using bytecode transformations, we note that an alternative strategy would implement the re-execution procedure entirely at the VM level (*ie*, all the code modifications necessary to support rollbacks would only involve the compiler and the Java run-time system). This approach simply requires that the current state of the VM (*ie*, contents of local variables, non-volatile registers, stack pointer, *etc*) be remembered upon entry to a synchronized section, and restored when a rollback 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 incorrectly computed, since a rollback 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 runtime 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 rollback 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 the design. Moreover, inserting write-barriers at the bytecode level to log changes would require optimizations to remove them to be re-implemented at this level as well, since existing optimizations are currently implemented internally by Jikes on a different intermediate representation.

The middle-ground approach that we adopted fulfills all our design requirements and was relatively easy to implement. We also managed to keep the number of modifications to the virtual machine and the compilers small, and they are mostly machine independent.[6] Thus, our implementation is easily portable.

4 Experimental evaluation

We quantify the overhead of the mechanism using a detailed micro-benchmark. We measure programs that exhibit priority inversion to verify if the increased overheads induced by our implementation are mitigated by higher overall throughput of high-priority threads.

Our algorithm to detect priority inversion is reasonably simple. A thread acquiring a monitor deposits its priority in the header of the monitor object. Before another thread can attempt acquisition of the same monitor, it checks whether its own priority is higher than the priority of the thread currently executing within the synchronized section. If it is, the scheduler initiates a context-switch and triggers rollback of the low priority thread at the next yield point. After the low-priority thread rolls back its changes and releases the monitor, the high-priority thread acquires control of the synchronized section. 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 section.

The Jikes RVM does not include a priority scheduler; threads are scheduled in a round-robin fashion. This does not affect the generality of our solution nor does it invalidate the results obtained, since problems solved by the mechanism proposed in this work 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 implemented prioritized monitor queues. 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 waiting high-priority threads.

4.1 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 sections. Our benchmark is structured so that only low-priority threads will actually employ this functionality.[7] Every thread executes 100 synchronized sections. Each synchronized section contains an inner loop executing an interleaved sequence of read and write operations. We emphasize that our micro-benchmark has been constructed to gauge overheads inherent in our technique (the costs of re-execution, logging, *etc*) and not necessarily to simulate any particular workload of a real-life application. We strived to avoid biasing the benchmark structure toward our solution 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 section directly proportional to the number of shared data operations performed within that section. 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.

[6] The exception is the insertion of the barriers in the baseline compiler, which had to be implemented in assembly language.

[7] However, updates of both low-priority and high-priority threads are logged for fairness, even though in the case of this micro-benchmark high-priority threads can never be rolled back.

- the ratio of write to read operations performed within a synchronized section – 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 equal to a single thread quantum in Jikes RVM) right before an entry to the synchronized section, to ensure random arrival of threads at the monitors guarding the sections.

Our thesis is that the total elapsed time of high-priority threads can be improved using the rollback 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 section 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 section. 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 runtime changes to interrupt and revoke execution of synchronized sections 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 section at the next yield point, rolling back any changes to shared data made from the time it began executing inside the section. The benchmark code executed on this VM is compiled using the modified version of the Jikes RVM optimizing compiler described in Section 3.1.2.

To measure the total elapsed time of high-priority threads we take the first time-stamp at the beginning of the `run()` method of every high priority thread and the second time-stamp at the end of the `run()` method of every high-priority thread. We compute the total elapsed time for all high-priority threads by calculating the time elapsed from the earliest time-stamp of the first set (*ie*, start times) to the latest time-stamp of the second set (*ie*, end times). We also present the impact that our solution has on the overall elapsed time of the entire micro-benchmark, including low-priority elapsed times; this is a simple generalization of the above description, with time-stamps taken at the beginning and end of the `run()` method for all threads, regardless of their priority.

The measurements were taken on an 800MHz Intel Pentium III (Coppermine) with 1024MB of RAM running Linux kernel version 2.4.20-13.7 (RedHat 7.0) in single-user mode. We ran each benchmark in its own invocation of the VM, 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. 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. Even in this configuration there remain some methods (*eg*, class initializers) that are still baseline compiled, in both the original and modified VMs alike.

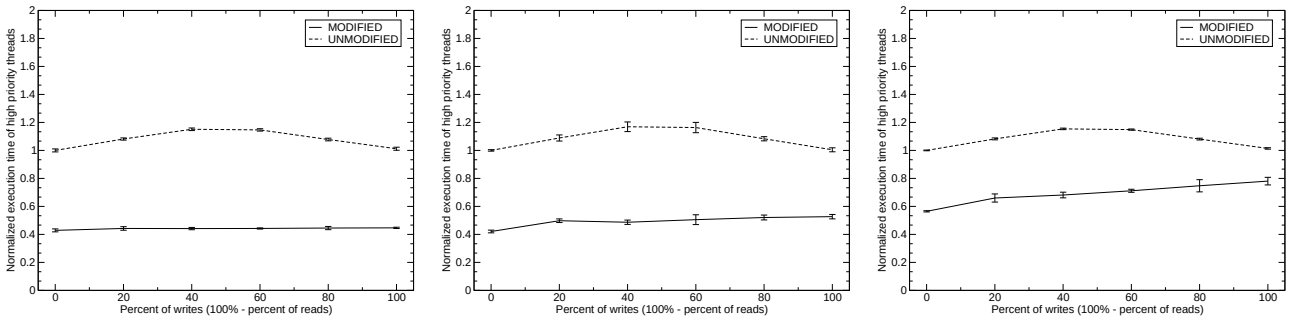
4.2 Results

Figures 5 and 6 plot elapsed times for high priority threads executed on both the modified (solid line) and unmodified (dotted line) VM, normalized with respect to the configuration executing 100% reads on an unmodified VM. In Figure 5 every high priority thread executes 100K internal iterations; in Figure 6 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 5-6 (a)(b)), our hybrid implementation improves throughput for high-priority threads by 25% to 100% over the unmodified implementation. Average elapsed-time percentage 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, with larger numbers of high-priority threads than low-priority ones, the average elapsed time of a high-priority thread is twice as fast as in the reference implementation.

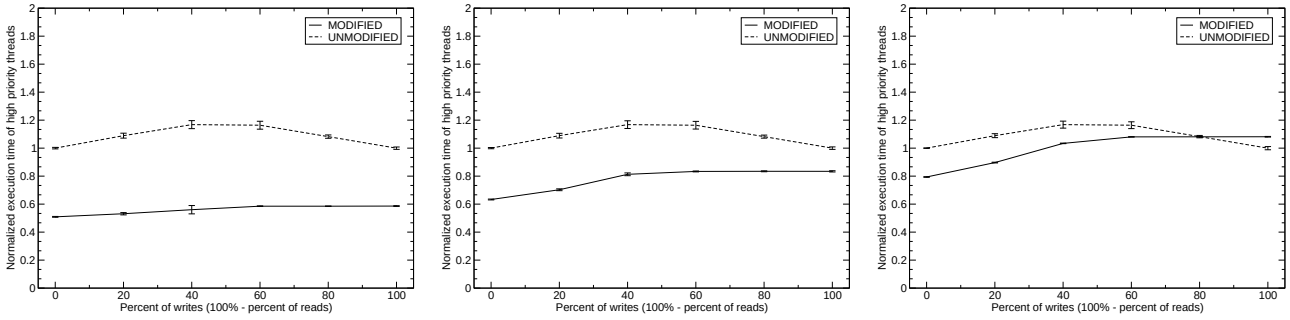
Note that the influence of different read-write ratios on overall performance is small; recall that all threads, regardless of their priority, log all updates within a synchronized section. This implies that the cost of operations related to log maintenance and rollback of partial results is also small, compared to the elapsed time of the entire benchmark. Indeed, the actual "workload" (contents of the synchronized section) 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 section is sufficiently large, the overhead of logging and rollbacks may start outweighing potential benefit. For example, in Figure 6(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 section. We believe that synchronized sec-



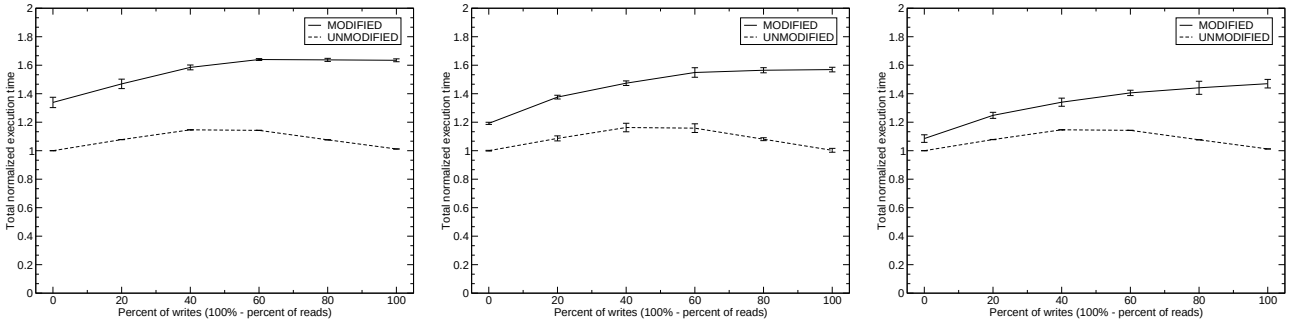
(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 5. Total time for high-priority threads, 100K iterations



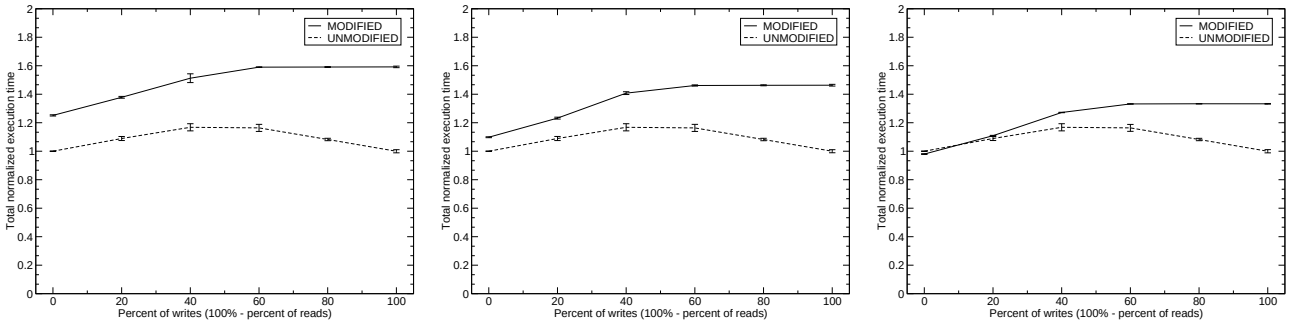
(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 6. Total time for high-priority threads, 500K iterations



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

Figure 7. 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 8. Overall time, 500K iterations

tions 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 5(c) and 6(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 rollback-enabled VM has weaker performance than the unmodified implementation, the average difference in execution time is only a few percent.

Figures 7 and 8 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 of synchronized sections, 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 mechanism for the modified VM: barriers, log maintenance, *etc*). However, if the execution of synchronized sections can be interrupted and revoked, low-priority threads executing on the modified VM will re-execute parts of their synchronized sections thus lengthening overall elapsed time. Since our focus is on lowering elapsed times of 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.

5 Related work

Priority inversion is a well-studied problems in concurrent programming. Avoiding priority inversion is especially important in mission critical or real-time applications [15, 20, 21]. Priority inheritance and priority ceiling are two well-known protocols that attempt to avoid priority inversion. The priority ceiling emulation 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. 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. Yet another alternative is to have privileged threads, for example those executing on behalf of the operating system. They can often disable interrupts or preemption that effectively prevents lower-priority threads from acquiring critical resources. Regardless of the approach, once a thread enters a synchronized section its locks cannot be summarily relinquished without potentially violating synchronization invariants. In contrast, our use of compiler-assisted rollbacks provides a *dynamic* approach to resolving priority inversion issues. Since the overheads to perform rollbacks are charged only to low-priority threads, our scheme biases throughput in favor of threads that actually require it.

Our use of rollbacks to redo computation inside synchronized sections as a result of an undesirable scheduling is reminiscent of optimistic concurrency protocols first introduced in the 1980's [13] 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, 9, 23]. Devising fast and efficient techniques to confirm that a schedule is correct remains an important topic of study.

Transactional techniques such as the kind proposed here have also been applied to a broader setting. For example, researchers have investigated lock-free data structures [18, 12] and transactional memory implementations [11, 22, 10] which generalize transactional protocols for database systems, to any concurrent system. Our solution differs from these efforts in that it is not limited to support for specific lock-free data structures, requires no hardware support and relies only on limited transactional support (requiring no resolution of conflicts between data accesses, deadlock-related situations, *etc*). It is also transparent to the programmer since it modifies only the implementation of the language and not its semantics.

Rinard [19] 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. His approach does not ensure atomic commitment of multiple variables. In the case of our solution, in contrast to a low-level facility, the code protected by monitors may span an arbitrary dynamic context.

There has been much recent interest in data race detection for Java. Some approaches [6, 3, 4] present new type systems using, for example, ownership types [5] to verify the absence of data races and deadlock. There has also been recent work on generalizing type systems to reason about higher-level atomicity properties of concurrent programs that subsume data race detection [7]. Other techniques [24] employ static analysis, such as escape analysis, along with runtime instrumentation that meters accesses to synchronized data.

The approach presented here shares similar goals with these efforts but differs in some important respects. In particular, our implementation does not rely on global static analysis (although it may benefit from it), programmer annotations, or alternative type systems. Instead, our use of rollback permits discarding effects of undesirable schedules.

6 Conclusions

We have presented a revocation-based priority inversion avoidance scheme and demonstrated its utility in improving throughput of high priority threads in a priority schedul-

ing environment. The solution proposed is relatively simple to implement, portable, and can be adopted to solve other types of problems (*eg*, deadlocks). Our techniques use compiler support that insert barriers on read and write operations to log acceses and updates to shared data, and runtime modifications to implement revocation when a preemption event occurs.

Although our preliminary experiments are encouraging, we believe there are numerous opportunities to improve the performance of our design by incorporating compiler optimizations to eliminate overheads currently incurred to deal with logging and commits. For example, read barriers on code not protected by locks could be removed if such regions were identified. We also intend to evaluate the performance of our technique for real-world applications to precisely measure the impact of our enforced non-revocability of monitors.

Acknowledgments This work is supported by the National Science Foundation under grants Nos. CCR-9711673, STI-0334141, IIS-9988637, and CCR-0085792, by the Defense Advanced Research Program Agency, and by gifts from Sun Microsystems, IBM, and NEC.

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] 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 *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Nov.). *ACM SIGPLAN Notices 34*, 10 (Oct. 1999), pp. 314–324.

[3] 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.

[4] BOYAPATI, C., AND RINARD, M. A parameterized type system for race-free Java programs. In OOPSLA'01 [17], pp. 56–69.

[5] 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.

[6] FLANAGAN, C., AND FREUND, S. N. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Vancouver, Canada, June). *ACM SIGPLAN Notices 35*, 6 (June 2000), pp. 219–232.

[7] 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.

[8] GOSLING, J., JOY, B., STEELE, JR., G., AND BRACHA, G. *The Java Language Specification*, second ed. Addison-Wesley, 2000.

[9] HERLIHY, M. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. Database Syst. 15*, 1 (1990), 96–124.

[10] 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.

[11] 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.

[12] 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.

[13] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 9*, 4 (June 1981), 213–226.

[14] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[15] LOCKE, D., SHA, L., RAJIKUMAR, R., LEHOCZKY, J., AND BURNS, G. Priority inversion and its control: An experimental investigation. In *Proceedings of the second international Workshop on Real-time Ada issues* (Moretonhampstead, Devon, England, June). 1988, pp. 39–42.

[16] MANSON, J., AND PUGH, W. JSR133: Java memory model and thread specification, 2004.

[17] *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Tampa, Florida, Oct.). *ACM SIGPLAN Notices 36*, 11 (Nov. 2001).

[18] 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.

[19] RINARD, M. Effective fine-grained synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Trans. Comput. Syst. 17*, 4 (Nov. 1999), 337–371.

[20] SCHMIDT, D. C., MUNGEE, S., FLORES-GAITAN, S., AND GOKHALE, A. S. Alleviating priority inversion and non-determinism in real-time CORBA ORB core architectures. In *IEEE Real Time Technology and Applications Symposium* (Denver, Colorado, June). 1998, pp. 92–101.

[21] 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.

[22] 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.

[23] STONEBRAKER, M., AND HELLERSTEIN, J., Eds. *Readings in Database Systems*, third ed. Morgan Kaufmann, 1998.

[24] VON PRAUN, C., AND GROSS, T. R. Object race detection. In OOPSLA'01 [17], pp. 70–82.

[25] YODAIKEN, V. Against priority inheritance. http://www.fsmlabs.com/articles/inherit/inherit.html.