# Transactional Monitors for Concurrent Objects

Adam Welc, Suresh Jagannathan, and Antony L. Hosking

Department of Computer Sciences
Purdue University
West Lafayette, IN 47906
{welc,suresh,hosking}@cs.purdue.edu

**Abstract.** *Transactional monitors* are proposed as an alternative to monitors based on mutual-exclusion synchronization for object-oriented programming languages. Transactional monitors have execution semantics similar to mutual-exclusion monitors but implement monitors as lightweight transactions that can be executed concurrently (or in parallel on multiprocessors). They alleviate many of the constraints that inhibit construction of transparently scalable and robust applications.

We undertake a detailed study of alternative implementation schemes for transactional monitors. These different schemes are tailored to different concurrent access patterns, and permit a given application to use potentially different implementations in different contexts.

We also examine the performance and scalability of these alternative approaches in the context of the Jikes Research Virtual Machine, a state-of-the-art Java implementation. We show that transactional monitors are competitive with mutual-exclusion synchronization and can outperform lock-based approaches up to five times on a wide range of workloads.

## 1 Introduction

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

A 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 others, and all updates performed by a thread within a guarded region become visible to other threads *atomically*, once the executing thread exits the region.

Guarded regions are usually implemented using mutual-exclusion locks: a thread acquires a lock before it is allowed to enter the guarded region and blocks if the lock has already been acquired by another thread. Isolation results since threads must execute the guarded region serially: only one thread at a time can be active in the region, although this serial order is not necessarily deterministic. Atomicity of updates is also achieved with

M. Odersky (Ed.): ECOOP 2004, LNCS 3086, pp. 518–541, 2004.

respect to shared data accessed within the region; updates are visible to other threads only when the current thread releases the lock.

Unfortunately, enforcing isolation and atomicity using mutual-exclusion locks suffers from a number of potentially serious drawbacks. Most importantly, locks often serve as poor abstractions since they do not help to guarantee high-level properties of concurrent programs such as atomicity or isolation that are often implicitly assumed in the specification of these programs. In other words, locks do not obviate the programmer from the responsibility of (re)structuring programs to guarantee atomicity, consistency, or isolation invariants defined in a program's specification. The mismatch between the low-level semantics of locks, and the high-level reasoning programmers should apply to define concurrent applications leads to other well-known difficulties. For example, threads waiting to acquire locks held by other threads may form cycles, resulting in deadlock. Priority inversion may result if a high-priority thread must wait to enter a guarded region because a low-priority thread is already active in it. Finally, for improved performance, code must often be specially tailored to provide adequate concurrency. 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 or leaves 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 leads to implementation complexity, and requires more memory to hold the necessary lock state.

Recognition of these issues has prompted a number of research efforts aimed at higher-level abstract notions of concurrency that omit any definition based on mutual-exclusion locks [25,24,20,19]. In this paper, we propose *transactional monitors* as an alternative to mutual exclusion for object-oriented programming languages. Transactional monitors implement 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's transaction.[1]

Our work is distinguished from previous efforts in two major respects. First, we provide a semantics and detailed exploration of alternative implementation schemes for transactional monitors, all of which enforce desired isolation and atomicity properties. These different schemes are tailored to different concurrent access patterns, and permit a given application to use potentially different transactional monitor implementations in different contexts. We focus on two specific alternatives: an approach that works well when contention for shared data is low (*eg*, mostly read-only guarded regions), and a scheme better suited to handle highly concurrent accesses with a more uniform mix of

[1] 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.

reads and updates. These alternatives reflect likely patterns of use in realistic concurrent programs.

Second, we examine the performance and scalability of these different approaches in the context of a state-of-the-art Java compiler and virtual machine, the Jikes Research Virtual Machine (RVM) [2] from IBM. Jikes RVM is an ideal platform in which to explore alternative implementations of transactional monitors, and to compare them with lock-based mutual exclusion, since Jikes 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 and benefits in using lightweight transactions as an alternative to lock-based mutual exclusion.

## 2 Overview

Unlike mutual-exclusion monitors (*eg*, synchronized blocks and methods in Java), which force threads to 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*. That is, the effects of concurrent execution of the monitor are equivalent to *some* serial schedule that would arise if no interleaving 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 the same observable behavior, transactional monitors permit a higher degree of concurrency.

Transactional monitors maintain serializability by tracking accesses to shared data 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, *atomically* performing all logged operations at once with respect to any other commit operation. If the shared data changes in such a way as to invalidate the log, the monitored code block is re-executed, and the commit retried. A log is invalidated if committing its changes would violate the serializability property of the monitored region.

For example, consider the code sample shown in Fig. 1 (using Java syntax). Thread $T_1$ computes the total balance of both checking and savings accounts. Thread $T_2$ 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, 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 $T_1$ or $T_2$ 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 $T_1$ would be correct (it would in fact be the same in both cases).

$T_1$

```
monitored (account_monitor)
{
  balance1 = checking.getBalance();
  balance2 = savings.getBalance();
  print(balance1 + balance2);
}
```

$T_2$

```
monitored (account_monitor)
{
  checking.withdraw(amount);
  savings.deposit(amount);
}
```

**Fig. 1.** Bank account example

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 Fig. 2 results in both threads successfully committing their logs – it preserves serializability since $T_2$'s withdrawal from the checking account does not compromise $T_1$'s read from the savings account. This interleaving is equivalent to a serial execution in which $T_1$ executes before $T_2$.

The interleaving presented in Fig. 3 results in $T_1$'s execution of the monitored code being aborted since $T_1$ reads an inconsistent state. Serializability is enforced by re-executing the guarded region of thread $T_1$.

| | $T_1$ | $T_2$ |
|---|---|---|
| (1) | checking.getBalance | |
| (2) | | checking.withdraw |
| (3) | savings.getBalance | |
| (4) | | savings.deposit |

**Fig. 2.** Serializable execution.

These examples illustrate several issues in formulating an implementation of transactional monitors. Threads executing within a transactional monitor must execute in *isolation*: 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 see the updates to shared state by other threads. Transactional monitor implementations must permit threads to detect state changes that violate isolation and to abort, roll-back, and restart their execution in response to such violations.

| | $T_1$ | $T_2$ |
|---|---|---|
| (1) | | checking.withdraw |
| (2) | checking.getBalance | |
| (3) | savings.getBalance | |
| (4) | | savings.deposit |

**Fig. 3.** Non-serializable execution.

In Fig. 3, the execution of thread $T_1$ is **not** isolated from the execution of thread $T_2$ since thread $T_1$ sees the effects of the withdrawal but does not see the effects of the deposit. Thus, $T_1$ is obliged to abort and re-execute its operations. In general, a thread may abort 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, 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.

## 3 Design

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. *Barrier Insertion*: A code fragment used within a guarded region to track accesses to shared data is called a *barrier*. Barriers can be inserted at the source-code level, injected into the code stream at compile time, or handled explicitly at runtime (in the case of interpreted languages).
3. *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).
4. *Re-Execution Model*: When a region guarded by a transactional monitor aborts, the updates performed by the thread in that region must be discarded. An important

design decision is whether threads perform updates directly on shared data, reverting these updates on aborts (an undo model), or whether threads perform updates on a local journal, propagating them to the corresponding (original) shared objects upon successful commit (a redo model).

5. *Nesting*: Transaction models often permit transactions to nest freely [32], 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).

Our choice for the barrier placement is to have the compiler insert the barriers (rather than, for example, inserting them at the source-level). We plan to take advantage of existing compiler optimizations (*eg*, escape analysis) to be able to remove unnecessary barrier overhead automatically (*eg*, for thread-private or immutable objects).

The decision about *when* a thread should attempt to detect serializability violation 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 at the exit from the monitor, or potentially multiple times if performed in access barriers), and reducing the amount of computation performed by a thread that may eventually abort.

Our design assumes a *redo* semantics for aborts of guarded regions. Implementations must therefore provide thread-specific redo logs to enable re-execution of guarded regions. We chose a redo semantics because the space overheads related to maintaining logs is not excessive since a log (associated with a thread object) needs to be maintained only when a thread is executing within a transactional monitor, and can be discarded upon exit from the monitor. Our design requires all updates performed within a transactional monitor to be re-directed to the log, and atomically installed in the shared (globally-visible) heap upon successful commit (no action is taken upon abort).

An alternative design might consider the use of *undo* logs in which all updates are performed directly on shared data, and reverted using information from the log upon abort. However, using undo logs can lead to *cascading aborts*[2] 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. Per-access locking also has the disadvantage of requiring deadlock detection or avoidance.

Modularity principles dictate that our design support nested transactional monitors. A given monitor region may contain a number of child monitors. Because monitors are

[2] All threads that have seen updates of a thread being aborted must be aborted as well.

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 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.

## 4 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 violation 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 towards any of these parameters, it is not clear how effectively it would perform under 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 guarded region must be re-executed.

Our current implementation does not yet include support for nested transactions.

### 4.1 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 [3], then reads and writes can operate directly over original data. To preserve correctness, the implementation must still prevent multiple non-serializable writes to objects and must disallow readers from seeing partial or inconsistent updates to the the objects performed by the writers.

To address these concerns, we define an implementation that stores the following information in the transactional monitor object:

- *writer*: uniquely identifies a thread currently executing within a given monitor that has performed writes to this object;
- *thread count*: number of threads concurrently operating within a given monitor.

*Initialization:* A thread attempting to enter the monitor must first check whether there are any active writers within the monitor. If there are no active writers, the thread can freely proceed. Otherwise, shared data is not guaranteed to be in a consistent state, and the entering thread must spin until there are no more active writers, thus achieving serializability of guarded execution.

*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 record whether writes performed by other threads which have yet to exit the monitor have taken place. A write to a shared object can occur if one of the following conditions exist:

- The writer field in the monitor object is nil, indicating there are no active writers. In this case, the current thread atomically sets the writer field, increments the thread count, 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 this object within the same monitor. The current write can proceed.

If either condition fails, the thread must re-execute the monitor.

*Conflict Detection:* In order for the shared data operations of a thread exiting a monitor to be consistent and serializable with respect to other threads, there must be no other writers still active within this monitor besides the exiting thread. It is guaranteed (by the actions taken in the write barrier) that if the exiting thread performed any writes when executing within a monitor, it is the only active writer within this monitor. If the guarded region executed by the exiting thread was read-only and there is an active writer still executing within the monitor, it implies that the exiting thread might have seen an

[3] An example of the low-contention scenario could be multiple mostly read-only threads traversing a tree-like structure or accessing a hash-table

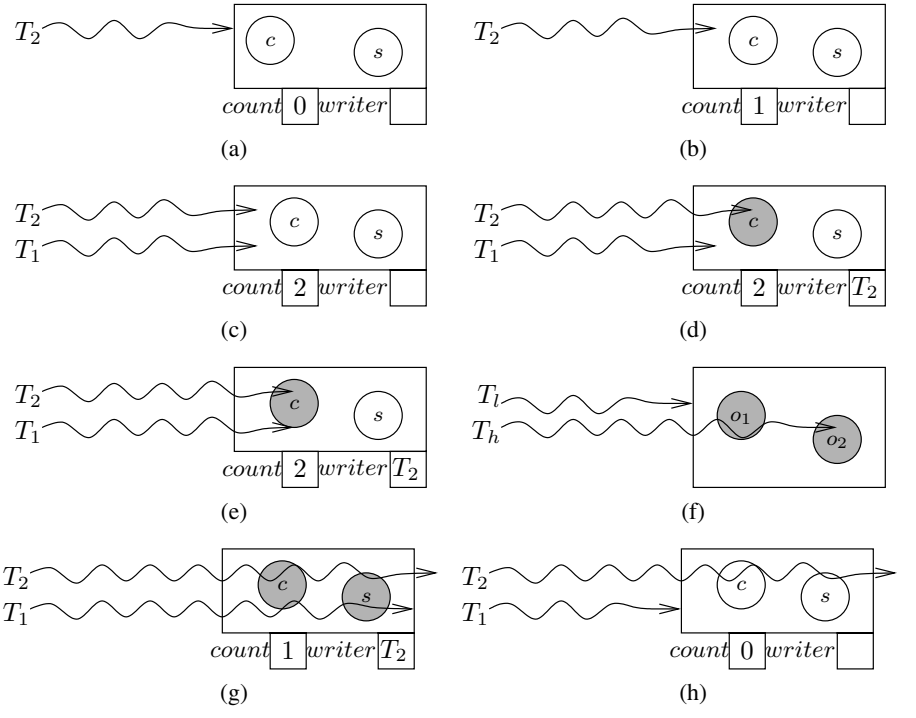



**Fig. 4.** Low contention scheme example

inconsistent state which leads to a conflict. If the execution was read-only and there are no active writers, it implies that any threads concurrently executing within the monitor have performed only reads and no conflicts were possible.

*Monitor Exit:* Any thread that exits a transactional monitor must atomically decrement the thread count. When a writer exits a monitor, it must first check whether the thread count is one. A number greater than one indicates that there are still other active threads executing within a monitor. To ensure that these threads are aware that writes have occurred when they perform conflict detection, the writer field cannot be reset. The last thread to exit the monitor as part of the monitor exit procedure will decrement the count to zero, and reset the writer field. Since there are no copies or logs, all updates are immediately visible in the original copy.

The actions performed in this scheme executing the account example from Fig. 3 is illustrated in Fig. 4, where wavy lines represent threads $T_1$ and $T_2$, circles represent objects $c$ (checking account) and $s$ (saving account), updated objects are marked gray. 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 additional information associated with the monitor: writer field (initially nil) and thread count (initially 0). In Fig. 4(a) thread $T_2$ is about to enter the monitor, which it does in Fig. 4(b) incrementing thread count. In Fig. 4(c) thread $T_1$ also enters the

monitor and increments thread count. In Fig. 4(d) thread $T_2$ updates object $c$ and sets the writer to itself. Subsequently thread $T_1$ reads object $c$ (Fig. 4(e)), thread $T_2$ updates object $s$ and exits the monitor (Fig. 4(f)) (no conflicts are detected since there were no intervening writes on behalf of other threads executing within the monitor). Thread count gets decremented but the writer cannot be reset since thread $T_1$ is still executing within the monitor. In Fig. 4(g) thread $T_1$ reads object $s$ and attempts to exit the monitor, but the writer field still points to thread $T_2$ indicating a potential conflict[4] – guarded region of thread $T_1$ must be re-executed. Since thread $T_1$ is the last one to exit the monitor, in addition to decrementing thread count it also resets the writer field.

### 4.2 High-Contention Concurrency

When there is notable contention for shared data, the previous strategy is not likely to perform well because attempts to execute multiple writes even to distinct objects result in a conflict, and subsequent aborts of all but one writer executing within the same monitor. We can relax this restriction by allowing threads to manipulate *copies* of shared objects, committing their changes when it does not conflict with other shared data operations [5]. 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 serializability of a guarded region's execution is not compromised. However, since applications tend to perform a lot more reads than writes, we decided to use a copy-on-write strategy [6] to reduce the cost of read operations (trading it for a potential loss of precision in detecting serializability violations).

In this scheme, the following information is stored in each monitor:

- *global write map*: identifies objects written by threads executing within the monitor. This map is implemented as a bitmap with a bit being set for every modified object. The mapping is conservative and multiple objects can potentially be hashed into the same bit;
- *thread count*: number of threads concurrently operating within a transactional monitor.

The monitor object also contains information about whether any thread executing within a monitor has already managed to install its updates. The global write map and thread count can be combined into one data structure with the thread count occupying the *n* lowest bits of the write map. In addition to the data stored in the monitor object, the header of every object is extended to hold the following information:

- *copies*: circular list of the object copies created by threads executing within transactional monitors (original object is the head of the list)
- *writer*: if the object is copy generated by a $T$ within a transactional monitor, this field contains a reference to $T$.

[4] This example is based on the interleaving of operations where the conflict really exists (serializability property is violated)

[5] An example of the high-contention scenario could be multiple threads traversing disjoint subtrees of a tree-like structure or accessing different buckets in a hash-table

[6] Instead of creating copies on both reads and writes

There is also the following (local) information associated with every thread:

- *local writes*: list of object copies created by a given thread when executing within a transactional monitor;
- *local read map*: identifies objects read by a given thread when executing within a monitor (implemented in the same way as the global write map), and is used for conflict detection;
- *local write map*: identifies objects written by a given thread when executing within a monitor (implemented the same way as the global write map), and is used to optimize read and write barriers.

*Initialization:* The first thread attempting to enter a monitor must initialize the monitor by initializing 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 the updates have already been installed, the remaining threads still executing within the monitor are allowed to continue their execution, but no more threads are allowed to enter the monitor (they spin) to allow for the global write map to be cleaned up (otherwise out-dated information about updates performed within the monitor could be retained for indefinite amount of time forcing entering threads to repeatedly abort). Each thread entering a monitor must also initialize its local data structures.

*Read and Write Barriers:* The barriers implement a copy-on-write semantics. The following actions are taken on writing an object:

- If the bit in the local write map representing the object is not set, *ie*, the current thread has not yet written to this object, a copy of the original object is created [7], the local write map is tagged, and the write is redirected to the copy.
- If the bit in the local write map representing the object is set, *ie*, the current thread has potentially (it is a conservative mapping because of our hash construction) written to this object, the copy is located by traversing the list of copies to find the one created by the current thread. Otherwise, a new copy is created and the write is redirected to the copy.

The following actions are taken on reading an object:

- If the bit in the local write map representing the object is not set, *ie*, the current thread has not yet written to this object, read from the original object, tag the local read map end exit the barrier.
- If the bit in the local write map representing the object is set, *ie*, the current thread has potentially written to this object, and a copy of the object created by this thread exists, the contents of this copy is read. If no such copy exists because the thread did not actually write to it, the contents of the original object is read.

[7] Creation of a copy also involves inserting this copy to the appropriate copy and local write lists.

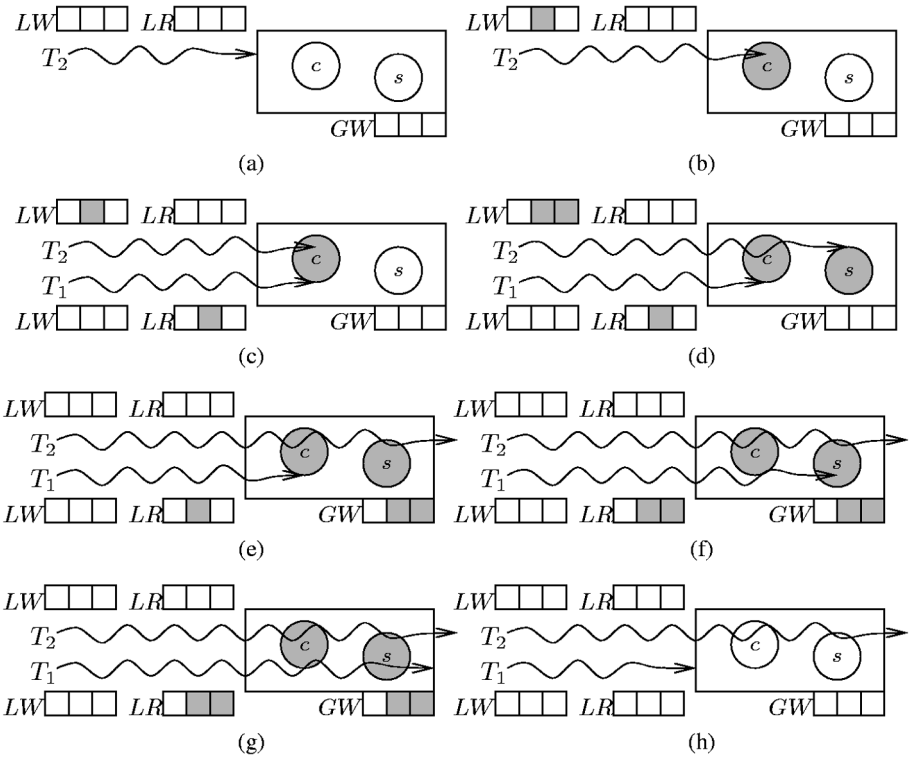



**Fig. 5.** High contention scheme example

*Conflict Detection:* When a thread exits the monitor, a conflict detection algorithm checks if the global write map associated with the monitor object is disjoint for the local read map associated with the thread. If so, it means that no reads of the current thread were interleaved with the committed writes of other threads executing within the same monitor; otherwise a potentially harmful interleaving could occur violating serializability – guarded region must be re-executed. (The global write map gets updated after a thread installs its local updates into the shared objects.)

*Monitor Exit:* If a thread is allowed to commit, all updates to copies (accessible from the local writes list) performed during monitored execution must be installed in the original objects and the local write map must be merged with the global write map to reflect writes performed by the current thread (both these operations must be performed atomically with respect to other threads potentially exiting the same monitor at the same time). Regardless of whether a thread is committing or aborting, all the lists containing copies created by this thread must be at this point updated. An exiting thread must also decrement the thread counter and free the monitor if the counter reaches zero (no active threads executing within the monitor).

The actions performed in this scheme executing the account example from Fig. 3 is illustrated in Fig. 5, where wavy lines represent threads $T_1$ and $T_2$, circles represent objects $c$ (checking account) and $s$ (saving account), updated objects are marked gray, and the box represents the dynamic scope of a common transactional monitor $account\_monitor$ guarding code regions executed by the threads. Both global write map ($GW$) associated with the monitor and local maps (local write map $LW$ and local read map $LR$) associated with each thread have three slots. Local maps above the wavy line representing thread $T_2$ belong to $T_2$ and local maps below the wavy line representing thread $T_1$ belong to $T_1$. In Fig. 5(a) thread $T_2$ is about to enter the monitor, which it does in Fig. 5(b), modifying object $c$. Object $c$ is shaded and information about the update gets reflected in the local write map of $T_2$ (we assume that object $c$ hashes into the second slot of the map). In Fig. 5(c) thread $T_1$ enters the same monitor and reads object $c$ (the read operation gets reflected in the local read map of $T_1$). In Fig. 5(d) thread $T_2$ modifies object $s$, object $s$ gets shaded and the update also gets reflected in $T_2$'s local write map (we assume that object $s$ hashes into the third slot of the map). In Fig. 5(e) thread $T_2$ exits the monitor. Since no conflicts are detected (there were no intervening writes on behalf of other threads executing within the monitor), $T_2$ installs its updates, modifies the global write map to reflect updates performed within the guarded region and resets its local maps. Thread $T_1$ subsequently reads object $s$ marking its local read map (Fig 5(f)) and attempts to exit the monitor (Fig. 5(g)). In the case of thread $T_1$ however its local read map and the global write map overlap indicating a potential conflict[8]; thus, the guarded region of thread $T_1$ must be re-executed (Fig. 5(h)). Since thread $T_1$ is the last thread to exit the monitor, in addition to reseting its local maps, it also frees the monitor by resetting the global write map.

## 5 Experimental Evaluation

We validate the effectiveness of transactional monitors in a prototype implementation for IBM's Jikes Research Virtual Machine (RVM) [2]. The Jikes RVM is a state-of-the-art Java virtual machine with performance comparable to many production virtual machines. It is itself written almost entirely in Java and is self-hosted (*ie*, it does not require another virtual machine to run). Java bytecodes in the Jikes RVM are compiled directly to machine code. The Jikes RVM's public distribution includes both a "baseline" and optimizing compiler. The "baseline" compiler performs a straightforward expansion of each individual bytecode into a corresponding sequence of assembly instructions. The optimizing compiler generates high quality code due in part to sophisticated optimizations implemented at various levels of intermediate representation, and because it uses adaptive compilation techniques [3] to selectively target code best suited for optimizations. Our transactional monitors prototype targets the Intel x86 architecture.

[8] This example is also based on the interleaving of operations where the conflict really exists and serializability invariants are violated)

### 5.1 Java-Specific Issues

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

*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 escape the thread (*eg*, a call to obtain the current system time), it can safely execute within a guarded region. In the abstract, it may be possible to provide compensation code to be invoked when a transaction aborts that will revert the effects of the native method calls executed within the 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.

*Existing Synchronization Mechanisms:* Double guarding a code fragment with both a transactional monitor and a mutual-exclusion monitor (expressed using `synchronized` methods or blocks) 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.

*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 [9]). 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. [10]

[9] This requirement is identical to the original Java execution semantics – a thread invoking wait or notify must hold the corresponding monitor.

[10] Notification modifies the shared state of a program and is therefore subject to the same visibility rules as other shared updates.

### 5.2 Compiler Support

Transactional monitors are implemented in both optimizing and "baseline" compilers. This is necessary because Jikes RVM configured to use only the optimizing compiler may still have certain methods (*eg*, class initializers) compiled by the "baseline" compiler. The implementation for both compilers is analogous. For the sake of brevity, our description here is limited to modifications to the optimizing compiler.

*Barriers:* Read and write barriers conceptually consist of two parts: (1) a check to determine if the operation occurs inside the dynamic context of a transactional monitor, and (2) if so, the actions to be undertaken to support a specific implementation strategy as described earlier. An inlined static method first checks whether any special processing of the operation is required. If the current thread is executing inside of RVM code or transactional monitors are turned off (*eg*, during RVM startup), no further action is required. Code for read and write barriers is inserted at an early stage of compilation allowing the compiler to apply appropriate optimizations during subsequent compilation stages.

*Re-execution of a Guarded Region:* When a thread attempts to commit its updates within a region guarded by a transactional monitor, and a conflict is detected, the thread must abort its changes and re-execute the region. Our implementation adapts the Jikes RVM exception handling mechanism to return control to the beginning of the aborted region and uses bytecode rewriting[11] to save program state (values of local variables and method parameters) for restoration on re-execution. Each code region guarded by a transactional monitor is wrapped within an exception scope that catches an internal *rollback* exception. The rollback exception is thrown internally by the RVM, but the code to catch it (implementing re-execution) is injected into the bytecode stream. We also modify the compiler and run-time system to suppress generation (and invocation) of "default" exception handlers during a rollback operation. The "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 the requirement that an aborted synchronized block produce no side-effects.

### 5.3 Benchmark

To evaluate the performance of the prototype implementation, we chose a multi-threaded version of the OO7 object operations benchmark [14], originally developed in the database community. Our incarnation of OO7 benchmark 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 runtime 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

[11] We use the Bytecode Engineering Library (BCEL) from Apache for this purpose.

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.

The benchmark operates on a synthetic design database consisting of a set of composite parts (see Fig. 6). Each composite part consists of a graph of atomic parts and a document object. Composite parts are arranged in an assembly hierarchy, called a module. Each assembly contains either composite parts (base assemblies) or other assemblies (composite assemblies).

The multi-threaded workload consists of multiple threads running a set of parameterized traversals composed of primitive operations. A traversal chooses a single path through the assembly hierarchy and at the composite part level randomly chooses a fixed number of composite parts to visit. When the traversal reaches the composite part, it has two choices: (a) it may perform read-only traversal of a graph of atomic parts; or, (b) it may perform read-write traversal of a graph of atomic parts, swapping certain scalar fields in each atomic part visited. To foster some degree of interesting interleaving and contention, the benchmark defines a parameter that allows overhead to be added to read operations to increase the time spent performing traversals.

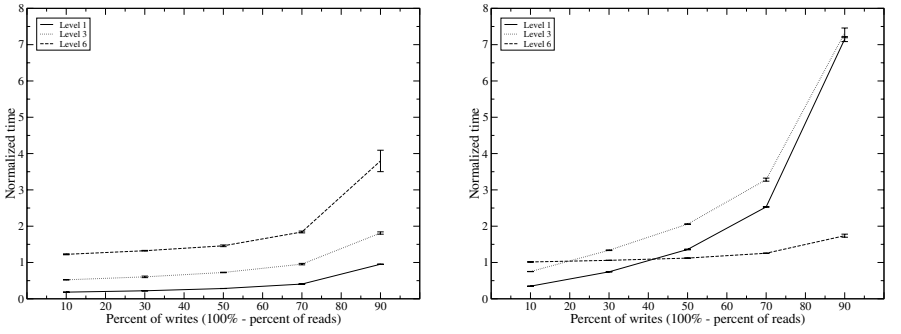
Our implementation of OO7 conforms to the standard OO7 database specification. Our traversals differ from the original OO7 traversals in allowing multiple composite parts to be visited during a single traversal rather than just one as in the original specification, and in allowing entry of monitors at various levels of the database hierarchy.

| 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 |

**Fig. 6.** Component organization of the OO7 benchmark.

### 5.4 Measurements

Our measurements were taken 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,




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

**Fig. 7.** Normalized execution time for 64 threads running on 8 processors

in which the benchmark classes are loaded and compiled, to eliminate the overheads of compilation.

When running the benchmarks we varied the following parameters:

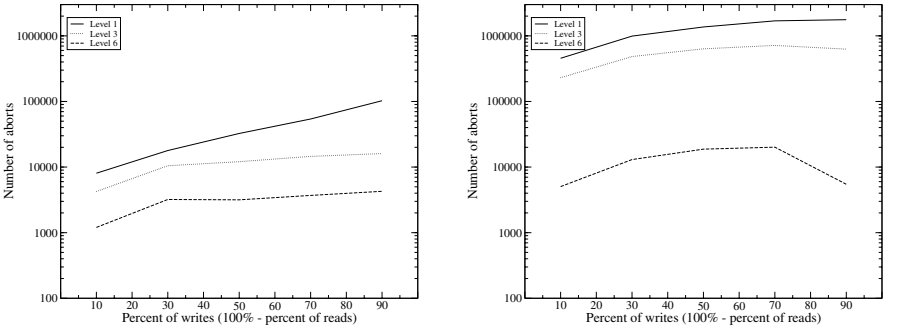
- number of threads competing for shared data access along with the number of processors executing the threads: we ran $P * 8$ threads (where $P$ is the number of processors) for $P = 1, 2, 4, 8$.
- ratio of shared reads to shared writes: from 10% shared reads and 90% shared writes (mostly read-only guarded regions) to 90% shared reads and 10% shared writes (mostly write-only guarded regions)
- level of the benchmark database at which monitors were entered: level one (module level), level three (second layer of composite parts) and level six (fifth layer of composite parts)

Every thread performed 1000 traversals (entered 1000 guarded regions) and visited 2000k atomic parts during each iteration.

### 5.5 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




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

**Fig. 8.** Total number of aborts for 64 threads running on 8 processors

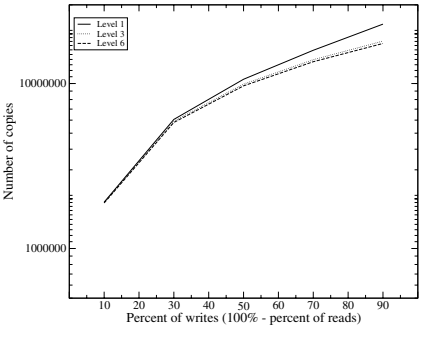
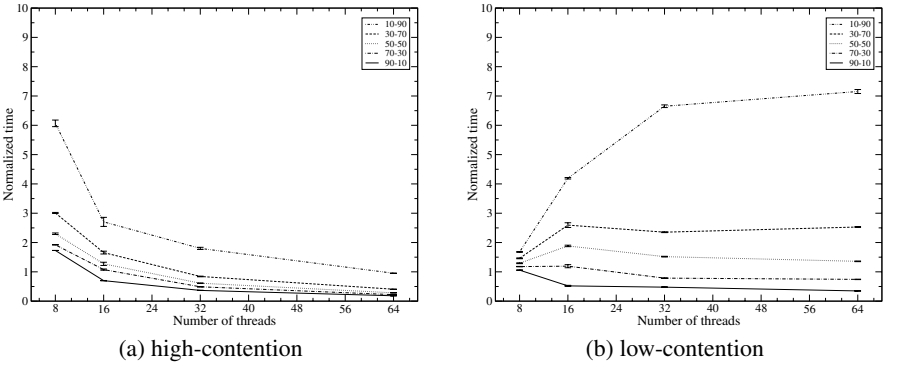



**Fig. 9.** Total number of copies created for 64 threads running on 8 processors

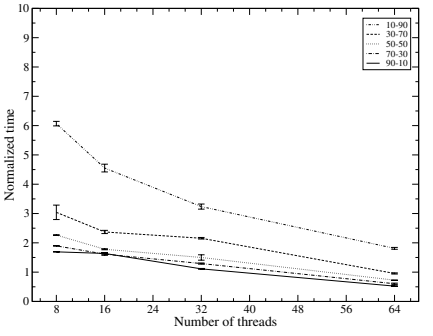



(a) high-contention (b) low-contention

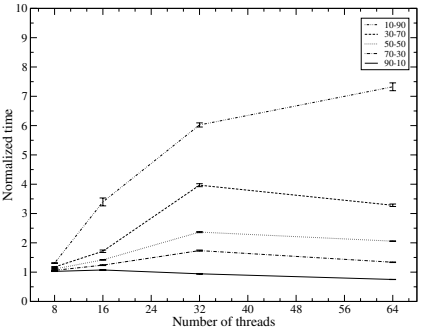
**Fig. 10.** Normalized execution times – monitor entries at level 1

with the amount of contention on entering monitors [12]. Fig. 7 plots execution time for 64 threads running on 8 processors for the high-contention scheme (Fig. 7(a)) and

[12] Threads contend on entering a monitor only if they enter the *same* monitor

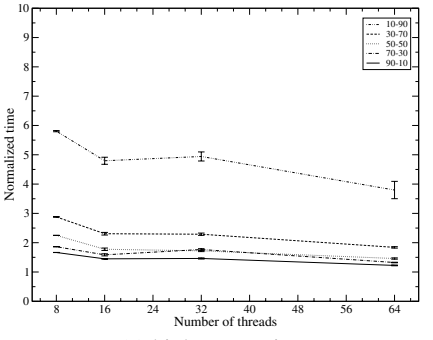



(a) high-contention

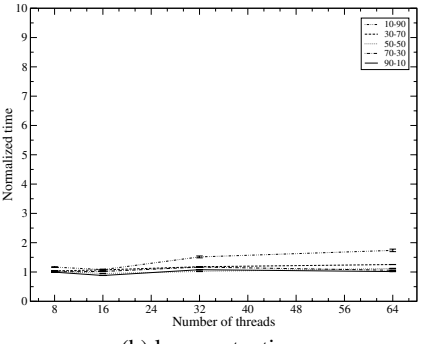



(b) low-contention

**Fig. 11.** Normalized execution times – monitor entries at level 3




(a) high-contention




(b) low-contention

**Fig. 12.** Normalized execution times – monitor entries at level 6

low-contention scheme (Fig. 7(b)) normalized to the execution time for standard mutual-exclusion monitors [13], 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 creates any reasonable contention (and thus 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 Fig. 7(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 Fig. 7(b): it outperforms mutual-exclusion monitors for configurations where write operations constitute 30%

[13] 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.

of all data operations (low contention on shared data accesses). The total number of aborts across all iterations for both high-contention scheme and low-contention scheme appears in Fig. 8(a-b). The total number of copies created across all iterations for the high-contention scheme appears in Fig. 9. The remaining graphs illustrate the scalability of both schemes by plotting normalized execution times for the high-contention scheme (Figs. 10-12(a)) and low-contention scheme (Figs. 10-12(b)) when varying the number of threads (and processors) for monitor entries placed at levels one, three, and six (Figs. 10-12, respectively).

## 6 Related Work

Several recent efforts explore alternatives to lock-based concurrent programming. Harris *et al* [24] 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 section 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 [35,28] and transactional memory [26,38] are also closely related to transactional monitors. Herlihy *et al* [25] present a solution closest in spirit to transactional monitors. They introduce an 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 [35] also shares similar goals with our work, but their implementation relies on hardware support.

Rinard [37] 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.

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 [20,19]. Other techniques [41] employ static analyses such as escape analysis along with runtime 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 synchronization sections, 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.

Incorporating explicit concurrency abstractions within high-level languages has a long history [22,23,18,9,36], as does deriving parallelism from unannotated programs either through compiler analysis [31] or through explicit annotations and pragmas [39]. Our ideas differ from these efforts insofar as we are concerned with providing abstractions that simplify the complexity of locking and synchronization. Although we do not elaborate on this point in this paper, we believe transactional monitors can be generalized to serve as a building block upon which higher-level concurrency abstractions can be defined and implemented. We believe such an approach might profitably be used as part of a Java-centric operating system.

There have been several attempts to reduce locking overhead in Java. Agesen *et al* [1] 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 [30,16,21]. 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 [10] that encode transaction-style semantics into the pi-calculus and its variants. The work of Busi, Gorrieri and Zavattaro [11] and Busi and Zavattaro [13] formalize the semantics of JavaSpaces, a transactional coordination language for Linda, and discuss the semantics of important extensions such as leasing [12]. 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 [40].

## References


1. Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In OOPSLA'99 [34], pages 207–222.
2. Bowen Alpern, C. R. Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice C. Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In OOPSLA'99 [34], pages 314–324.
3. Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 35, pages 47–65, October 2000.
4. David Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 33, pages 258–268, May 1998.
5. Martin Berger and Kohei Honda. The Two-Phase Commitment Protocol in an Extended pi-Calculus. In Luca Aceto and Bjorn Victor, editors, *Electronic Notes in Theoretical Computer Science*, volume 39. Elsevier, 2003.
6. Andrew Black, Vincent Cremet, Rachid Guerraoui, and Martin Odersky. An equational theory for transactions. Technical Report CSE 03-007, Department of Computer Science, OGI School of Science and Engineering, 2003.
7. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 37, pages 211–230, November 2002.
8. Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In OOPSLA'01 [33], pages 56–69.
9. Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Pena. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*. IEEE Press, 1997.
10. R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in the join calculus. In Lubos Brim, Mojmír Kretíský Petr Jancar, and Antonín Kucera, editors, *International Conferenec on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 321–337, 2002.
11. Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the Semantics of JavaSpaces. In *Formal Methods for Open Object-Based Distributed Systems IV*, volume 177. Kluwer, 2000.
12. Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Temporary Data in Shared Dataspace Coordination Languages. In *FOSSACS'01*, pages 121–136. Springer-Verlag, 2001.
13. Nadia Busi and Gianluigi Zavattaro. On the serializability of transactions in JavaSpaces. In *Proc. of International Workshop on Concurrency and Coordination (CONCOORD'01)*. Electronic Notes in Theoretical Computer Science 54, Elsevier, 2001.
14. Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the ACM International Conference on Management of Data*, volume 22, pages 12–21, June 1993.
15. Tom Choithia and Dominic Duggan. Abstractions for fault-tolerant computing. Technical Report 2003-3, Department of Computer Science, Stevens Institute of Technology, 2003.
16. Panos Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.
17. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 33, pages 48–64, October 1998.

18. Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimizations. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 209–220, 1995.
19. Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 35, pages 219–232, June 2000.
20. Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 1–12, 2003.
21. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Data Management Systems. Morgan Kaufmann, 1993.
22. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
23. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.
24. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 38, pages 388–402, November 2003.
25. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.
26. Antony L. Hosking and J. Eliot B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 28, pages 288–303, October 1993.
27. Suresh Jagannathan and Jan Vitek. Optimistic Concurrency Semantics for Transactions in Coordination Languages. In *Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 183–198, 2004.
28. E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical report, Lawrence Livermore National Laboratories, 1987.
29. H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 9(4):213–226, June 1981.
30. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.
31. G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications*, 2000. Special issue on High Level Models and Languages for Parallel Processing.
32. J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.
33. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 36, November 2001.
34. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 34, October 1999.
35. Ravi Rajwar and James R Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 37, pages 5–17, October 2002.
36. John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
37. Martin Rinard. Effective fine-grained synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, November 1999.

38. Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.
39. Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithms + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.
40. Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking. A semantic framework for designer transactions. In David E. Schmidt, editor, *Proceedings of the European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 249–263, 2004.
41. Christoph von Praun and Thomas R. Gross. Object race detection. In OOPSLA'01 [33], pages 70–82.