Transparently Reconciling Transactions with Locking for Java Synchronization

Adam Welc, Antony L. Hosking, and Suresh Jagannathan

Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA
{welc, hosking, suresh}@cs.purdue.edu

Abstract. Concurrent data accesses in high-level languages like Java and C# are typically mediated using mutual-exclusion locks. Threads use locks to *guard* the operations performed while the lock is held, so that the lock's guarded operations can never be interleaved with operations of other threads that are guarded by the same lock. This way both *atomicity* and *isolation* properties of a thread's guarded operations are enforced. Recent proposals recognize that these properties can also be enforced by concurrency control protocols that avoid well-known problems associated with locking, by transplanting notions of *transactions* found in database systems to a programming language context. While higher-level than locks, software transactions incur significant implementation overhead. This overhead cannot be easily masked when there is little contention on the operations being guarded.

We show how mutual-exclusion locks and transactions can be reconciled transparently within Java's monitor abstraction. We have implemented monitors for Java that execute using locks when contention is low and switch over to transactions when concurrent attempts to enter the monitor are detected. We formally argue the correctness of our solution with respect to Java's execution semantics and provide a detailed performance evaluation for different workloads and varying levels of contention. We demonstrate that our implementation has low overheads in the uncontended case (7% on average) and that significant performance improvements (up to $3\times$) can be achieved from running contended monitors transactionally.

1 Introduction

There has been much recent interest in new concurrency abstractions for high-level languages like Java and C#. These efforts are motivated by the fact that concurrent programming in such languages currently requires programmers to make careful use of mutual-exclusion locks to mediate access to shared data. Threads use locks to *guard* the operations performed while the lock is held, so that the lock's guarded operations can never be interleaved with operations of other threads that are guarded by the same lock. Rather, threads attempting to execute a given guarded sequence of operations will execute the entire sequence serially, without interruption, one thread at a time. In this way, locks, when used properly, can enforce both *atomicity* of their guarded operations (they execute as a single unit, without interruption by operations of other threads that

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 148–173, 2006.

are guarded by the same lock), and *isolation* from the side-effects of all operations by other threads guarded by the same lock.

Unfortunately, synchronizing threads using locks is notoriously difficult and error-prone. Undersynchronizing leads to safety violations such as race conditions. Even when there are no race conditions, it is still easy to mistakenly violate atomicity guarantees [14]. Oversynchronizing impedes concurrency, which can degrade performance even to the point of deadlock. To improve concurrency, some languages provide lower-level synchronization primitives such as *shared* (i.e., read-only) locks in addition to the traditional mutual-exclusion (i.e., read-write) locks. Correctly using these lower-level locking primitives requires even great care by programmers to understand thread interactions on shared data.

Recent proposals recognize that properties such as *atomicity* and *isolation* can be enforced by concurrency control protocols that avoid the problems of locking, by transplanting notions of *transactions* found in database systems to the programming language context [17, 20, 36]. Concurrency control protocols ensure *atomicity* and *isolation* of operations performed within a transaction while permitting concurrency by allowing the operations of different transactions to be interleaved only if the resulting schedule is *serializable*: the transactions (and their constituent operations) *appear* to execute in some serial order. Any transaction that might violate serializability is aborted in mid-execution, its effects are revoked, and it is retried. Atomicity is a powerful abstraction, permitting programmers more easily to reason about the effects of concurrent programs independently of arbitrary interleavings, while avoiding problems such as deadlock and priority inversion. Moreover, transactions relieve programmers of the need for careful (and error-prone) placement of locks such that concurrency is not unnecessarily impeded while correctness is maintained. Thus, transactions promote programmability by reducing the burden on programmers to resolve the tension between fine-grained locking for performance and coarse-grained locking for correctness.

Meanwhile, there is comprehensive empirical evidence that programmers almost always use mutual-exclusion locks to enforce properties of atomicity and isolation [14]. Thus, making transaction-like concurrency abstractions available to programmers is generating intense interest. Nevertheless, lock-based programs are unlikely to disappear any time soon. Certainly, there is much legacy code (including widespread use of standard libraries) that utilizes mutual-exclusion locks. Moreover, locks are extremely efficient when contention for them is low – in many cases, acquiring/releasing an uncontended lock is as cheap as setting/clearing a bit using atomic memory operations such as compare-and-swap. In contrast, transactional concurrency control protocols require much more complicated tracking of operations performed within the transaction as well as validation of those operations before the transaction can finish. Given that transaction-based schemes impose such overheads, many programmers will continue to program using exclusion locks, especially when the likelihood of contention is low. The advantages of transactional execution (i.e., improved concurrency, deadlock-freedom) accrue only when contention would otherwise impede concurrency and serializability violations are low.

These tradeoffs argue for consideration of a hybrid approach, where existing concurrency abstractions (such as Java's monitors) used for atomicity and isolation can be

mediated both by locks and transactions. In fact, whether threads entering a monitor acquire a lock or execute transactionally, so long as the language-defined properties of the monitor are enforced, all is well from the programmer's perspective. Dynamically choosing which style of execution to use based on observed contention for the monitor permits the best of both worlds: low-cost locking when contention is low, and improved concurrency using transactions when multiple threads attempt to execute concurrently within the monitor.

Complicating this situation is the issue of nesting, which poses both semantic and implementation difficulties. When a nested transaction completes, isolation semantics for transactions mandate that its effects are not usually globally visible until the outermost transaction in which it runs successfully commits. Such nesting is referred to as *closed*, and represents the purest expression of nested transactions as preserving atomicity and isolation of their effects. In contrast, Java monitors expressed as synchronized methods/blocks reveal all prior effects upon exit, even if the synchronized execution is nested inside another monitor. Obtaining a meaningful reconciliation of locks with transactions requires addressing this issue.

Our Contribution

In this paper, we describe how locks and transactions can be *transparently* reconciled within Java's monitor abstraction. We have implemented monitors for Java that execute using locks when contention is low and switch over to transactions when concurrent attempts to enter the monitor are detected. Our implementation is for the Jikes Research Virtual Machine (RVM). To our knowledge, ours is the first attempt to consider hybrid execution of Java monitors using both mutual-exclusion and transactions within the same program.

Our treatment is transparent to applications: programs continue to use the standard Java synchronization primitives to express the usual constraints on concurrent executions. A synchronized method/block may execute transactionally even if it was previously executed using lock-based mutual exclusion, and vice versa. Transactional execution dynamically toggles back to mutual-exclusion whenever aborting a given transaction becomes infeasible, such as at native method calls. In both cases, hybrid execution does not violate Java semantics, and serves only to improve performance.

We make the following contributions:

1. The design and implementation of a Java run-time system that supports implementation of Java monitors (i.e., synchronized methods/blocks) using both mutual-exclusion and software transactions based on optimistic concurrency. A given monitor will execute using either concurrency control mechanism depending on its contention profile.
2. An efficient implementation of monitors as closed nested transactions. We introduce a new mechanism called *delegation* that significantly reduces the overhead of nested transactions when contention is low. Support for delegation is provided through extensions to the virtual machine and run-time system.
3. A formal semantics that defines safety criteria under which mutual exclusion and transactions can co-exist. We show that for programs that conform to prevalent atomicity idioms, Java monitors can be realized using either transactions or mutual-

exclusion with no change in observable behavior. In this way, we resolve the apparent mismatch in the visibility of the effects of Java monitors versus closed nested transactions.

4. A detailed implementation study that quantifies the overheads of our approach. We show that over a range of single-threaded benchmarks the overheads necessary to support hybrid execution (i.e., read barriers, meta-data information on object headers, etc.) is small, averaging less than 10%. We also present performance results on a comprehensive synthetic benchmark that show how transactions that co-exist with mutual-exclusion locks lead to clear run-time improvements over mutual-exclusion only and transaction-only non-hybrid implementations.

2 A Core Language

To examine notions of safety with respect to transactions and mutual exclusion, we define a two-tiered semantics for a simple dynamically-typed call-by value language similar to Classic Java [16] extended with threads and synchronization. The first tier describes how programs written in this calculus are evaluated to yield a *schedule* that defines a sequence of possible thread interleavings, and a memory model that reflects how and when updates to shared data performed by one thread are reflected in another. The second tier defines constraints used to determine whether a schedule is safe based on a specific interpretation of what it means to protect access to shared data; this tier thus captures the behavior of specific concurrency control mechanisms.

Before describing the semantics, we first introduce the language informally (see Figure 1). In the following, we take metavariables `L` to range over class declarations, `C` to range over class names, `t` to denote thread identifiers, `M` to range over methods, `m` to range over method names, `f` and `x` to range over fields and parameters, respectively, ℓ to range over locations, and `v` to range over values. We use P for process terms, and `e` for expressions.

SYNTAX:

$$
\begin{array}{l}
P ::= (P \mid P) \mid \mathtt{t}[\mathtt{e}] \\
\mathtt{L} ::= \mathtt{class\ C}\,\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \\
\mathtt{M} ::= \mathtt{m}(\overline{\mathtt{x}})\,\{\,\mathtt{e}\,\} \\
\mathtt{e} ::= \mathtt{x} \mid \ell \mid \mathtt{this} \mid \mathtt{e.f} \mid \mathtt{e.f} := \mathtt{e} \mid \mathtt{new\ C}() \\
\quad \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{let\ x} = \mathtt{e\ in\ e\ end} \mid \mathtt{guard}\,\{\mathtt{e}\}\,\mathtt{e} \\
\quad \mid \mathtt{spawn}\,(\mathtt{e})
\end{array}
$$

Fig. 1. A simple call-by-value object-based concurrent language

A program defines a collection of class definitions, and a collection of processes. Classes are all uniquely named, and define a collection of instance fields and instance methods which operate over these fields. Every method consists of an expression whose value is returned as the result of a call to that method. Every class has a unique (nullary) constructor to initialize object fields. Expressions can read the contents of a field, store a new value into an instance field, create a new object, perform a method call, define local bindings to enforce sequencing of actions, or guard the evaluation of a subexpression.

To evaluate an expression of the form, $\texttt{guard}\{\texttt{e}_1\}\ \texttt{e}$, $\texttt{e}_1$ is first evaluated to yield a reference ℓ; we refer to ℓ as a *monitor*. A monitor acts as a locus of contention, and mediates the execution of the guard body. When contention is restricted to a single thread, the monitor behaves like a mutual exclusion lock. When contention generalizes to several threads, the monitor helps to mediate the execution of these threads within the guard body by enforcing serializability on their actions.

Mutual exclusion results when monitor contention is restricted to a single thread. In contrast, transactions can be used to allow multiple threads to execute concurrently within the same region. In this sense, a transaction defines the set of object and field accesses made by a thread within a guarded region. When a thread exits a region, it consults the monitor to determine if its transaction is serializable with the transactions of other threads that have executed within the same region. If so, the transaction is allowed to commit, and its accesses are available for the monitor to mediate the execution of future transactions in this region; if not, the transaction aborts, and the thread must start a new transaction for this region.

Since we are interested in transparently using either of these protocols, two obvious questions arise: (1) when is it *correct* to have a program use mixed-mode execution for its guarded regions; (2) when is it *profitable* to do so? We address the first question in the following section, and the second in Section 5.

3 Semantics

The semantics of the language are given in Figure 2. A value is either the distinguished symbol `null`, a location, or an object $\texttt{C}(\overline{\ell})$, that denotes an instance of class `C`, in which field $\texttt{f}_\texttt{i}$ has value ℓ_i.

In the following, we use over-bar to represent a finite ordered sequence, for instance, $\overline{\texttt{f}}$ represents $\texttt{f}_1\,\texttt{f}_2\,\ldots\,\texttt{f}_n$. The term $\overline{\alpha}\alpha$ denotes the extension of the sequence $\overline{\alpha}$ with a single element α, and $\overline{\alpha}\,\overline{\alpha}'$ denotes sequence concatenation, $S.\textsc{op}_\texttt{t}$ denotes the extension of schedule S with operation $\textsc{op}_\texttt{t}$. Given schedules S and S', we write $S \preceq S'$ if S is a subsequence of S'.

Program evaluation and schedule construction is specified by a reduction relation, $P, \Delta, \Gamma, S \implies P', \Delta', \Gamma', S'$ that maps program states to new program states. A state consists of a collection of evaluating processes (P), a thread store (Δ) that maps threads to a local cache, a global store (Γ) that maps locations to values, and a schedule (S) that defines a collection of interleaved actions. This relation is defined up to congruence of processes ($P|P' = P'|P$, etc.). An auxiliary relation $\rightsquigarrow_\texttt{t}$ is used to describe reduction steps performed by a specific thread `t`. Actions that are recorded by a schedule are those that read and write locations, and those that acquire and release locks, the latter generated as part of guard evaluation. Informally, threads evaluate expressions using their local cache, loading and flushing their cache at synchronization points defined by guard expressions. These semantics roughly correspond to a release consistency memory model similar to the Java memory model [22].

The term $\texttt{t}[\mathcal{E}[\texttt{e}]]$ evaluated in a reduction rule is identified by the thread `t` in which it occurs; thus $\mathcal{E}^\texttt{t}_P[\texttt{e}]$ denotes a collection of processes containing a process with thread

PROGRAM STATES

$$
\begin{array}{rrl}
\mathtt{t} \in \mathit{Tid} & & \\
P \in \mathit{Process} & & \\
\mathtt{x} \in \mathit{Var} & & \\
\ell \in \mathit{Loc} & & \\
\mathtt{v} \in \mathit{Val} & = & \mathtt{null} \mid \mathtt{C}(\overline{\ell}) \mid \ell \\
\sigma \in \mathit{Store} & = & \mathit{Loc} \to \mathit{Val} \\
\Gamma \in \mathit{SMap} & = & \mathit{Loc} \to \mathit{Store} \\
\Delta \in \mathit{TStore} & = & \mathit{Tid} \to \mathit{Store} \\
\mathrm{OP}_{\mathtt{t}}^{\Gamma}\,\ell, \mathrm{OP}_{\mathtt{t}}\,\ell \in \mathit{Ops} & = & \{\mathbf{rd}, \mathbf{wr}\} \times \mathit{Tid} \times \mathit{Loc} + \\
 & & \{\mathbf{acq}, \mathbf{rel}\} \times \mathit{Tid} \times \mathit{Loc} \times \mathit{SMap} \\
S \in \mathit{Schedule} & = & \mathit{Ops}^* \\
\Lambda \in \mathit{State} & = & \mathit{Process} \times \mathit{Store} \times \mathit{Schedule}
\end{array}
$$

EVALUATION CONTEXTS

$$
\begin{array}{rl}
\mathcal{E} ::= & \bullet \mid \mathcal{E}.\mathtt{f} := \mathtt{e} \mid \ell.\mathtt{f} := \mathcal{E} \\
 & \mid \mathcal{E}.\mathtt{m}(\overline{\mathtt{e}}) \mid \ell.\mathtt{m}(\overline{\ell}\ \mathcal{E}\ \overline{\mathtt{e}}) \\
 & \mid \mathtt{let\ x\ =}\ \mathcal{E}\ \mathtt{in\ e\ end} \\
 & \mid \mathtt{guard}\,\{\mathcal{E}\}\,\mathtt{e}
\end{array}
$$

$$\mathcal{E}_P^{\mathtt{t}}[\mathtt{e}] ::= P \mid \mathtt{t}[\mathcal{E}[\mathtt{e}]]$$

SEQUENTIAL EVALUATION RULES

$$\frac{}{\mathtt{let\ x\ =\ v\ in\ e\ end}, \sigma, S \rightsquigarrow_{\mathtt{t}} \mathtt{e}[\mathtt{v}/\mathtt{x}], \sigma, S}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}) \quad \sigma(\ell) = \mathtt{C}(\overline{\ell})}{\ell.\mathtt{m}(\overline{\mathtt{v}}), \sigma, S \rightsquigarrow_{\mathtt{t}} [\,{}^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}},\ {}^{\ell}/_{\mathtt{this}}]\mathtt{e}, \sigma, S}$$

$$\frac{\mathit{field}(\mathtt{C}) = \overline{\mathtt{f}} \quad \sigma(\ell) = \mathtt{C}(\overline{\ell}) \quad S' = S \,.\, \mathbf{rd}_{\mathtt{t}}\,\ell}{\ell.\mathtt{f}_i, \sigma, S \rightsquigarrow_{\mathtt{t}} \ell_i, \sigma, S'}$$

$$\frac{\sigma(\ell) = \mathtt{C}(\overline{\ell''}) \quad \sigma(\ell') = \mathtt{v} \quad \sigma' = \sigma[\ell_i'' \mapsto \mathtt{v}] \quad S' = S \,.\, \mathbf{rd}_{\mathtt{t}}\,\ell' \,.\, \mathbf{wr}_{\mathtt{t}}\,\ell_i''}{\ell.\mathtt{f}_i := \ell', \sigma, S \rightsquigarrow_{\mathtt{t}} \ell', \sigma', S'}$$

$$\frac{\ell', \overline{\ell}\ \text{fresh} \quad \sigma' = \sigma[\ell' \mapsto \mathtt{C}(\overline{\ell}), \overline{\ell} \mapsto \mathtt{null}] \quad S' = S \,.\, \mathbf{wr}_{\mathtt{t}}\,\ell_1 \,.\, \ldots \mathbf{wr}_{\mathtt{t}}\,\ell_n \,.\, \mathbf{wr}_{\mathtt{t}}\,\ell' \quad \ell_1, \ldots, \ell_n \in \overline{\ell}}{\mathtt{new\ C}(), \sigma, S \rightsquigarrow_{\mathtt{t}} \ell', \sigma', S'}$$

GLOBAL EVALUATION RULES

$$\frac{\Delta(\mathtt{t}) = \sigma \quad e, \sigma, S \rightsquigarrow_{\mathtt{t},\Gamma} e', \sigma', S'}{\mathcal{E}_P^{\mathtt{t}}[\mathtt{e}], \Delta, \Gamma, S \implies \mathcal{E}_P^{\mathtt{t}}[\mathtt{e}'], \Delta[\mathtt{t} \mapsto \sigma'], \Gamma, S'}$$

$$\frac{\begin{array}{c}\sigma = \Delta(\mathtt{t}) \quad \sigma' = \sigma \circ \Gamma(\ell) \quad \Delta' = \Delta[\mathtt{t} \mapsto \sigma'] \\ \ell \notin \mathit{lockset}(S, \mathtt{t}) \\ \mathcal{E}_P^{\mathtt{t}}[\mathtt{e}], \Delta', \Gamma, \phi \Longrightarrow^* P' \mid \mathtt{t}[\mathtt{v}], \Delta'', \Gamma', S' \\ \Gamma'' = \Gamma'[\ell \mapsto \Gamma'(\ell) \circ \Delta''(\mathtt{t})]\end{array}}{\begin{array}{c}\mathcal{E}_P^{\mathtt{t}}[\mathtt{guard}\,\{\ell\}\,\mathtt{e}], \Delta, \Gamma, S \\ \Longrightarrow \mathcal{E}_{P'}^{\mathtt{t}}[\mathtt{v}], \Delta'', \Gamma'', S.\mathbf{acq}_{\mathtt{t}}^{\Gamma}\,\ell.S'.\mathbf{rel}_{\mathtt{t}}^{\Gamma'}\,\ell\end{array}}$$

$$\frac{\mathtt{t}'\ \text{fresh} \quad \Delta' = \Delta[\mathtt{t}' \mapsto \Delta(\mathtt{t})] \quad P' = P \mid \mathtt{t}'[\mathtt{e}]}{\mathcal{E}_P^{\mathtt{t}}[\mathtt{spawn}\ (\mathtt{e})], \Delta, \Gamma, S \implies \mathcal{E}_{P'}^{\mathtt{t}}[\mathtt{null}], \Delta', \Gamma, S}$$

Fig. 2. Semantics

identifier t executing expression e with context $\mathcal{E}$. The expression "picked" for evaluation is determined by the structure of evaluation contexts.

Most of the rules are standard: holes in contexts can be replaced by the value of the expression substituted for the hole, let expressions substitute the value of the bound variable in their body. Method invocation binds the variable this to the current receiver object, in addition to binding actuals to parameters, and evaluates the method body in this augmented environment. Read and write operations augment the schedule

in the obvious way. Constructor application returns a reference to a new object whose fields are initialized to `null`.

To evaluate expression `e` within a separate thread, we first associate the new thread with a fresh thread identifier, set the thread's local store to be the current local store of its parent, and begin evaluation of `e` using an empty context.

Let ℓ be the monitor for a guard expression. Before evaluating the body, the local store for the thread evaluating the guard is updated to load the current contents of the global store at location ℓ. In other words, global memory is indexed by the set of locations that act as monitors: whenever a thread attempts to synchronize against one of these monitors (say, ℓ), the thread augments its local cache with the store associated with ℓ in the global store. The body of the guard expression is then evaluated with respect to this updated cache. When the expression completes, the converse operation is performed: the contents of the local cache are flushed to the global store indexed by ℓ. Thus, threads that synchronize on different references will not have their updates made visible to one another. Observe that the semantics do not support a single global store; to propagate effects performed in one thread to all other threads would require encoding a protocol that uses a global monitor for synchronization. To simplify the presentation, we prohibit nested guard expressions from synchronizing on the same reference ($\ell \notin \mathit{lockset}(S, \mathtt{t})$).

3.1 Schedules

When the body of the guard is entered, the schedule is augmented to record the fact that there was monitored access to `e` via monitor ℓ by thread `t` ($\mathbf{acq}_{\mathtt{t}}^{\Gamma}\ \ell$). When evaluation of the guard completes, the schedule is updated to reflect that reference ℓ is no longer used as a monitor by `t` ($\mathbf{rel}_{\mathtt{t}}^{\Gamma'}\ \ell$). The global store recorded in the schedule at synchronization acquire and release points will be used to define safety conditions for mutual-exclusion and transactional execution as we describe below.

These semantics make no attempt to enforce a particular concurrency model on thread execution. Instead, we specify safety properties that dictate the legality of an interleaving by defining predicates on schedules. To do so, it is convenient to reason in terms of *regions*, subschedules produced as a result of guard evaluation:

$$\mathit{region}(S) = \{S_i \preceq S | S_i = \mathbf{acq}_{\mathtt{t}}^{\Gamma}\ \ell . S_i' . \mathbf{rel}_{\mathtt{t}}^{\Gamma'}\ \ell\}$$

For any region $R = \mathbf{acq}_{\mathtt{t}}^{\Gamma}\ \ell . S_i' . \mathbf{rel}_{\mathtt{t}}^{\Gamma'}\ \ell$, $\mathcal{T}(R) = \mathtt{t}$, and $\mathcal{L}(R) = \ell$.

The predicate $\mathit{msafe}(S)$ defines the structure of schedules that correspond to an interpretation of guards in terms of mutual exclusion:

Definition 1. *Msafe*

$$\frac{\forall R \in \mathit{region}(S)\ \mathcal{T}(R) = \mathtt{t}, \mathcal{L}(R) = \ell \qquad \mathtt{t} \neq \mathtt{t}' \rightarrow\ \mathbf{acq}_{\mathtt{t}'}^{\Gamma'}\ \ell \notin R}{\mathit{msafe}(S)}$$

For a schedule to be safe with respect to concurrency control based on mutual exclusion, multiple threads cannot concurrently execute within the body of a guarded region

protected by the same monitor. Thus if thread t is executing within a guard protecting expression e using monitor ℓ, no other thread can attempt to acquire ℓ until t relinquishes it.

We can also interpret thread execution within guarded expressions in terms of transactions. Under this interpretation, multiple threads can execute transactionally within the same guarded expression concurrently. To ensure the legality of such concurrent executions, we impose constraints that capture notions of transactional *isolation* and *atomicity* on schedules:

Definition 2. *Isolated*

$$\frac{\begin{array}{c}\forall R \in \mathit{region}(S) = \mathbf{acq}_{\mathtt{t}}^{\Gamma}\ \ell.S'.\mathbf{rel}_{\mathtt{t}}^{\Gamma'}\ \ell \\ \forall \mathbf{rd}_{\mathtt{t}}\ \ell' \in S' \\ \Gamma(\ell) = \sigma \wedge \Gamma'(\ell) = \sigma' \rightarrow \sigma(\ell') = \sigma'(\ell')\end{array}}{\mathit{isolated}(S)}$$

Isolation ensures that locations read by a guarded region are not changed during the region's evaluation. The property is enforced by requiring that the global store associated with a region's monitors is not modified during the execution of the region. Note that the global store Γ' at the point control exits a guarded expression does reflect global updates performed by other threads, but does not reflect *local* updates performed by the current thread that have yet to be propagated. Thus, the isolation rule captures visibility constraints on schedules corresponding to execution within a guard expression; namely, any location read within the schedule cannot be modified by other concurrently executing threads.

Definition 3. *Atomic*

$$\frac{\begin{array}{c}\forall N, R \in \mathit{region}(S), \quad \ell = \mathcal{L}(N) \quad R = S_b.N.S_a \\ \mathcal{T}(N) = \mathcal{T}(R) = \mathtt{t} \wedge\ \mathtt{t} \neq \mathtt{t}' \rightarrow \mathbf{acq}_{\mathtt{t}'}^{\Gamma}\ \ell \notin S_a\end{array}}{\mathit{atomic}(S)}$$

Atomicity ensures that the effects of a guarded region are not propagated to other threads until the region completes. Observe that our semantics propagate updates to the global store when a guarded region exits; these updates become visible to any thread that subsequently executes a guard expression using the same monitor. Thus, a nested region may have its effects made visible to any thread whose execution is mediated by the same monitor. This would violate our intuitive notion of atomicity for the enclosing guarded region since its partial effects (i.e., the effects performed by the inner region) would be visible before it completes. Our atomicity rule thus captures the essence of a closed nested transaction model: the effects of a nested transaction are visible to the parent, via the local store, but are propagated to other threads only when the outermost transaction completes.

Our safety rules are distinguished from other attempts at defining atomicity properties [14, 15] for concurrent programs because they do not rely on mutual-exclusion semantics for lock acquisition and release. For example, consider a schedule in which two threads interleave execution of two guarded regions protected by the same monitor. Such an execution is meaningless for semantics in which synchronization is defined

in terms of mutual-exclusion, but quite sensible if guarded regions are executed transactionally. Isolation is satisfied if the actions performed by the two threads are non-overlapping. Atomicity is satisfied even if these guarded regions execute in a nested context because the actions performed within a region by one thread are not witnessed by the other due to the language's release consistency memory model.

Definition 4. *Tsafe.* We also define $tsafe(S)$ (read "transaction-safe") to hold if both $atomic(S)$ and $isolated(S)$ hold.

3.2 Safety

In order to allow implementations to choose adaptively either a transactional or mutual-exclusion based protocol for guarded regions, dictated by performance considerations, it must be the case that there is no observable difference in the structure of the global store as a consequence of the decisions taken. We show that programs that satisfy *atomicity* and *isolation* exhibit this property.

Suppose program P induces schedule S_P and $tsafe(S_P)$ holds. Now, if $msafe(S_P)$ also holds, then any region in S_P can be implemented either transactionally or using mutual-exclusion. Suppose, however, that $msafe(S_P)$ does not hold. This is clearly possible: consider an interleaving in which distinct threads concurrently evaluate guard expressions protected by the same monitor, but whose bodies access disjoint locations.

Our soundness theorem shows that every such schedule can be permuted to one that satisfies both $msafe$ and $tsafe$. In other words, for every transaction-safe schedule, there is an equivalent schedule that also satisfies constraints defining mutual-exclusion. Thus, as long as regions in a program obey atomicity and isolation, they can be implemented by either one of the mutual-exclusion or closed nested transaction mechanisms without violating program semantics.

Theorem 1. *Soundness.* Let

$$\mathtt{t}[\mathtt{e}], \Delta_0, \Gamma_0, \phi \Longrightarrow^* \mathtt{t}[\mathtt{v}], \Delta, \Gamma, S$$

and suppose $tsafe(S)$ holds but $msafe(S)$ does not. Then, there exists a schedule S_f such that

$$\mathtt{t}[\mathtt{e}], \Delta_0, \Gamma_0, \phi \Longrightarrow^* \mathtt{t}[\mathtt{v}], \Delta', \Gamma', S_f$$

where $tsafe(S_f)$ and $msafe(S_f)$ hold, and in which $\Gamma = \Gamma'$.

Proof Sketch. Let S be $tsafe$, $R \preceq S$, and suppose $msafe(R)$ does not hold, and thus $msafe(S)$ does not hold. Suppose $R = \mathbf{acq}_{\mathtt{t}}^{\Gamma}\ \ell.S'.\mathbf{rel}_{\mathtt{t}}^{\Gamma'}\ \ell$. Since $msafe(R)$ does not hold, there must be some $R' \preceq S$ such that $R' = \mathbf{acq}_{\mathtt{t}'}^{\Gamma''}\ \ell.S''.\mathbf{rel}_{\mathtt{t}'}^{\Gamma'''}\ \ell$ where $\mathbf{acq}_{\mathtt{t}'}^{\Gamma''}\ \ell \in S'$. Since $isolated(S)$ holds, $isolated(R)$ must also hold, and thus none of the actions performed by $\mathtt{t}'$ within S' are visible to actions performed by $\mathtt{t}$ in S'. Similarly, since atomicity holds, actions performed by $\mathtt{t}$ in S' are not visible to operations executed by $\mathtt{t}'$ in S'. Suppose that $\mathbf{rel}_{\mathtt{t}'}^{\Gamma'''}\ \ell$ follows R in S. Then, effects of S' may become visible to operations in S'' (e.g., through nested synchronization actions). But, then $isolated(R')$ would not hold. However, because $tsafe(S)$ holds, we can construct a

permuted schedule S_P of S' in which actions performed by R' are not interleaved with actions performed by R, thus ensuring that $msafe(S_P)$, $isolated(S_P)$, and $atomic(S_P)$ all hold.

4 Design Overview

Our design is motived by three overarching goals:

1. The specific protocol used to implement guarded regions must be completely transparent to the application. Thus, Java `synchronized` blocks and methods serve as guarded regions, and may be executed either transactionally or exclusively depending upon heuristics applied at run-time.
2. The modifications necessary to support such transparency should not lead to performance degradation in the common case – single-threaded uncontended execution within a guarded region – and should lead to notable performance gain in the case when there is contention for entry to the region.
3. Transparency should not come at the expense of correctness. Thus, transactional execution should not lead to behavior inconsistent with Java concurrency semantics.

Issue 3 is satisfied for any Java program that is transaction-safe as defined in the previous section. Fortunately, recent studies have shown that the overwhelming majority of concurrent Java programs exhibit monitor usage that satisfy the correctness goal by using monitors solely as a mechanism to enforce atomicity and isolation for sequences of operations manipulating shared data [14]. We thus focus our attention in the remainder of this section on the first two goals.

Note that lock-based synchronization techniques for languages such as Java are already heavily optimized for the case where monitors are uncontended [2, 5]. Indeed, the Jikes RVM platform that serves in our experiments already supports very efficient lock acquisition and release in this common case: atomically (using "test-and-set" or equivalent instructions) set a bit in the header of the monitor object on entry and clear it on exit. Only if another thread tries to acquire the monitor does *lock inflation* occur to obtain a "fat" lock and initiate full-blown synchronization with wait queues, etc. Thus, the second of our goals has already been met by the current-state-of-the-art.

Supporting transactional execution of guarded regions in place of such highly-optimized locking techniques is thus a significant engineering challenge, if they are to have any advantage at all. As discussed below, our implementation uses *optimistic* concurrency control techniques to minimize the overhead of accesses to shared data [21].

We also make the obvious but important assumption that a guarded region cannot be executed concurrently by different threads using different protocols (i.e., locking or transactional). Any thread wishing to use a different protocol (e.g., locking) than the one currently installed (e.g., transactional) for a given monitor must wait until all other threads have exited the monitor.

4.1 Nesting and Delegation

Since Java monitors support nesting, our transparency requirement means that transactional monitors must also support nesting. There is no conceptual difficulty in dealing

with nesting; recall that the definition of atomicity and isolation captures the essence of the closed nested transaction model [24], and that the prevalent usage of monitors is to enforce atomicity and isolation [14].

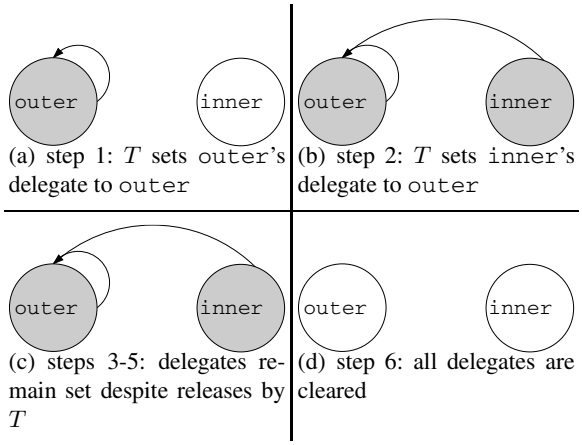
Unfortunately, nesting poses a performance challenge since each monitor defines a locus of contention, for which we must maintain enough information to validate the serializability invariants that guarantee atomicity and isolation. Nesting exacerbates this overhead since nested monitors must record separate access sets used to validate serializability.

However, there is no *a priori* reason why accesses must be mediated by the immediately enclosing monitor that guards them. For example, a single global monitor could conceivably be used to mediate all accesses within all monitors. Under transactional execution, a single global monitor effectively serves to implement the `atomic` construct of Harris and Fraser [17]. Under lock-based execution, a single global monitor defines a global exclusive lock. The primary reason why applications choose *not* to mediate access to shared regions using a single monitor is because of increased contention and corresponding reduced concurrency. In the case of mutual exclusion, a global lock reduces opportunities for concurrent execution; in the case of transactional execution, a global monitor would have to mediate accesses from logically disjoint transactions, and is likely to be inefficient and non-scalable.

Nonetheless, we can leverage this observation to optimize an important specific case for transactional execution of monitors. Consider a thread T acquiring monitor `outer`, and prior to releasing `outer`, also acquiring monitor `inner`. If no other thread attempts concurrent acquisition of `inner` (i.e., the monitor is uncontended) then the acquisition of `inner` can be *delegated* to `outer`. In other words, instead of synchronizing on monitor `inner` we can establish `outer` as `inner`'s delegate and synchronize on `outer` instead. Since monitor `inner` is uncontended, there is nothing for `inner` to mediate, and no loss of efficiency accrues because of nesting (provided that the act of setting a delegate is inexpensive). Of course, when monitor `inner` *is* contended, we must ensure that atomicity and isolation are appropriately enforced. Note that if `inner` was an exclusive monitor, there would be no benefit in using delegation since acquisition of an uncontended mutual-exclusion monitor is already expected to have low overhead.

Protocol Description. Figure 3 illustrates how the delegation protocol works for a specific schedule; for simplicity, we show only Java-level monitor acquire/release operations. The schedule consists of steps 1 through 6 enumerated in the first column of the schedule table. The right-hand side of Figure 3 describes the state of the transactional monitors, used throughout the schedule, with respect to delegation. A monitor whose delegate has been set is shaded grey; an arrow represents the reference to its delegate. To begin, we assume that the delegates of both monitor `outer` and monitor `inner` have not been set. Thread T starts by (transactionally) "acquiring" monitor `outer`, creating a new transaction whose accesses are mediated by `outer` and setting `outer`'s delegate to itself (step 1 in Figure 3(a)). Then T proceeds to (transactionally) "acquire" monitor `inner`. Because there is no delegate for `inner`, and T is already executing within a transaction mediated by `outer`, T sets `inner`'s delegate to refer to `outer` (step 2 in Figure 3(b)).

	T	T'
1	**acq**(outer)	
2	**acq**(inner)	
3		**acq**(inner)
4	**rel**(inner)	
5	**rel**(outer)	
6		**rel**(inner)



(a) step 1: T sets outer's delegate to outer

(b) step 2: T sets inner's delegate to outer

(c) steps 3-5: delegates remain set despite releases by T

(d) step 6: all delegates are cleared

Fig. 3. Delegation example

This protocol implements a closed nested transaction model: the effects of T's execution within monitor inner are not made globally visible until the outer transaction commits, since only outer is responsible for mediating T's accesses and validating serializability.

The delegates stay set throughout steps 3, 4 and 5 (Figure 3(b)), even after thread T, the setter of both delegates, commits its top-level transaction and "releases" outer. In the meantime, thread T' attempts to "acquire" inner. The delegate of inner is at this point already set to outer so thread T' starts its own transaction whose accesses are mediated by outer. The delegates are cleared only after T''s transaction, mediated by outer, commits or aborts. At this point there is no further use for the delegates.

Note that some precision is lost in this example: the transactional meta-data maintained by outer is presumably greater than what would be necessary to simply implement consistency checks for actions guarded by inner. However, despite nesting, only one monitor (outer) has been used to mediate concurrent data accesses and only one set of transactional meta-data was created for outer. However, observe that if the actions of steps 2 and 3 are reversed so that T' acquires inner before T then inner's delegate would not be set, and T' would begin its new transaction mediated in this case by inner, so transactional meta-data for both outer and inner would be needed.

4.2 Reverting to Mutual Exclusion

Optimistic concurrency control assumes the existence of a revocation mechanism so that the effects of a transaction can be reversed on abort. In real world Java applications some operations (e.g., I/O) are irrevocable, so their effects cannot be reversed. To handle such situations, we force any thread executing transactionally guarded by some monitor, but which attempts an irrevocable operation, to revert immediately to mutual exclusion. To support this, each thread executing transactionally must record the monitors it has "acquired" in order of acquisition. Our implementation reverts to mutual exclusion calls to native methods, and at explicit thread synchronization using wait/notify. At the point where such operations arise, we attempt acquisition of all the monitors that the

thread acquired transactionally, in order of acquisition. Successful acquisition of all the monitors implies that all other threads executing transactions against those monitors have completed, exited the monitors, and cleared their respective delegates. From that point on the locking thread can proceed in mutual-exclusion mode, releasing the locks as it exits the monitor scopes. If the transition is unsuccessful (because some other thread acquired the monitors in lock-mode) then the thread executing the irrevocable operation is revoked (i.e., its innermost transaction is aborted) and re-executed from its transaction starting point.

5 Implementation

Our transactional delegation protocol reduces overheads for uncontended nested monitors executed transactionally by deploying nested transaction support only when absolutely necessary. Thus, transactions are employed only for top-level monitors or for contended nested monitors, as described earlier.

Transactions are implemented using an optimistic protocol [21], divided into three phases: *read*, *validate* and *write-back*. Each transaction updates private copies of the objects it manipulates: a copy is created when the transaction (thread) first writes to an object. The validation phase verifies transaction-safety, aborting the transaction and discarding the copies if safety is violated. Otherwise, the write-back phase lazily propagates updated copies to the shared heap, installing each of them atomically.

In the remainder of this section we discuss our strategy for detecting violation of serializability via dependency tracking, our solutions for revocation and re-execution on abort, and details of the implementation platform.

5.1 Platform

Our prototype implementation is based on the Jikes Research Virtual Machine (RVM) [4]. The Jikes RVM is a reserch Java virtual machine with performance comparable to many production virtual machines. Jikes RVM itself is written almost entirely in Java and is self-hosted (i.e., 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 distribution includes both a *baseline* and an *optimizing* compiler. The baseline compiler performs a straightforward expansion of each bytecode instruction into its corresponding sequence of assembly instructions. Our prototype targets the Intel x86 architecture.

5.2 Read and Write Barriers

Our technique to control and modify accesses to shared data uses compiler-inserted read and write *barriers*. These barriers are code snippets emitted by the compiler to augment each heap read and write operation. They trigger creation of versions (copy-on-write) and redirection of reads to the appropriate version, as well as tracking data dependencies.

5.3 Detecting Validity

Threads executing concurrently in the scope of a given monitor will run as separate transactions. Each transaction hashes its shared data accesses into two private hash

maps: a read-map and a write-map, mapping each shared object to a single bit. Once a transaction commits and propagates its updates into the shared heap it also propagates information about its own updates to a global write-map associated with the monitor mediating the transaction. Other transactions whose operations are mediated by the same monitor will then, during their validation phase, intersect their local read-maps with the global write-map to determine if the shared data accesses caused a violation of serializability. When nesting results in an inner monitor running a distinct nested transaction (as opposed to piggy-backing on its delegate-parent) there will be a separate global write-map for each transaction level, so validation must check all global write-maps at all nesting levels. The remaining details of our implementation are the same as in our earlier work [36].

Since for most Java programs reads significantly outnumber writes, reducing the number of read barriers is critical to achieving reasonable performance. Our implementation therefore trades-off accuracy for run-time efficiency in detecting violation of transaction safety. Instead of placing barriers on all reads to shared heap variables (e.g., reading an integer from an object), we assume that the first time a reference is loaded from the heap, it will eventually be used to read from its target object. Thus, read barriers are placed only on loads of *references* from the heap. In other words, we "pre-read" (tagging the local-read map appropriately) all objects whose references are loaded to the stack of a transactional thread. As a result, even objects that are never read, but only written, are conservatively pre-read. This greatly simplifies version management and enables early detection of serializability violations, as described below. This read barrier optimization is applied only for objects and arrays. All other accesses, including all reads from static variables and all writes to shared items incur an appropriate barrier.

5.4 Revocation

Our revocation procedure is identical to our prior work [36], allowing for transaction abort at arbitrary points during its execution. The abort is signaled by throwing a `Revoke` exception. Undo and re-execution procedures are implemented using a combination of bytecode re-writing and virtual machine modifications. Even though Java monitors are lexically scoped, it is necessary to support transaction aborts at arbitrary points to correctly handle native method calls as well as `wait` and `notify` operations, as described in Section 4.2.

In the case of optimistic protocols, the decision about whether a transaction should be committed or aborted is made during the validation phase. Since transactions are lexically scoped, it is relatively easy to encapsulate the computation state at the beginning of the transaction so that it can be reinstated if the transaction aborts, by copying the closure of thread-local state at that point. We use bytecode rewriting in conjunction with a modified exception handling mechanism to restore this saved state on abort.

5.5 Versioning

We use shared data versioning to prevent the effects of incomplete transactions from being made visible to other threads until they commit. We maintain versions of both objects and arrays, as well as static (global) variables. Object and array versioning are

handled exactly the same. Statics use a slightly modified approach, requiring boxing and unboxing of the static values.

Because our array versioning procedure is identical to that used for versioning objects, we refer only to objects in the following description. Versions are accessible through a forwarding pointer from the original object. We use a "copy-on-write" strategy for creating new versions. A transaction creates a new (uncommitted) copy right before performing first update to an object, and redirects all subsequent read and write operations to access that version. It is important to note that for transaction safety all programs executed in our system must be race-free (a prerequisite for atomicity): all accesses by all threads to a given shared data item must be guarded by the same monitor [14]. As a result, writes to the same location performed by different threads will be detected as unsafe by our validity check described above. This also means that only the first transaction writing to a given object need create a version for it. Other transactions accessing that object are aborted when the writing transaction commits and discovers the overlap.

Upon successful commit of a transaction, the current version becomes the *committed version* and remains accessible via a *forwarding pointer* installed in the original object. Subsequent accesses are re-directed (in the read and write barriers) via the forwarding pointer to the committed version. When a transaction aborts all its versions are discarded. Note that at most two versions of an object exist at any given time: a committed version and an uncommitted version.

As noted above, the read barriers are only executed on reference loads. In general, multiple on-stack references may end up pointing to *different* versions of *the same* object. This is possible, even though read barriers are responsible for retrieving the most up-to-date version of the object, writes may occur after the reference has been loaded to multiple locations on the stack. The run-time system must thus ensure that the version of an object accessible through an on-stack reference is the "correct" one. The visibility rules for the Java Memory Model [22] mean that at certain synchronization points (e.g., monitor entry, access to volatile variables, etc.) threads are obliged to have the same view of the shared heap. As a result, it is legal to defer fixing on-stack references until specific synchronization points (e.g., monitor enter/exit, wait/notify). At these points all on-stack references must be forwarded to the most up-to-date version. Reference forwarding is implemented using a modified version of thread stack inspection as used by the garbage collector.

In addition to performing reference forwarding at synchronization points, when a version is first created by a transaction, the thread creating the version must forward all references on its stack to point to the new version. This ensures that all subsequent accesses (by the same thread) observe the results of the update.

5.6 Example

We now present an example of how these different implementation features interact. Figure 4 describes actions concerning shared data versioning and serializability violation detection, performed by threads T, T' and T'' executing the schedule shown in Table 1. Figure 4(a) represents the initial state, before any threads have started executing. Wavy lines represent threads, and circles represent objects `o1` and `o2`. The objects

Table 1. A non-serializable schedule

Step	T	T'	T''
1	**acq**(outer)		
2	**wt**(o2)		
3			**acq**(inner)
4			**wt**(o1)
5		**acq**(outer)	
6		**acq**(inner)	
7		**rd**(o1)	
8			**rel**(inner)
9	**rel**(outer)		
10		**rd**(o1)	
11		**rel**(inner)	
12		**rel**(outer)	

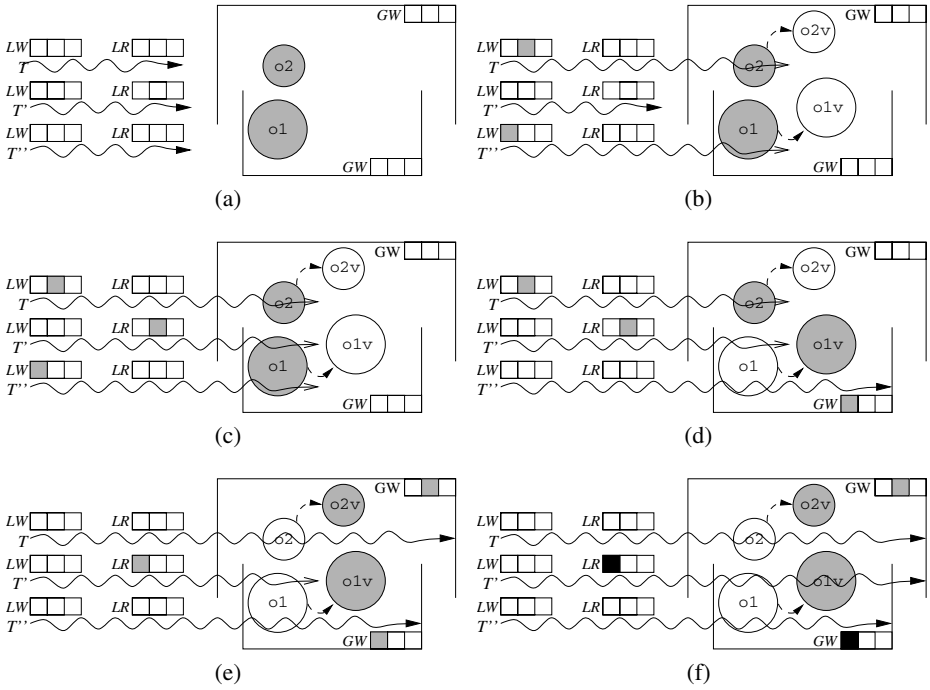


Fig. 4. A non-serializable execution

have not yet been versioned – they are shaded grey because at the moment they contain the most up-to-date values. The larger box (open at the bottom) represents the scope of transactional monitor `outer`, the smaller box (open at the top) represents the scope of transactional monitor `inner`. Both the global write map (GW) associated with the monitor and the local maps (write map LW and read map LR) associated with each thread have three slots. Maps that belong to a given thread are located above the wavy

line representing this thread. We assume that accesses to object `o1` hash to the first slot of every map and accesses to object `o2` hash to the second slot of every map

Execution begins with threads T and T'' starting to run transactions whose operations are mediated by monitors `outer` and `inner` (respectively) and performing updates to objects `o2` and `o1` (respectively), as presented in Figure 4(b). The updates trigger creation of copies `o2v` and `o1v` for objects `o2` and `o1`, respectively, and tagging of the local write maps. Thread T tags the second slot of its local write map since it modifies object `o2`, whereas thread T'' tags the first slot of its local write map since it modifies object `o1`. In Figure 4(c) thread T' starts executing, running the outermost transaction mediated by monitor `outer` and its inner transaction mediated by monitor `inner`, and then reads object `o1`, which tags the local read map. In Figure 4(d) T'' attempts to commit its transaction. Since no writes by other transactions mediated by monitor `inner` have been performed, the commit is successful: `o1v` becomes the committed version, the contents of T'''s local write map are transferred to `inner`'s global write map and the local write map is cleared. Similarly, in Figure 4(e), T's transaction commits successfully: `o2v` becomes the committed version and the local write map is cleared after its contents has been transfered to the global write map associated with monitor `outer`. In Figure 4(f) thread T' proceeds to again read object `o1` and then to commit its transactions (both inner and outer). However, because a new committed version of object `o1` has been created, `o1v` is read by T' instead of the original object. When attempting to commit both its inner and outer transactions, thread T' must intersect its local read map with the global maps associated with both monitor `outer` and monitor `inner`. The first intersection is empty (no writes performed in the scope of monitor `outer` could compromise reads performed by T'), the second however is not – both transactions executed by T' must be aborted and re-executed.

5.7 Header Compression

For performance we need efficient access to several items of meta-data associated with each object (e.g., versions and their identities, delegates, identity hash-codes, access maps, etc.). At the same time, we must keep overheads to a minimum when transactions are not used. The simplest solution is to extend object headers to associate the necessary meta-data. Our transactional meta-data requires up to four 32-bit words. Unfortunately, Jikes RVM does not easily support variable header sizes and extending the header of each object by four words has serious overheads of space and performance, even in the case of non-transactional execution. On the other hand keeping meta-data "on the side" (e.g., in a hash table), also results in a significant performance hit.

We therefore implement a compromise. The header of every object is extended by a single *descriptor word* that is lazily populated when meta-data needs to be associated with the object. If an object is never accessed in a transactional context, its descriptor word remains empty. Because writes are much less common than reads, we treat the information needed for reads as the common case. The first transactional read will place the object's identity hash-code in the descriptor (we generate hash codes independently of native Jikes RVM object hashcodes to ensure good data distribution in the access maps). If additional meta-data needs to be associated with the object (e.g., a new version on write) then the descriptor word is overwritten with a reference to a new *descriptor*

object containing all the necessary meta-data (including the hash-code originally stored in the descriptor word). We discriminate these two cases using the low-order bit of the descriptor word.

5.8 Code Duplication

Transactional support (e.g., read and write barriers) is required only when a thread decides to execute a given monitor transactionally. However, it is difficult to determine if a particular method is going to be used only in a non-transactional context. To avoid unnecessary overheads during non-transactional execution, we use bytecode rewriting to duplicate the code of all (user-level) methods actually being executed by the program. Every method can then be compiled in two versions: one that embeds transactional support (transactional methods) and one that does not (non-transactional methods). This allows the run-time system to dynamically build a call chain consisting entirely of non-transactional methods for non-transactional execution. Unfortunately, because of our choice to access most up-to-date versions of objects through forwarding pointers, we cannot fully eliminate read barriers even in non-transactional methods. We can however eliminate all write barriers and make the non-transactional read barriers very fast in the common case – they must simply differentiate objects that have never been accessed transactionally from those that have. In addition to the usual reference load, such barriers consist only of a null check, one condition, and one load. These instructions verify that the descriptor word is empty, indicating that the object has never been accessed transactionally, so no alternate version has ever been created.

5.9 Triggering Transactional Execution

Our implementation must be able to determine whether to execute a given monitor transactionally or exclusively. We use a very light-weight heuristic to detect monitor contention and trigger transactional execution only for contended monitors . The first thread to enter a monitor always executes the monitor exclusively. It is only after a thin mutual-exclusion lock is "inflated" by being turned into a fat lock (on contended acquisition of the lock) that the monitor in question is asserted to be contended. All threads queued waiting for the monitor will then execute transactionally once the currently executing (locking) thread exits the monitor. We recognize that there are more advanced and potentially more conservative heuristics that a production system may wish to use. For example, programmer annotations could be provided to mark the concurrency control mechanism that is to be used for different monitors. Adaptive solutions based on dynamic profiling or solutions utilizing off-line profiles may also provide more refined information on when it is best to execute monitors transactionally.

6 Experiments

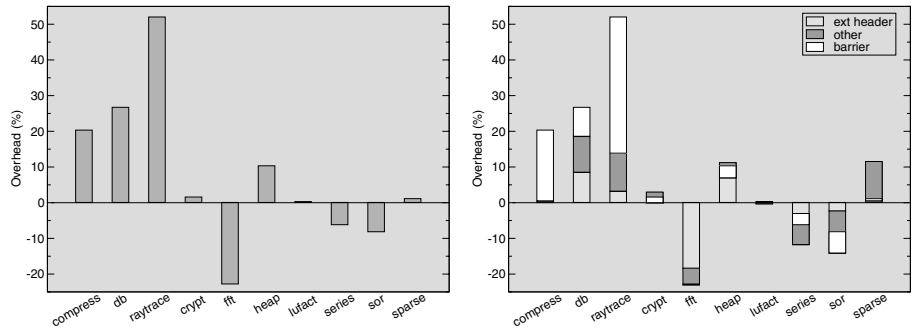
The performance evaluation of our prototype implementation is divided into two parts. We use a number of single-threaded benchmarks (from the SPECjvm98 [31] and Java Grande [30] suites) to measure the overheads of supporting hybrid-mode execution (e.g., compiler-inserted barriers, code-duplication, object layout modifications, etc.)

when monitors are uncontended. We also use an extended version of the OO7 object database benchmark [10], to expose the range of performance when executing under different levels of monitor contention. We measure the behavior when all monitors are executed transactionally and when using the hybrid scheme that executes monitors transactionally only when sufficient monitor contention is detected. Our measurements were taken on an eight-way 700MHz Intel Pentium III symmetric multi-processor (SMP) with 2GB of RAM running Linux kernel version 2.4.20-31.9smp (RedHat 9.0). Our implementation uses version 2.3.4+CVS (with 2005/12/08 15:01:10 UTC timestamp) of Jikes RVM for all the configurations used to take the measurements (mutual-exclusion-only, transactions-only and hybrid). We ran each benchmark configuration in its own invocation of the virtual machine, 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.

6.1 Uncontended Execution

A summary of our performance evaluation results when monitors are uncontended is presented in Figure 5. Our current prototype implementation is restricted to running bytecodes compiled with debugging information for local variables; this information is needed by the bytecode rewriter for generating code to store and restore local state in case of abort. Therefore, we can only obtain results for those SPECjvm98 benchmarks for which source code is available.

In Figure 5(a) we report total summary overheads for a configuration that supports hybrid-mode execution. The overheads are reported as a percentage with respect to a "clean" build of the "vanilla" unmodified Jikes RVM. The average overhead is on the order of 7%, with a large portion of the performance degradation attributed to execution of the compiler-inserted barriers, as described below. Figure 5(b) reveals how different mechanisms for transactional execution affect performance in the uncontended case. The bottom of every bar represents the effect of extending the header of every object by one word (as needed to support transaction-related meta-data). The middle of every bar represents the cost of all other system modifications, excluding compiler-inserted



(a) total overhead (b) component overheads

Fig. 5. Uncontended execution

barriers.[1] The top bar captures overhead from execution of the barriers themselves (mostly read barriers but also barriers on static variable accesses).

Observe that changing the object layout can by itself have a significant impact on performance. In most cases, the version of the system with larger object headers indeed induces overheads over the clean build of Jikes RVM, but in some situations (e.g., `FFT` or `Series`), its performance actually improves over the clean build by a significant amount; variations in cache footprint is the most likely cause. The performance impact of the compiler-inserted barriers is also clearly noticeable, especially in the case of benchmarks from the SPECjvm98 suite. When discounting overheads related to the execution of the barriers, the average overhead with respect to the clean build of Jikes RVM drops to a little over 1% on average. This result is consistent with that reported by Blackburn and Hosking [7] for garbage collection read barriers that can incur overheads up to 20%. It would be beneficial for our system to use a garbage collector that might help to amortize the cost of the read barrier. Fortunately, there exist modern garbage collectors (e.g., [6]) that fulfill this requirement.

6.2 Contended Execution

The OO7 benchmark suite [10] provides a great deal of flexibility of benchmark parameters (e.g., database structure, fractions of reads/writes to shared/private data). The multi-user OO7 benchmark [9] allows control over the degree of contention for access to shared data. In choosing OO7 as a benchmark our goal was to accurately gauge the various trade-offs inherent with our implementation over a wide range of different workloads, rather than emulating specific workloads of potential applications. We believe the benchmark captures essential features of scalable concurrent programs that can be used to quantify the impact of the design decisions underlying our implementation.

The OO7 benchmark operates on a synthetic design database, consisting of a set of *composite parts*. Each composite part comprises a graph of *atomic parts*, and a `document` object containing a small amount of text. Each atomic part has a set of attributes (i.e., fields), and is connected via a bi-directional association to several other atomic parts. The connections are implemented by interposing a separate connection object between each pair of connected atomic parts. Composite parts are arranged in an *assembly* hierarchy; each assembly is either made up of composite parts (a *base* assembly) or other assemblies (a *complex* assembly). Each assembly hierarchy is called a *module*, and has an associated *manual* object consisting of a large amount of text. Our results are all obtained with an OO7 database configured as in Table 2.

Our implementation of OO7 conforms to the standard OO7 database specification. Our traversals are a modified version of the multi-user OO7 traversals. A traversal chooses a single path through the assembly hierarchy and at the composite part level randomly chooses a fixed number of composite parts to visit (the number of composite parts to be visited during a single traversal is a configurable parameter). When the traversal reaches the composite part, it has two choices:

1. Do a *read-only* depth-first traversal of the atomic part subgraph associated with that composite part; or

[1] The measurements were taken after artificially removing compiler-inserted barriers from the "full" version of the system. Naturally our system cannot function without barriers.

Table 2. Component organization of the OO7 benchmark

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

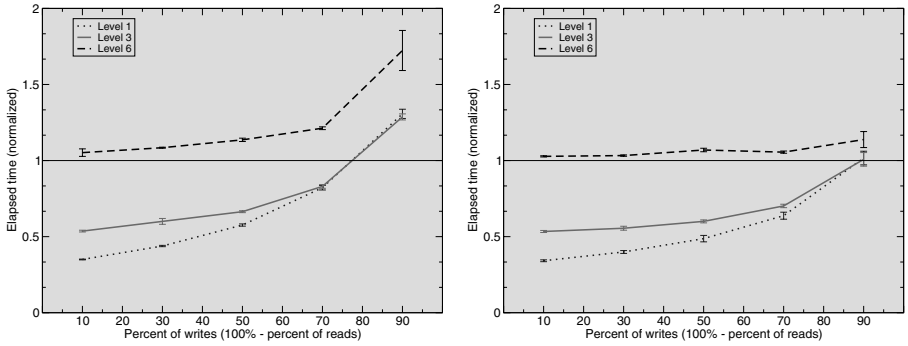
2. Do a *read-write* depth-first traversal of the associated atomic part subgraph, swapping the x and y coordinates of each atomic part as it is visited.

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

Our experiments here use traversals that always operate on the *shared* module, since we are interested in the effects of contention on performance of our system. Our implementation of OO7 conforms to the standard OO7 database specification. Our traversals differ from the original OO7 traversals in adding a parameter that controls entry to monitors at varying levels of the database hierarchy. We run 64 threads on 8 physical CPUs. Every thread performs 1000 traversals (enters 1000 monitors) and visits 4M atomic parts during each iteration. When running the benchmarks we varied the following parameters:

- ratio of shared reads to shared writes: from 10% shared reads and 90% shared writes (mostly read-only workload) to 90% shared reads and 10% shared writes (mostly write-only workload).
- level of the assembly hierarchy at which monitors are entered: level one (module level), level three (second layer of complex assemblies) and level six (fifth layer of complex assemblies). Varying the level at which monitors are entered models different granularities of user-level synchronization from coarse-grained through to fine-grained and diversifies the degree of monitor contention.

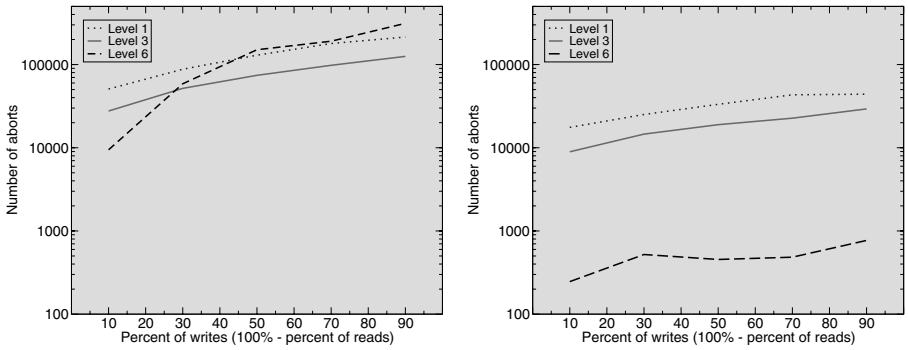
Figure 6 plots execution times for the OO7 benchmark when all threads execute all monitors transactionally (Figure 6(a)) and when threads execute in hybrid mode, where the mode is chosen based on monitor contention (Figure 6(b)). The execution times are normalized with respect to the performance of the "clean" build of Jikes RVM (90% confidence intervals are also reported). Figure 7 plots the total number of aborts for both transactions-only (Figure 7(a)) and hybrid (Figure 7(b)) executions. Different lines on the graphs represent different levels of user-level synchronization granularity – one being the most coarse-grained and six being the most fine-grained.



(a) transactions-only (b) hybrid-mode

Fig. 6. Normalized execution times for the OO7 benchmark

When there is a suitable level of monitor contention, and when the number of writes is moderate, transactional execution significantly outperforms mutual exclusion by up to three times. The performance of the transactions-only scheme degrades as the number of writes increases (and so does the number of generated hash-codes) since the number of bitmap collisions increases, leading to a large number of aborts even at low contention (Figure 7(b)). Extending the size of the maps used to detect serializability violations would certainly remedy the problem, at least in part. However, we cannot use maps of an arbitrary size. This could unfavorably affect memory overheads (especially compared to mutual-exclusion locks) but more importantly we have determined that the time to process potentially multiple maps at the end of the outermost transaction must be bounded. Otherwise, the time spent to process them becomes a source of significant delay (currently each map contains over 16,000 slots). The increased number of aborts certainly has a very significant impact on the difference in performance between the transactions-only and hybrid schemes. The overheads of the transactions-only scheme cannot however be attributed only to the increased abort rate – observe that the shape of



(a) transactions-only (b) hybrid-mode

Fig. 7. Total number of aborts for the OO7 benchmark

the graphs plotting execution times and aborts are different. During hybrid-mode execution, monitors are executed transactionally only when monitor contention is detected, read and write operations executed within uncontended monitors incur little overhead, and the revocations are very few. Thus, instead of performance degradation of over 70% in the transactions-only case when writes are dominant, our hybrid scheme incurs overhead on the order of only 10%.

7 Related Work

The design and implementation of our system has been inspired by the optimistic concurrency protocols first introduced in the 1980's [21] 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, 19, 32]. Devising fast and efficient techniques to confirm that a schedule is correct remains an important topic of study.

There have been several attempts to reduce overheads related to mutual-exclusion locking in Java. Agesen et al. [2] and Bacon et al. [5] describe locking implementations for Java that attempt to optimize lock acquisition overhead when there is no contention on a shared object. Other recent efforts explore alternatives to lock-based concurrent programming [17, 36, 20, 18]. In these systems, threads are allowed to execute within a guarded region (e.g., protected by monitors) concurrently, but are monitored to ensure that safety invariants (e.g., serializability) are not violated. If a violation of these invariants by some thread is detected, the computation performed by this thread is revoked, any updates performed so far discarded and the thread is re-executed. Our approach differs from these in that it seamlessly integrates different techniques to manage concurrency within the same system. When using our approach, the most appropriate scheme is dynamically chosen to handle concurrency control in different parts of *the same* application.

There is also a large body of work on removing synchronization primitives when it can be shown that there is never contention for the region being guarded [3, 28, 33]. The results derived from these efforts would equally benefit applications running in the system supporting hybrid-mode execution.

There has been much recent interest in devising techniques to detect data races in concurrent programs. Some of these efforts [13, 8] present new type systems using, for example, ownership types [12] to verify the absence of data races and deadlocks. Others such as Eraser [29] employ dynamic techniques to check for races in programs [25, 23, 34]. There have also been attempts to leverage static analyses to reduce overheads and increase precision of purely dynamic implementations [11, 35].

Recent work on deriving higher-level safety properties of concurrent programs [15, 14] subsumes data-race detection. It is based on the observation that race-free programs may still exhibit undesirable behavior because they violate intuitive invariants such as atomicity that are not easily expressed using low-level abstractions such as locks.

8 Conclusions

Existing approaches to providing concurrency abstractions for programming languages offer disjoint solutions for mediating concurrent accesses to shared data throughout

the lifetime of the entire application. Typically these mechanisms are either based on mutual exclusion or on some form of transactional support. Unfortunately, none of these techniques is ideally suited for all possible workloads. Mutual exclusion performs best when there is no contention on guarded region execution, while transactions have the potential to extract additional concurrency when contention exists.

We have designed and implemented a system that seamlessly integrates mutual exclusion and optimistic transactions to implement Java monitors. We formally argue correctness (with respect to language semantics) of such a system and provide a detailed performance evaluation of our hybrid scheme for different workloads and varying levels of contention. Our implementation and experiments demonstrate that the hybrid approach has low overheads (on the order of 7%) in the uncontended (base) case and that significant performance improvements (speedups up to $3\times$)can be expected from running contended monitors transactionally.

Acknowledgements

We thank the anonymous referees for their suggestions and improvements to this paper. This work is supported by the National Science Foundation under grants Nos. CCR-0085792, CNS-0509377, CCF-0540866, and CNS-0551658, and by gifts from IBM and Microsoft. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

References

[1] Adya, A., Gruber, R., Liskov, B., and Maheshwari, U. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record 24*, 2 (June 1995), 23–34.

[2] Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y. S., and White, D. An efficient meta-lock for implementing ubiquitous synchronization. In OOPSLA'99 [26], pp. 207–222.

[3] Aldrich, J., Sirer, E. G., Chambers, C., and Eggers, S. J. Comprehensive synchronization elimination for Java. *Science of Computer Programming 47*, 2-3 (2003), 91–120.

[4] Alpern, B., Attanasio, C. R., Barton, J. J., Cocchi, A., Hummel, S. F., Lieber, D., Ngo, T., Mergen, M., Shepherd, J. C., and Smith, S. Implementing Jalapeño in Java. In OOPSLA'99 [26], pp. 314–324.

[5] Bacon, D., Konuru, R., Murthy, C., and Serrano, M. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Montréal, Canada, June). *ACM SIGPLAN Notices 33*, 5 (May 1998), pp. 258–268.

[6] Bacon, D. F., Cheng, P., and Rajan, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (New Orleans, Lousiana, Jan.). *ACM SIGPLAN Notices 38*, 1 (Jan. 2003), pp. 285–298.

[7] Blackburn, S. M., and Hosking, A. L. Barriers: Friend or foe? In *Proceedings of the ACM International Symposium on Memory Management* (Vancouver, Canada, Oct., 2004), D. F. Bacon and A. Diwan, Eds. ACM, 2004, pp. 143–151.

[8] 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.

[9] Carey, M. J., DeWitt, D. J., Kant, C., and Naughton, J. F. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, Oct.). *ACM SIGPLAN Notices 29*, 10 (Oct. 1994), pp. 414–426.

[10] Carey, M. J., DeWitt, D. J., and Naughton, J. F. The OO7 benchmark. In *Proceedings of the ACM International Conference on Management of Data* (Washington, DC, May). *ACM SIGMOD Record 22*, 2 (June 1993), pp. 12–21.

[11] Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Berlin, Germany, June). *ACM SIGPLAN Notices 37*, 5 (May 2002), pp. 258–269.

[12] 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.

[13] Flanagan, C., and Freund, S. N. Type-based race detection for Java. In PLDI'00 [27], pp. 219–232.

[14] Flanagan, C., and Freund, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Venice, Italy, Jan.). 2004, pp. 256–267.

[15] 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.

[16] Flatt, M., Krishnamurthi, S., and Felleisen, M. Classes and mixins. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (San Diego, California, Jan.). 1998, pp. 171–183.

[17] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Anaheim, California, Nov.). *ACM SIGPLAN Notices 38*, 11 (Nov. 2003), pp. 388–402.

[18] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, Illinois, June). 2005, pp. 48–60.

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

[20] 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.

[21] Kung, H. T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 9*, 4 (June 1981), 213–226.

[22] Manson, J., Pugh, W., and Adve, S. The Java memory model. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Long Beach, California, Jan.). 2005, pp. 378–391.

[23] Mellor-Crummey, J. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, Nov.). 1991, pp. 24–33.

[24] Moss, J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.

[25] O'Callahan, R., and Choi, J.-D. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, June). 2003, pp. 167–178.

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

[27] *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Vancouver, Canada, June). *ACM SIGPLAN Notices 35*, 6 (June 2000).

[28] Ruf, E. Effective synchronization removal for Java. In PLDI'00 [27], pp. 208–218.

[29] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (Nov. 1997), 391–411.

[30] Smith, L. A., Bull, J. M., and Obdrzálek, J. A parallel Java Grande benchmark suite. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Denver, Colorado, Nov.). 2001, p. 8.

[31] SPEC. SPECjvm98 benchmarks, 1998. http://www.spec.org/osg/jvm98.

[32] Stonebraker, M., and Hellerstein, J., Eds. *Readings in Database Systems*, third ed. Morgan Kaufmann, 1998.

[33] Ungureanu, C., and Jagannathan, S. Concurrency analysis for Java. In *Proceedings of the International Static Analysis Symposium* (Santa Barbara, California, Jun./Jul.), J. Palsberg, Ed. vol. 1824 of *Lecture Notes in Computer Science*. 2000, pp. 413–432.

[34] von Praun, C., and Gross, T. R. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Tampa, Florida, Oct.). *ACM SIGPLAN Notices 36*, 11 (Nov. 2001), pp. 70–82.

[35] von Praun, C., and Gross, T. R. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (San Diego, California, June). 2003, pp. 115–128.

[36] Welc, A., Jagannathan, S., and Hosking, A. L. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming* (Oslo, Norway, June), M. Odersky, Ed. vol. 3086 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004, pp. 519–542.