Feedback-Directed Barrier Optimization in a Strongly Isolated STM

Nathan G. Bronson Christos Kozyrakis Kunle Olukotun
Computer Systems Laboratory
Stanford University
{nbronson, kozyraki, kunle}@stanford.edu

Abstract

Speed improvements in today's processors have largely been delivered in the form of multiple cores, increasing the importance of abstractions that ease parallel programming. Software transactional memory (STM) addresses many of the complications of concurrency by providing a simple and composable model for safe access to shared data structures. Software transactions extend a language with an `atomic` primitive that declares that the effects of a block of code should not be interleaved with actions executing concurrently on other threads. Adding barriers to shared memory accesses provides atomicity, consistency and isolation.

Strongly isolated STMs preserve the safety properties of transactions for all memory operations in a program, not just those inside an `atomic` block. *Isolation barriers* are added to non-transactional loads and stores in such a system to prevent those accesses from observing or corrupting a partially completed transaction. Strong isolation is especially important when integrating transactions into an existing language and memory model. Isolation barriers have a prohibitive performance overhead, however, so most STM proposals have chosen not to provide strong isolation.

In this paper we reduce the costs of strong isolation by customizing isolation barriers for their observed usage. The customized barriers provide accelerated execution by blocking threads whose accesses do not follow the expected pattern. We use hot swap to tighten or loosen the hypothesized pattern, while preserving strong isolation. We introduce a family of *optimization hypotheses* that balance verification cost against generality.

We demonstrate the feasibility of dynamic barrier optimization by implementing it in a bytecode-rewriting Java STM. Feedback-directed customization reduces the overhead of strong isolation from 505% to 38% across 11 non-transactional benchmarks; persistent feedback data further reduces the overhead to 16%. Dynamic optimization accelerates a multi-threaded transactional benchmark by 31% for weakly-isolated execution and 34% for strongly-isolated execution.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code generation, Compilers, Optimization, Run-time environments; D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features – Concurrent programming structures

General Terms Algorithms, Design, Experimentation, Languages, Measurement, Performance

Keywords Transactional Memory, Strong Isolation, Weak Isolation, Hot Swap, Bytecode Rewriting, Deoptimization

```
public class BigDecimal {
  private volatile transient String stringCache;
  public String toString() {
    if (stringCache == null)
      stringCache = layoutChars(true);
    return stringCache;
  }
}
```

Figure 1. A fragment of `java.math.BigDecimal`. The interface has immutable semantics, so it is difficult for a programmer to detect accesses that are incorrect in a weakly isolated STM.

1. Introduction

Speed improvements in today's processors have largely been delivered in the form of multiple cores, increasing the importance of abstractions that ease parallel programming. Software transactional memory (STM) addresses many of the complications of concurrency by providing a simple and composable model for safe access to shared data structures. Software transactions extend a language with an `atomic` primitive that declares that the effects of a block of code should not be interleaved with actions executing concurrently on other threads. Implementation techniques for atomic regions have been studied extensively [1, 2, 6, 9, 10, 18, 23]. Unfortunately, STMs for imperative languages have struggled to provide good performance while retaining intuitive semantics.

1.1 Strong vs. Weak Isolation

An STM proposal that integrates atomic regions into an existing imperative language must define the outcome when a non-transactional access is made to a memory location that is also read or written by a transaction. A system in which non-transactional code cannot observe or corrupt a partially completed transaction is said to provide *strong isolation*. The alternative is *weak isolation*, in which consistency is guaranteed only when heterogeneous accesses do not occur.

Strong isolation is easy for the programmer to reason about and straightforward to integrate into a language's memory model. Grossman et al. show how to add successful transactions to the set of actions included in Java's happens-before ordering [8]. Strong isolation allows the programmer to ignore the effect of partially completed or failed transactions when reasoning about the correctness of their program,[1] and it allows the programmer to reason about whether or not a memory location is currently shared.

The weakest form of isolation provides correct execution for a program only if each memory location is either never accessed in an atomic region *or* always accessed in an atomic region. Programs that violate this convention can observe arbitrarily bad behavior, including producing values from thin air. Libraries often hide stores to shared data inside operations that are semantically read-only, making it difficult or impossible to reason about the correctness of the whole program under weak isolation.

As a specific example, consider the snippet of Java code in Figure 1. `BigDecimal` presents an immutable interface, so it is not unreasonable for a user to expect to be able to call `toString()` for a shared instance both inside and outside an atomic region. However, because weak isolation breaks the carefully defined semantics of `volatile`, the result might be a string full of `'\0'`. In an STM that performs updates in-place, the non-transactional invocation might race with rollback of a transaction that populated the cache and subsequently rolled back. A write-buffering STM can cause a problem during a normal commit because the buffered writes may be issued in an order that violates the Java memory model.

Transaction isolation is broken when non-transactional code on a concurrent thread observes or affects the loads and stores used internally by the STM implementation. Those loads and stores are not actually performed atomically, so a direct access to memory is not a party to the STM's illusion of serializability and consistency. Concurrent execution with strong isolation requires that non-transactional threads be prevented from observing the internal state of a partially completed transaction by adding *isolation barriers* to non-transactional loads and stores [11].

Despite its subtle semantics, weak isolation is attractive because it does not involve modifications to non-transactional code. This localizes any performance impact from `atomic` to code that actually uses transactions. As a result there is now active research in models that constrain the STM's execution schedule to enable some of the useful programming idioms precluded by weak isolation while avoiding the use of isolation barriers [14,20,23]. These models still require whole-program reasoning by the programmer to establish the correctness of individual critical regions.

Static whole-program analysis has been used to identify isolation barriers that may be safely removed [21]. This approach seems promising for environments that can include such an analysis in their development workflow, but is difficult to reconcile with the dynamic features of managed languages such as Java and C#.

1.2 Our Contributions

Research into weakly isolated software transactions is predicated on the assumption that strong isolation adds a prohibitive performance overhead. In this paper we tackle those overheads by dynamically optimizing isolation barriers while preserving strong isolation:[2]

- We construct a family of customized memory barriers that dynamically check that accesses to a class's field or array elements follow an *optimization hypothesis*. Conforming accesses are accelerated (often optimally), while threads that attempt a nonconforming access are blocked. We include stateless access patterns and patterns that are sensitive to the object instance's history (Section 3).

[1] We assume that the STM guarantees forward progress.

[2] We compare our work to the concurrent research of Schneider et al. [19] in Section 9, Dynamic NAIT.

```
// Initially x==0

// Thread 1
atomic {
  ++x;
  ++x;
}

// Thread 2
r1 = x;
assert (r1 % 2 == 0);
```

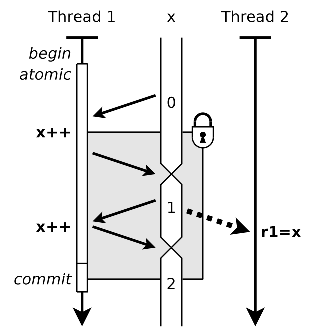


Figure 2. An intermediate dirty read in a weakly isolated STM. Thread 2 observes an inconsistent value for x because it ignores Thread 1's lock. The middle column represents the current value of x in memory, and the dotted line represents the dirty read.

- We describe how to hot swap barrier implementations in a running system, with minimal impact on unaffected threads. We use this hot swap mechanism reactively to rescue threads blocked due to an incorrect optimization hypothesis (Section 5).
- We use runtime profiling to identify fields whose access pattern is compatible with a more aggressive optimization hypothesis, and proactively hot swap the barriers for those fields. Because incorrect optimizations will be corrected by a reactive relaxation, this dynamic analysis does not need to be conservative or sound (Section 6).
- We demonstrate the practicality of dynamic barrier customization by implementing it in an STM for Java. Our evaluation shows that the dynamically optimizing system converges on the performance achieved by an oracle static analysis, the cost of hot swap is quickly recovered, and barrier optimization reduces the overheads of both isolation and transaction barriers (Section 8).

2. Barriers for Strong Isolation

Strong isolation can be provided by adding barriers to non-transactional code. Direct memory loads and stores from another thread can break a transaction's isolation because the STM implementation cannot actually perform all of its loads and stores atomically. Isolation barriers must be used to protect an atomic region from concurrent non-transactional accesses [11]. The details of how those barriers interact with the transactional metadata are specific to the STM implementation, but in all systems they must guarantee that reads do not observe uncommitted data and that writes are either communicated to active transactions or blocked until after all conflicting transactions commit.

To simplify our discussion we will assume, unless otherwise noted, an update-in-place STM implementation similar to that of McRT-STM [18]. Our work is also applicable to write-buffering STMs that use strict two-phase locking to protect updates, including TL2 [6]. In our assumed STM:

- Updates are performed in-place and protected by strict two-phase locking;
- An undo log is used to restore values on rollback;
- Reads are lazily validated using version numbers;
- Versions and locks are maintained at an object granularity; and
- Each object contains its own metadata.

```
// Initially x==0

// Thread 1
atomic {
  r1 = x;
  r2 = x;
  assert (r1 == r2);
}

// Thread 2
x = 1;
```

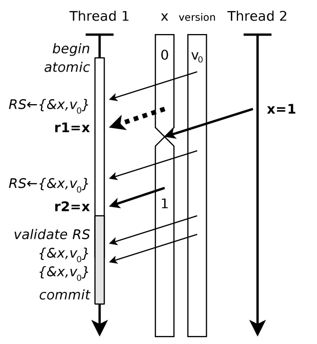


Figure 3. A non-repeatable read in a weakly isolated STM. Thread 2 does not increment the version metadata, so Thread 1's read set (RS) validation incorrectly allows commit. The center columns give the current value of x and x's metadata, and the dotted line represents the non-repeatable read.

2.1 Isolation Requirements

Consider the code and execution described in Figure 2, which illustrates the *intermediate dirty read* problem (examples are adapted from [21]). Thread 1 is using an atomic region to enforce the invariant that x is even, but Thread 2 can observe an inconsistent intermediate state because it has no read isolation barrier. Weak isolation can also cause atomic regions to behave incorrectly. Figure 3 shows a *non-repeatable read*. Thread 1 might commit despite perceiving a spontaneous change in x, because Thread 2 does not increment the version metadata. Many more weak isolation anomalies have been described, but all share a cause with one of the problems we have just described: non-transactional reads bypass the STM's locks and non-transactional writes bypass the STM's transaction validation.

2.2 Ordering Requirements

One of the lock-based idioms that must be supported by `atomic` regions is *privatization*, in which a critical section is used to obtain exclusive access to an object that was previously shared. After privatization the object can be modified with no danger of concurrent access. Weakly isolated transactions can confound the programmer's reasoning in this case, however, because after the privatizing transaction's commit there might still be 'zombie' transactions accessing the data. Although these zombie transactions will eventually roll back (after discovering that their read set is inconsistent) they may perform spurious writes to the privatized object.

The privatization problem was first recognized for the special case of memory reclamation, in which a zombie transaction might access memory that has been reused or returned to the operating system [6]. More recently researchers have studied it in a more general form, and restrictions on transaction commit order have been proposed that provide correctness without isolation barriers for this particular idiom [1,20,23].

The complementary *publication* problem has also been studied. Publication without non-transactional barriers can be made safe by additional restrictions on commit order if an atomic region is used to read the reference to the published object [14]. Many existing usages of the publication idiom, however, use volatile fields.

Java's memory model guarantees release semantics for a volatile store [12]. This can be used to implement a lockless cache, as in Figure 1. It can also be used to (correctly) implement the double-checked locking idiom. When an object's construction and publication are wrapped in a weakly isolated transaction, however, the memory model's guarantees are no longer provided. In a write-buffering STM the actual stores might be issued out of order during commit. In an update-in-place STM the stores are issued in the original order, but if the transaction rolls back then additional stores

Algorithm 1 Generic read and write isolation barriers.

```
procedure OptimisticRead($ref$)
    repeat
        $v_0 \leftarrow version[ref]$
        if $v_0 < 0$ then                    ▷ Negative means locked
            $v_0 \leftarrow$ WaitUntilUnlocked($ref$)
        end if
        $x \leftarrow field[ref]$
    until $v_0 = version[ref]$
    return $x$
end procedure

procedure LockedWrite($ref, x$)
    repeat
        $v_0 \leftarrow version[ref]$
        if $v_0 < 0$ then                    ▷ Negative means locked
            $v_0 \leftarrow$ WaitUntilUnlocked($ref$)
        end if
    until CompareAndSwap($version[ref], v_0, -v_0 - 1$)
    $field[ref] \leftarrow x$
    $version[ref] \leftarrow v_0 + 1$
end procedure
```

will be made to undo the previous writes. If a non-transactional volatile load has observed the uncommitted publication then it might access an object whose updates have been undone. 53 of the 66 `volatile` references found in classes that are in `java.*`, but not `java.util.concurrent.*`, are vulnerable to this problem. A weak model that correctly executes this idiom could be constructed, but it would still require the programmer to reason globally about the correctness of code.

Weak isolation sacrifices one of the most important characteristics of memory transactions: the ability to reason locally about correctness [13]. An analogy has been made with memory management, in which the system primitive of garbage collection replaces local reasoning about the whole-program property of reachability [7]. Without strong isolation both transactional and non-transactional accesses can produce surprising results because of an interaction in some other part of the code.

2.3 Isolation Barrier Implementation

Algorithm 1 shows implementations of read and write isolation barriers for our representative STM. OptimisticRead checks the version metadata before and after loading the field's value, retrying if a lock or race is detected. This mimics the optimistic conflict detection used by transactions, which maximizes scalability by using invisible reads [18]. LockedWrite acquires exclusive access to the field before storing, and then updates the version when the lock is released. This guarantees that any transaction that has read the old value will eventually detect that its read set is inconsistent.

The code for the read and write isolation barriers reveals why almost all STM implementations to date have adopted weak isolation: extra loads, extra stores, conditional branches, and atomic compare-and-swaps. In addition to the direct costs of the instructions, barriers complicate compiler optimizations and decrease the effectiveness of both software and hardware code caches.

2.4 Static Optimizations

Several static optimizations have been proposed that can reduce the overhead of isolation barriers, either by eliminating barriers that can be proved unnecessary or by aggregating operations to reduce the amortized synchronization cost. The analyses required to justify these optimizations vary from local to whole-program.

Escape Analysis

Concurrent access to a field can only occur after it has become reachable from another thread, so accesses to an unescaped object do not require a barrier. Shpeisman et al. use static escape analysis to eliminate isolation barriers. They also implement a dynamic escape analysis that allows faster execution of isolation barriers for object instances that are not dynamically reachable from a shared root object but for which the static analysis fails [21].

Barrier Aggregation

If a sequence of barriers is observed that protect the same memory location, they can be replaced with a single barrier that protects all of the accesses. This may merge multiple reads, multiple writes, or reads and writes (if the STM's write barriers also serve as read barriers). When conflict detection is performed at object granularity, barrier aggregation can often be used to merge consecutive accesses to different fields of the same object.

For example, the non-transactional code

```
    $field[ref] \leftarrow field[ref] + 1$
```

might be initially expanded into

```
    $x \leftarrow$ OPTIMISTICREAD($ref$)
    ACQUIRELOCK($ref$)
    $field[ref] \leftarrow x + 1$
    RELEASELOCK($ref$)
```

After barrier aggregation the sequence would be

```
    ACQUIRELOCK($ref$)
    $field[ref] \leftarrow field[ref] + 1$
    RELEASELOCK($ref$)
```

This optimization has been shown to be effective for transactional code [2]. Barrier aggregation has also been evaluated for reducing the overhead of isolation barriers in non-transactional code [21].

Static Not-Accessed-In-Transaction Analysis

If a field is never accessed inside an atomic region then no isolation barriers are required by non-transactional reads and writes. A static analysis that proves this *not-accessed-in-transaction* (NAIT) property can thus be used to remove isolation barriers[3]. Shpeisman et al. implement NAIT in a transactional compiler that provides strong isolation for Java [21]. Isolation barrier removal requires a whole-program analysis, which restricts the use of dynamic class loading and reflection. It also requires a conservative analysis, which may restrict its use for utility classes that are used both inside and outside transactions, e.g. `java.util.HashMap`.

3. Acceleration Using Optimization Hypotheses

We propose optimizing by forming a hypothesis about the dynamic interleaving of a field's accesses, and then choosing a set of barrier implementations customized for that pattern. Isolation and transaction barriers must cooperate to safely access a field. However *only barriers that return to the caller must synchronize with each other*. By allowing customized implementations to indefinitely block nonconforming accesses, we can make the remaining barriers smaller and faster. We provide a mechanism to rescue blocked threads, so an incorrect hypothesis only affects performance. All of our barrier implementations provide strong isolation whether or not the program conforms to the hypothesis, allowing us to use runtime profiling to safely guide our optimization.

As a motivating example consider a field protected by the barriers shown in Algorithm 2. These barriers allow direct non-transactional access to the field, but prevent access from inside an atomic region. No matter how they are invoked, the underlying access to the field will follow the NAIT access pattern and hence be strongly isolated. If the barrier invocations do not follow the NAIT pattern then the rescue mechanism must be invoked to resume forward progress.

Algorithm 2 Barriers customized for the optimization hypothesis NAIT.

```
procedure NONTXNREAD($ref$)
    return $field[ref]$                         ▷ Access OK
end procedure

procedure NONTXNWRITE($ref, x$)
    $field[ref] \leftarrow x$                          ▷ Access OK
end procedure

procedure TXNOPENFORREAD($ref$)
    ROLLBACKANDBLOCK              ▷ Nonconforming access
end procedure

procedure TXNOPENFORWRITE($ref$)
    ROLLBACKANDBLOCK              ▷ Nonconforming access
end procedure
```

An *optimization hypothesis* (OH) is an access pattern enforced by a customized barrier implementation. If and only if the invocations of the barrier conform to that access pattern the accesses will be passed through to the protected field or array element. We say that the OH or the barrier *admits* those accesses. If the barriers are invoked in a disallowed pattern then they block the calling thread, after first rolling back any enclosing transaction. *An OH is both a hypothesis about the customized barrier's invocation pattern and a guarantee about the actual memory accesses that will be performed by the corresponding specialized barrier.*

To rescue threads that have violated the optimization hypothesis of a field we use the hot swap feature of a managed runtime to replace all of the barriers for that field. Continuing the NAIT example, if an invocation of TXNOPENFORWRITE is detected for the field then all barriers for the field will be replaced with the unoptimized versions before the blocked thread is resumed. In Section 5 we discuss the details of safely hot swapping barriers without quiescing unaffected threads.

3.1 Will an OH Actually Allow Optimization?

Dynamic verification of an OH incurs extra costs compared to optimization based on a static proof: the cost of verifying that an access pattern is followed, and the cost of retaining the ability to revert to full isolation barriers. For NAIT (and for several other hypotheses introduced in Section 3.3) all of the verification code is *outside* the isolation barriers, so there is no checking cost when the OH is correct. Preserving the ability to revert to the full isolations barriers requires the system to add transactional metadata to objects even if their fields are currently hypothesized to be NAIT. If the field is never accessed in a transaction then this field is unused.

The complexity of a profitable OH is limited by the complexity of the original isolation barriers OPTIMISTICREAD and LOCKEDWRITE. If the specialized barriers are more expensive than the original (when weighted by their relative usage) then no acceleration will be achieved. This is especially limiting for stateful optimization hypotheses that require their customized read isolation barriers to modify metadata.

The cost to rescue a blocked thread from an incorrect OH is much higher than the savings from an individual barrier, so the system will not experience an overall speedup if rescues are common. If two optimization hypotheses admit barriers with similar performance characteristics, it is better to choose the OH that admits more executions.

[3] As originally described NAIT also refers to a not-written-in-transaction analysis that can only be used to remove read isolation barriers; we use NAIT to refer to the stronger property.

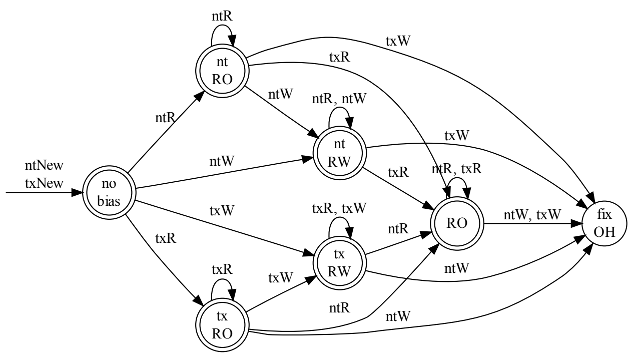


Figure 4. A finite state machine that enforces UAHA. The abbreviations are *nt* for non-transactional, *tx* for transactional, *RO* for read-only, and *RW* for read+write. The 'fix OH' state corresponds to a violation of the optimization hypothesis; the invoking thread will be blocked pending hot swap.

3.2 UAHA – Unmodified-after-heterogeneous-access

By moving the burden of proof from a static analysis to the barriers we gain the ability to check access patterns that are sensitive to an object instance's history. We introduce here *unmodified-after-heterogeneous-access* (UAHA), an access pattern that is expensive to check but general. In Section 3.3 we will describe a family of patterns (including NAIT) that allow a subset of the behaviors allowed by UAHA but that are practical to check.

Intuitively UAHA corresponds to a three-stage lifecycle for a field of an individual object. During the first stage no write barriers have been observed, so it is not known which type of reads should be considered heterogeneous. After a write barrier has been observed then the field is biased toward either transactional or non-transactional access. During this second stage both reads and writes are allowed, but they must be exclusively from inside an atomic region or exclusively from outside. The final stage is read-only, during which all reads are allowed. If both modes of reads are encountered then the read-only state must be entered.

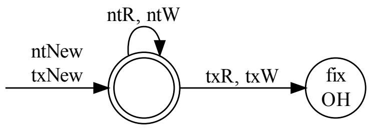
We observe that a customized implementation of NONTXNREAD can skip checking an object's version and lock metadata if every actual field store performed by a transaction happens-before the read isolation barrier's actual field load. Similarly, a customized implementation of NONTXNWRITE does not need to lock the object and increment the version number if every field store by the write isolation barrier happens-before every actual load or store to that field from a transaction. An instance is **UAHA** if:

1. No object instance is passed to both NONTXNWRITE and TXNOPENFORWRITE;
2. Every TXNOPENFORREAD happens-after every call to NONTXNWRITE; and
3. Every NONTXNREAD happens-after the completion of every transaction that called TXNOPENFORWRITE on the instance.

UAHA covers uses of the publication pattern that are not proved safe by a static escape analysis. It is even more general than dynamic escape analysis, because it detects actual shared accesses instead of reachability. Metadata is stored per object but optimization hypotheses concern individual fields, so maximum sensitivity is obtained by tracking heterogeneous access separately for each field.

Verifying that barrier invocations for a field follow the UAHA pattern requires that heterogeneous accesses (HA) be recorded. Until a write barrier is invoked we don't know whether HA corresponds to a read isolation barrier or transactional read barrier, so the checker must include states that record both types of reads. The resulting FSM has 6 states that return control to the caller (the 7th state is $\perp$), so it would require three bits of metadata per field (Figure 4). Synchronization is required for each state transition.

In contrast, consider the checking FSM for NAIT:



We can reproduce the customized barriers in Algorithm 2 by observing that zero bits are required to encode the single non-blocked state. NAIT implies UAHA, so clearly if the less general optimization hypothesis is sufficient we should use it instead.

There are several ways in which we can construct optimization hypotheses that imply UAHA, while having a lower checking overhead than the full FSM. We can reduce the number of states, reduce the number of state transitions, share some of the bits of the state representation between all fields of an object, and/or separate the starting states for transactional and non-transactional construction.

3.3 Our Family of Optimization Hypotheses

We augment each mutable object with a single transactional-creation (TC) bit and a fixed number of heterogeneous-access (HA) bits. TC is initialized to 1 if and only if the object is created inside an `atomic` block. Each field is mapped at compile time to one of the HA bits. If there are few mutable fields each one will be able to track heterogeneous accesses exactly; if an object has many fields then multiple fields will use the same HA bit, so false negatives are possible (this does not affect correctness). HA is used to record invocations of NONTXNREAD for transactionally created objects ($\text{TC} = 1$) and invocations of TXNOPENFORREAD for objects constructed outside an atomic region.

A field's full optimization hypothesis is composed of a simple OH p for object instances with $\text{TC} = 0$ and a simple OH q for objects where $\text{TC} = 1$, written $\langle p|q\rangle$. Thus an optimization hypothesis p that is not dependent on the construction context can be written $\langle p|p\rangle$, e.g. $\text{NAIT} = \langle\text{NAIT}|\text{NAIT}\rangle$. The simple optimization hypotheses are:

- **ANY** – any access. No limitations are made on the access pattern, and no acceleration is possible.
- **NONE** – no access. No barriers for the field may be executed.
- **RO** – read-only. No calls to NONTXNWRITE or TXNOPENFORWRITE are allowed. All write barriers for the field must have been eliminated by a static escape analysis.
- **NAIT** – not-accessed-in-transaction. No calls to TXNOPENFORREAD or TXNOPENFORWRITE are allowed for the field.
- **UATX** – unmodified-after-transactional-access. No calls to TXNOPENFORWRITE are allowed. TXNOPENFORREAD sets the field's HA bit, and NONTXNWRITE is allowed only if the HA bit has not yet been set.
- **NAOT** – not-accessed-outside-transaction. No calls to NONTXNREAD or NONTXNWRITE are allowed.
- **UATC** – unmodified-after-transaction-commit. NONTXNWRITE is not allowed. TXNOPENFORWRITE is not allowed unless the object is already locked by the current transaction and its version number is zero, indicating that it must have been locked during creation and never been committed[4].
- **UANT** – unmodified-after-non-transactional access. NONTXNWRITE is not allowed. NONTXNREAD sets the field's HA bit, and TXNOPENFORWRITE is allowed only if the HA bit has not yet been set.

[4] Alternately a bit could be added to the metadata to record the first commit.

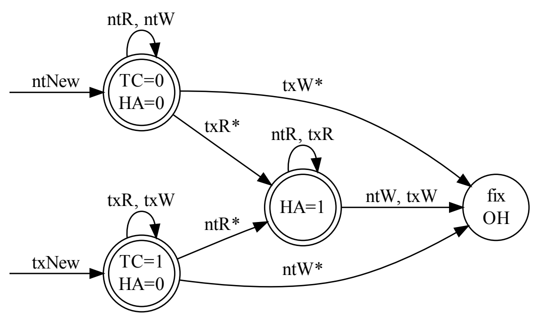


Figure 5. Enforcing the ⟨UATX|UANT⟩ optimization hypothesis. Edges with (*) are heterogeneous accesses.

Note that ⟨ANY|NONE⟩ is *not* the same as ⟨NAIT|NAIT⟩. The former specifies that any access is allowed to objects created outside a transaction, while the latter specifies that non-transactional access is allowed to all objects.

The UATX and UANT checkers use the HA bit to implement a two-stage lifecycle, so all actions that require $HA = 0$ must happen-before actions that set the HA bit. This means that the write barriers must lock fields that are changing, the heterogeneous read barriers may only set the HA bit on an unlocked object, and an atomic compare-and-swap must be used to set the HA bit. The first non-transactional read of an instance using a barrier customized for ⟨*|UANT⟩ will be more expensive than the full ⟨*|ANY⟩ barrier; subsequent non-transactional reads will be faster.

Because the definition of the HA bit is conditional on TC, the optimization hypotheses ⟨UANT|*⟩ and ⟨*|UATX⟩ are not supported. While we could successfully implement these possibilities with only a single bit by relaxing the definition of TC (perhaps calling it merely a 'mark bit'), we would require additional storage when performing our dynamic analysis (Section 6). Also, we exclude ⟨UATC|*⟩ because those hypotheses allow the same set of behaviors as ⟨RO|*⟩ (objects created outside a transaction can never meet UATC's criteria for TxnOpenForWrite).

3.4 Approximating UAHA with ⟨UATX|UANT⟩

The most general OH supported by our definitions of TC and HA is ⟨UATX|UANT⟩, whose checker is illustrated in Figure 5. This FSM can be derived from UAHA's checker (Figure 4) by immediately biasing transactionally created objects to the 'tx RW' state and non-transactionally created objects to the 'nt RW' state. This change is based on the intuition that the constructor and the first write to a field (not eliminated by escape analysis) are likely to be executed in the same context. The simplification reduces the storage cost from three bits per field to one bit per field plus one bit per object, and reduces the maximum number of state transitions for a field from three to one when the hypothesis is followed.

3.5 Synergy with Static Escape Analysis

The applicability of an optimization hypothesis can be substantially widened if a static escape analysis is first used to remove barriers that occur before an object might be concurrently accessed. For many fields this eliminates all write barriers [22]. Although this stationary field analysis most effective as a whole-program analysis, our implementation maintains compatibility with dynamic language features by limiting itself to an intraclass analysis performed during class loading.

To prevent confusion between optimizations performed statically at compile time and those performed optimistically at run time, we use the terms 'removed' or 'eliminated' only to refer to barriers statically proven to be unnecessary. Barriers that are unnecessary due to a currently valid optimization hypothesis are referred to as 'optimally accelerated.' Eliminated barriers will never be replaced, while optimally accelerated barriers may be hot swapped if the OH ceases to be correct.

3.6 Inheritance

Statically typed object-oriented languages such as C++ or Java associate every field access with a single declaration, even when inheritance is present. This means that no dynamic dispatch is required for field barriers even if the object's exact type is unknown. Optimization hypotheses are therefore associated with fields instead of classes.

Each instance contains only one set of associated STM metadata bits, so fields from subclasses must share access to the fixed number of HA bits. The mapping with the least bit aliasing can be computed when generating the barriers by counting the number of mutable instance fields declared by all superclasses. In the evaluated system 16 HA bits are provided for object instances.

3.7 Arrays

Arrays of references are covariant in Java, so an array isolation barrier typed to receive an `Object[]` may actually be accessing a `String[]`. This means that we must either use dynamic dispatch when calling array barriers or lose the benefits of optimization hypotheses that only hold for some array types. We choose the latter, since when successful it will yield better performance. For Java we track optimization hypotheses for 9 array types: the 8 primitive array types and `Object[]`. Multidimensional arrays are considered to be reference arrays.

An STM must choose the granularity at which data and metadata are associated. Object granularity is a widely accepted choice for instances, but the situation is less clear for arrays. Like adjacent fields of an object, consecutive array elements are likely to be accessed at the same time. We can optimize for this by using a single lock to protect all elements of an array. On the other hand, arrays form the basis of important shared data structures like `HashMap`. If the STM uses a single lock to cover such an array then concurrency will be lost. Our compromise is to use a single lock and HA bit to protect access to primitive arrays, and four locks and four HA bits to protect access to reference arrays. Reference array elements are mapped to the locks based on their index modulo four, so adjacent elements are protected by distinct locks.

4. Customized Barriers

Customized barriers provide strong isolation for all access patterns, but block threads that attempt accesses that do not conform to the optimization hypothesis for which they were specialized. The read and write isolation barriers and read and write transaction barriers for a field are all customized together.

4.1 Generating Barrier Specializations

Figure 6 gives an overview of the implementation required by isolation and transaction memory barriers. A barrier for a compound (TC-sensitive) OH is constructed by referring to the corresponding table entries for $TC = 0$ and $TC = 1$. If the entries specify the same operation, then no dynamic check of $tc[ref]$ is required. If they do not specify the same operation then an additional condition branch is required in the barrier.

Even after removing combinations that are not useful there are 44 barrier implementations. Rather than hand-code each of these possibilities we generate the specializations using traditional compilation techniques. We start with 'customizable' code that uses conditional tests of the optimization hypothesis to select between possible mechanisms for field access, then specialize for a single

OH	*non-txn read*	*non-txn write*	*txn read*	*txn write*
NONE	not allowed	not allowed	not allowed	not allowed
ANY	full	full	full	full
RO	direct	not allowed	no-op	not allowed
NAIT	direct	direct	not allowed	not allowed
UATX†	direct	full+check HA	set HA	not allowed
NAOT	not allowed	not allowed	full	full
UATC‡	direct	not allowed	no-op	check version
UANT‡	set HA+direct	not allowed	full	full+check HA

Figure 6. Implementations for simple optimization hypotheses, or for half of a compound OH after branching on the transactional creation (TC) bit. †Only if TC = 0 ‡Only if TC = 1

$TC{=}0 \backslash TC{=}1$	*ANY*	*RO*	*NAIT*	*NAOT*	*UATC*	*UANT*
ANY	$*$	✓	✓		✓	✓
RO	✓		✓+	✓+	✓++	✓+
NAIT	✓		✓++	✓+	✓+	✓+
UATX	✓		✓+	✓+	✓+	✓+
NAOT		✓+	✓+		✓+	✓+

Figure 7. Optimization hypotheses that allow isolation barriers to be accelerated: $*$ = base case, ✓ = faster for some invocations, ✓+ = faster and smaller, and ✓++ = optimal. Blank entries are subsumed.

Algorithm 3 Customizable read isolation barrier, before specialization.

```
 1: procedure NonTxnRead(ref)
        IMPL = { NONE ⇒ FAILURE,  ANY  ⇒ SLOW,
 2:              RO   ⇒ FAST,     NAIT ⇒ FAST,
                 UATX ⇒ FAST,     NAOT ⇒ FAILURE,
                 UATC ⇒ FAST,     UANT ⇒ HA_FAST }
 3:     $\langle p|q \rangle \leftarrow OH$            ▷ OH is constant during specialization
 4:     if $\text{IMPL}(p) = \text{IMPL}(q) \lor tc[ref] = 0$ then
 5:         $op \leftarrow \text{IMPL}(p)$
 6:     else
 7:         $op \leftarrow \text{IMPL}(q)$
 8:     end if
 9:     if $op = \text{FAILURE}$ then
10:         BlockUntilOHChange
11:         return NonTxnRead(ref)        ▷ Call new self after hot swap
12:     else if $op = \text{SLOW}$ then
13:         return OptimisticRead(ref)            ▷ Original slow barrier
14:     else
15:         if $op = \text{HA\_FAST} \land ha[ref] = 0$ then
16:             SetHA(ref)                       ▷ Uses CompareAndSwap
17:         end if
18:         return field[ref]                              ▷ Direct load
19:     end if
20: end procedure
```

OH by performing constant folding, constant propagation, dead code elimination, and common subexpression elimination[5].

To reproduce Algorithm 2's ⟨NAIT|NAIT⟩ read isolation barrier, for example, we perform the following transformations on the customizable barrier of Algorithm 3:

- Substitute the OH, after which p = NAIT and q = NAIT;
- Propagation and folding show that the conditional on line 4 evaluates to true;
- Additional propagation and folding shows that the conditionals on lines 9, 12, and 15 are all false; and
- Dead code elimination removes every line from NonTxnRead except line 18.

4.2 Useful Optimization Hypotheses

Not all optimization hypotheses are useful. Consider the OH ⟨ANY|NONE⟩. When accessing a non-transactionally created object (TC = 0) no acceleration is possible, since ANY requires full barriers. For objects created inside an atomic region (TC = 1) no access is allowed, despite all of the required code being present in the barrier. In addition, the conditional check required to detect nonconformance for the TC = 1 case makes the barriers larger and slower than those for the baseline ⟨ANY|ANY⟩.

[5] These transformations can also be performed in an integrated development environment with support for refactoring.

Codifying this reasoning leads us to compare *paid* costs. The size or execution speed of a barrier are only important for conforming accesses, because the cost of the hot swap triggered by an optimization failure dominates that case. Conversely, if oh_1 and oh_2 have the same costs for conforming accesses and oh_1 admits a strict superset of the behaviors admitted by oh_2 then there is no utility to ever choosing the second hypothesis. We say oh_1 *subsumes* oh_2 if:

1. oh_1 admits every execution allowed by oh_2;
2. For every execution admitted by oh_2, the barriers for oh_1 are no slower than the barriers for oh_2; and
3. Every barrier in oh_2 that does not always fail is at least as large as the corresponding barrier in oh_1.

Although the relative size and performance of different barriers is platform-specific, we can reasonably assume that adding a conditional branch to the underlying implementation results in a larger and slower barrier, and that a 'direct' barrier is no larger than a 'not allowed' barrier. We can then conclude that ⟨p|RO⟩ and ⟨p|NAIT⟩ subsume ⟨p|NONE⟩, that ⟨RO|p⟩ and ⟨NAIT|p⟩ subsume ⟨NONE|p⟩, that ⟨NONE|NONE⟩ is subsumed by any other OH, and that ⟨q|RO⟩ is subsumed by ⟨q|UATC⟩ for $q \in$ {RO, NAIT, UATX}. In Section 6.3 we will address the more difficult task of choosing the best OH when there is not a consensus among (1), (2), and (3).

Figure 7 shows which optimization hypotheses are useful for reducing the overhead of strong isolation. Of special note are the hypotheses ⟨NAIT|NAIT⟩ and ⟨RO|UATC⟩, for which all checking code is either outside the isolation barriers or in barriers that are not invoked unless the OH is incorrect.

5. Hot Swapping Barriers

Customized barrier implementations are faster when accesses conform to a field's optimization hypothesis, but they block threads that attempt a nonconforming access. Isolation barriers block internally, while transactional barriers roll back their transaction before blocking. To rescue a blocked thread we must relax the offending field's optimization hypothesis, hot swap its barrier implementations, and restart the offending access.

The HotSpot JVM has a robust on-stack deoptimization system to support speculative optimizations such as devirtualization. Deoptimization allows executing compiled code to be reverted to bytecode interpretation, reconstructing any eliminated local variables and stack frames. This system is required for aggressive optimization in an environment supporting dynamic class loading. It also underlies the JVM's hot swap feature, allowing hot swap to coexist with compiler transformations such as inlining and dead code elimination [15]. Java's hot swap mechanism is sufficient to replace all of the code for a field's isolation and transaction barriers. However we must ensure that the old and new barriers do not run simultaneously.

5.1 Ordering Requirements During an OH Change

Strong isolation cannot be guaranteed if barriers specialized for two optimization hypotheses run at the same time. The semantics of Java's hot swap, though, are to continue executing the old bytecodes for swapped methods that already have an activation record. This means that we can't merely invoke the JVM's hot swap and then proceed, because an old barrier implementation may still be active. To change an optimization hypothesis we must establish a happens-before relationship between all invocations of the old specialized barriers and all invocations of the replacements.

The simplest way to guarantee safety during the swap would be to quiesce all threads, blocking them outside isolation barriers and outside transactions (after waiting for commit or rollback). All old specializations happen before the quiesce is completed, the quiesce completion happens before the swap, and any subsequent barrier invocations will happen after the swap and hence use the new specialization. This stop-the-world approach is correct, but it blocks threads regardless of whether they require access to a changing barrier.

5.2 Swapping Without Stopping the World

Ideally we would like to quiesce only threads that might actually invoke a new specialization while a stale implementation is still active. In fact we can identify exactly these threads, by using hot swap to replace the stale barriers with a stub that quiesces the caller. Once this preliminary swap has been completed and all stale barrier implementations have finished we can install the new barriers.

After installing the required QUIESCE barriers the swapping thread increments a shared counter, the swap clock. When a thread is at a safepoint (somewhere outside a barrier) it copies the swap clock to a thread-local field. The swapping thread then polls the copies until it has observed that every thread has either become quiesced or passed a safepoint, at which time it can conclude that no copies of the old specialization are still active.

Safepoints inside an atomic region verify that the transaction has neither read nor written any of the fields affected by a hypothesis change before witnessing the updated swap clock. Some hypothesis changes could be safely tolerated but our current implementation simply rolls back any transaction that might be affected.

The situation is complicated by system calls, monitors, and by stale barriers blocked while acquiring an STM lock. Threads blocked on system calls or on monitor entry can be detected by checking the thread status; no swap clock witness is required for them because they cannot be executing a stale barrier. Blocking performed from inside a stale barrier specialization is handled by explicitly checking that the OH hasn't changed after the blocking call returns. This check is performed by comparing the return value of a trivial static method with a constant. The JIT will inline the static method call and perform dead code elimination on the branch. During an OH change the static method is replaced, resulting in deoptimization of the checking code and reappearance of the branch. If a barrier detects that it has been replaced, it forwards to the new implementation by making what appears to be a recursive call to itself.

Once it is certain that no stale implementations are still active (or rather that any remaining stale barriers are blocked and hence will behave properly once awoken) a second hot swap is used to install barriers customized for the new optimization hypothesis. The QUIESCE barriers are then instructed to unblock and forward to the new specializations, and execution continues.

6. Feedback-Directed Customization

To reap rewards from specialized barriers we must select optimization hypotheses that are specific enough to allow faster execution but general enough to admit all accesses actually performed by the program. Two basic strategies are possible: optimistic and pessimistic. The optimistic strategy starts with an OH that allows optimization (perhaps ⟨NAIT|NAIT⟩), and then reactively relaxes the OH as needed. The pessimistic strategy starts with a general OH and then proactively tightens it for fields identified as profitable opportunities. Side information may be used to choose an optimistic strategy for some fields and a pessimistic strategy for others.

If TC = 0

NC_NR	set by NONTXNREAD
NC_NW_BHA	set by NONTXNWRITE if HA = 0
NC_NW_AHA	set by NONTXNWRITE if HA = 1
NC_TR	set by TXNOPENFORREAD
NC_TW	set by TXNOPENFORWRITE

If TC = 1

TC_NR	set by NONTXNREAD
TC_NW	set by NONTXNWRITE
TC_TR	set by TXNOPENFORREAD
TC_TW_BC	set by TXNOPENFORWRITE if never committed
TC_TW_BHA	set by TXNOPENFORWRITE if HA = 0
TC_TW_AHA	set by TXNOPENFORWRITE if HA = 1

Figure 8. Observation bits. *NC* and *TC* abbreviate non-transactionally-created and transactionally-created, respectively. *BC* is before-commit, *BHA* is before-heterogeneous-access, and *AHA* is after-heterogeneous-access.

The optimal strategy for a field depends on the expected cost of hot swaps relative to the expected savings from accelerated barriers. Frequently accessed fields will be successfully accelerated by either the optimistic or pessimistic strategy. Infrequently accessed fields, however, do not provide the opportunity to recover the cost of an incorrect speculation, because each hot swap is much more expensive than an unaccelerated isolation barrier.

We therefore choose the proactive tightening (pessimistic) strategy. For all fields we start with generic ⟨ANY|ANY⟩ barriers that are augmented to record accesses. If many accesses are observed for a field we consider tightening it, because the computed OH is likely to be accurate and there are likely to be sufficient future accesses to recover the cost of the customizing swap.

6.1 Observation Bits

The RECORD barriers include the safe but slow functionality of generic isolation and transaction barriers, and they update the heterogeneous access (HA) bits. For each field declaration or array class we maintain an *observation word* and an invocation counter. The observation word contains 11 bits as described in Figure 8; each RECORD barrier invocation sets one bit and increments the counter. Reads and writes of profiling data are performed racily; the observation word will converge on the correct value and the invocation counter need only be a proxy for the activity of a field. The 11 observation bits correspond directly to the disallowed behaviors used to define our family of optimization hypotheses in Section 3.3. Figure 9 shows the mapping from observation word to allowed OH.

A background thread polls observations and initiates proactive tightening of an OH. Because hot swap occurs on a per-class basis, the incremental cost of customizing additional fields of a class is much smaller than the initial cost. We define two thresholds for initiating a proactive OH change, an *activation* threshold and an *inclusion* threshold (10,000 and 100 respectively for field barriers in the evaluated system, 100,000 and 1,000 respectively for array barriers). Only classes with a field that exceeds the activation threshold are rewritten, but when rewriting those classes we tighten all of the fields whose access count exceeds the inclusion threshold.

	NC_NR	NC_NW_BHA	NC_NW_AHA	NC_TR	NC_TW	TC_NR	TC_NW	TC_TR	TC_TW_BC	TC_TW_BHA	TC_TW_AHA
⟨ANY|*⟩	✓	✓	✓	✓	✓						
⟨RO|*⟩	✓			✓							
⟨NAIT|*⟩	✓	✓	✓						✓		
⟨UATX|*⟩	✓	✓		✓							
⟨NAOT|*⟩				✓	✓						
⟨*|ANY⟩						✓	✓	✓	✓	✓	✓
⟨*|RO⟩						✓		✓			
⟨*|NAIT⟩						✓	✓				
⟨*|NAOT⟩			✓					✓	✓	✓	✓
⟨*|UATC⟩						✓		✓	✓		
⟨*|UANT⟩						✓		✓	✓	✓	

Figure 9. Optimization hypotheses allowed by the 11 observations.

6.2 Persistence of Profiling Data

Profile data can be persisted across multiple executions of an application to decrease the number of hot swaps required during warm up. Because OH choices do not affect the correctness of the system this information can be used even after dynamic class loading or recompilation. For many applications this profile data is an oracle, eliminating the need to perform any swaps at all.

6.3 Choosing Between Allowed Hypotheses

The best OH for a given set of observations is the one whose barriers will produce the fastest execution of the overall program. Isolation barriers impact performance in many direct and indirect ways, so to actually build a system we must make simplifying assumptions.

The direct cost of a barrier is the expected number of cycles required to execute its instructions. Indirect costs include lost optimization opportunities, increased memory bandwidth and cache line contention, reduced instruction locality, and pressure on the virtual machine's compilation cache. The branches and volatile memory operations in a barrier limit the compiler's ability to reorder and eliminate memory references, while the overall size of a barrier has a large effect on the compiler's inlining choices.

We model the performance characteristics of an OH as a 23-element vector. For each of the four kinds of barriers {non-txn, txn} $\times$ {read, write} we include three elements: the relative execution ranking if TC $= 0$, the relative execution ranking if TC $= 1$, and the relative size of the barrier. The remaining 11 elements come from the observations that are compatible with the OH, with a 1 if the corresponding observation bit is allowed and a 2 if it is not. This definition means that smaller is better for each of the 23 elements. The execution orderings are as follows:

- NT read: direct $<$ test TC+direct $<$ set HA+direct $<$ full
- NT write: direct $<$ test TC+direct $<$ full $<$ full+check HA
- TX read: no-op $<$ test TC $<$ set HA $<$ full
- TX write: check version $<$ full $<$ full+check version

The execution cost of branching on TC for a compound OH comes mostly from fetching an object's metadata, so we separate it out only for barriers that would not otherwise perform that load. Barrier size is the number of words of bytecode in the evaluated system.

As an example, consider $cost(\langle\text{UATX}|\text{UANT}\rangle) =$
$\langle 2, 3, 52,\ \ 5, 0, 136,\ \ 3, 5, 47,\ \ 0, 4, 73,\ \ 1, 1, 2, 1, 2, 1, 2, 1, 1, 1, 2\rangle$

The first three elements of the vector indicate that the non-transactional read barriers for this OH are second fastest (test TC+direct) if TC $= 0$ and third fastest (set HA+direct) if TC $= 1$, and that they consist of 52 words of bytecode. The second three elements correspond to non-transactional write barriers. Because non-transactional writes are not allowed for this OH when TC $= 1$ the execution score for that case is given as 0.

Starting with the cost vector we then use the observation word to construct a *paid cost* vector by zeroing the execution cost elements where the execution is not compatible with the observation, and zeroing the size cost elements where neither TC $= 0$ nor TC $= 1$ executions are allowed. If $obs =$ NC_NR + NC_TR then $paid(\langle\text{UATX}|\text{UANT}\rangle, obs) =$
$\langle 2, 0, 52,\ \ 0, 0, 0,\ \ 3, 0, 47,\ \ 0, 0, 0,\ \ 1, 1, 2, 1, 2, 1, 2, 1, 1, 1, 2\rangle$

The corresponding paid cost vector for $\langle\text{RO}|\text{UATC}\rangle$ is
$\langle 1, 0, 5,\ \ 0, 0, 0,\ \ 1, 0, 2,\ \ 0, 0, 0,\ \ 1, 2, 2, 1, 2, 1, 2, 1, 1, 2, 2\rangle$

Element-wise comparison of these two vectors shows that $\langle\text{RO}|\text{UATC}\rangle$ is faster and smaller for every executed barrier (each of the first 12 elements is $\leq$), but that $\langle\text{UATX}|\text{UANT}\rangle$ admits more executions (each of the last 11 elements is $\geq$). We choose to compare paid cost vectors by reordering the elements and then comparing lexicographically, effectively giving absolute preference to the costs earlier in the order. This avoids the need to compute weights that allow comparison between metrics of different types.

When choosing the element preference order we give highest priority to execution time and second priority to barrier size. Because our focus is on reducing the overhead of strong isolation we prefer faster isolation barriers to faster transactional barriers, and we prefer acceleration for non-transactionally created instances to acceleration for transactionally created ones. Reads are typically more common than writes so we prioritize read performance [5]. The resulting order is:
$\langle 1, 4, 2, 5, 10, 7, 11, 8, 3, 6, 9, 12, 13, 14, 15, 18, 19, 20, 21, 22, 23, 16, 17\rangle$

This means that the first element of the paid cost vector is given priority, followed by the fourth element, and so on. This priority scheme reproduces the subsumption results from Section 4.2 automatically, and correctly prefers the 'perfect' hypotheses $\langle\text{RO}|\text{UATC}\rangle$ or $\langle\text{NAIT}|\text{NAIT}\rangle$ whenever they are compatible with the observation word. We precompute the mapping from obs to optimal OH and store it in a lookup table with 2048 entries.

7. AJ: a Bytecode-Rewriting Java STM in Java

To evaluate the feasibility of dynamic barrier customization we implemented it in AJ, a strongly isolated STM for Java. AJ is written in Java and uses the ASM toolkit [4] to rewrite bytecodes during class loading. Transactional metadata is added to classes containing mutable fields, methods are split into non-transactional and transactional version, and barriers are inserted. Reflection and dynamic class loading are allowed. Conditional retry, watch sets, and task composition are provided. AJ does not extend the Java language; rather it provides a static method `TM.atomic(Runnable task)` that causes `task.run()` to be executed atomically.

AJ is implemented almost entirely in Java by making use of Sun's `sun.misc.Unsafe` extension. The exception is a small JNI stub that allows the hot swap system to call JVMTI's `GetCurrentContendedMonitor` function, required to detect threads blocked on the hidden monitor that guards a class' static initializer. AJ has been tested on Sun's J2SE™1.6.0_06 on the `i686`, `x86_64`, and `sun4v` architectures, and on J2SE™1.7.0-ea-b29 on the `x86_64` architecture. It has been tested under Linux and Solaris.

Barriers are inserted into both application and library code. To reduce the number of circularity issues the JRE's classes are instrumented ahead of time. All other code is instrumented during class loading. 16 HA bits are provided for object instances. We map array instances to version and lock metadata by hashing. On 32-bit platforms we must assume TC = 1 and HA = 1. However, on 64-bit platforms current versions of HotSpot have 25 unused bits in each object's header, allowing us to store the TC and HA bits exactly for arrays.

Class initialization cannot be rolled back because the JVM maintains C++ shadows of the fields in `java.lang.Class`, outside AJ's reach. We therefore suspend transactions during execution of static initializers. Barriers are implemented as static method calls to synthetically generated helper classes. Barrier customization is accomplished by using the Instrumentation API to swap these helper classes.

AJ's implementation of dynamic optimization has reasonable overheads, and the cost of hot swaps during warm up can be reduced or eliminated by persisting profile data across executions. AJ duplicates analyses already performed by the compiler, must perform method splitting for all loaded classes, and cannot inform the JVM of invariants that are preserved during swaps. While the per-barrier and per-swap overhead of the STM would be reduced by tighter integration with the underlying JVM, AJ's all-Java implementation allows us to leverage Sun's mature hot swap implementation. We have implemented all of the optimization hypotheses from Section 3.3, the hot swap mechanism described in Section 5, the feedback-directed customization of Section 6, and the profile persistence from Section 6.2. Barrier customization is supported for both application and JDK classes.

8. Evaluation

We evaluated costs of strong isolation without optimizations, effectiveness of our specialized barriers at reducing those costs, and overheads involved in profiling and customizing barriers dynamically. We used benchmarks with no transactions and benchmarks that spend substantial time both inside and outside atomic regions. We measured performance and scalability impacts by running each of the benchmarks without isolation barriers, with unoptimized barriers, with specialized barriers installed before class loading, and with specialized barriers installed by hot swap based on runtime feedback. Dynamic barrier customization reduced the overhead of providing strong isolation in an STM and lowered STM overheads during transactional execution.

We performed experiments on computers with two quad-core 2.33Ghz Intel®Xeon®CPUs and 32GB of shared memory. We used Linux x86_64 kernel 2.6.18 and the 64-bit Server VM in Sun's Java™SE Runtime Environment, build 1.7.0-ea-b29, with a 1GB heap. We ran each configuration five times and averaged the results. Error bars illustrate the standard deviation of the samples.

8.1 Overheads of Basic Isolation Barriers

We measured the overhead of unoptimized isolation barriers by executing non-transactional benchmarks from the DaCapo suite with and without isolation barriers [3]. Of the fixed optimizations discussed in Section 2.4 only intraclass static escape analysis was used. To account for JVM warm up we timed the third repetition of the DaCapo programs. Figure 10 shows the additional execution time when unoptimized isolation barriers are used, normalized to the execution time of a program with no barriers, STM transformations, or AJ overheads. Execution time is also shown for programs that contain isolation barriers only for field accesses (object and static fields) and for programs that contain isolation barriers only for accesses to array elements. To measure the impact of isolation barriers on scalability we also ran SpecJBB2005 in these configurations with thread counts from 1 to 8. Each invocation of the benchmark produces scores for all thread counts, with an extra single-threaded execution providing a warm up for the other values. Figure 11 shows the gain in the average execution time of a business operation, normalized to the average execution time with no STM overheads for the corresponding thread count.

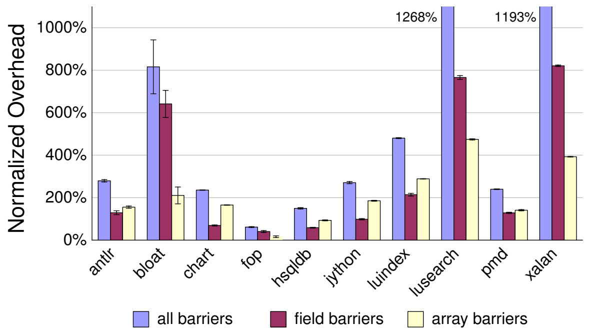


Figure 10. Overheads from unoptimized isolation barriers during non-transactional execution of the DaCapo benchmarks, normalized to performance with no STM.

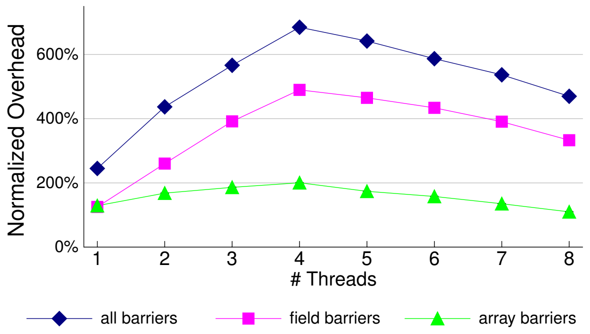


Figure 11. Overheads from unoptimized isolation barriers during non-transactional execution of SpecJBB2005, normalized to the performance with no STM for the corresponding thread count.

As expected, programs containing unoptimized isolation barriers were significantly slower and less scalable. Barriers protecting both field and array accesses were significant sources of overhead, with the ratio of importance varying between applications. `fop` experienced the smallest isolation barrier overhead (59%) because it performs a great deal of I/O. `lusearch` experiences the most slowdown (∼14 × slower) because its many query threads are impacted by the sequential and scaling overheads of isolation barriers.

8.2 Overheads of Dynamic Barrier Customization

We evaluated the cost of feedback-directed optimization by executing the DaCapo and SpecJBB2005 benchmarks with isolation barriers chosen by an oracle and with isolation barriers customized at runtime using the analysis of Section 6. We further examine the source of overheads by running the benchmarks with STM's method splitting and metadata injection but no isolation barriers, with and without swap safepoints. Figure 12 shows the performance overhead of the DaCapo benchmarks relative to an execution with no barriers or STM support. The 'weak' system includes method splitting and STM metadata for object instances, but has no isolation barriers. The 'weak+safepoints' system adds swap safepoints, which are required for safe hot swap of barriers. The 'oracle strong' system adds isolation barriers that are customized during class loading for the ⟨NAIT|NAIT⟩ optimization hypothesis. The 'dynamic strong' system starts with unoptimized RECORD barriers and uses hot swap to customize frequently used barriers using the scheme from Section 6. Figure 13 shows scaling of these configurations for SpecJBB2005.

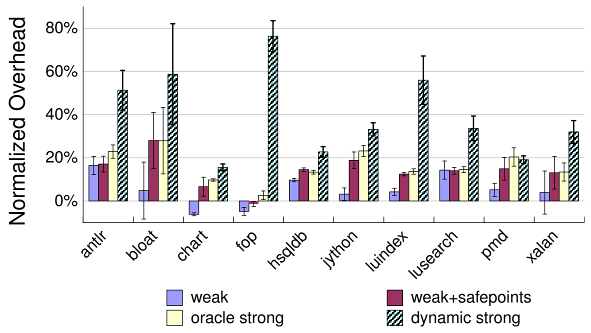


Figure 12. Isolation barrier overheads with oracle and feedback-directed barrier customization, normalized to performance with no STM.

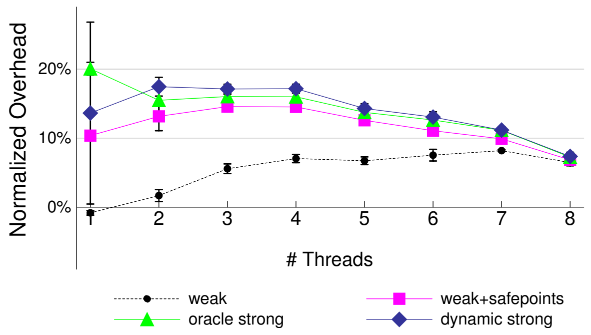


Figure 13. Isolation barrier overheads for non-transactional execution of SpecJBB2005 with oracle and dynamic barrier customization, normalized to performance with no STM for the corresponding number of threads.

	# swaps	*# fields*	*msec/field*
antlr	72.8	426	13.9
bloat	110.0	425	36.8
chart	76.2	427	22.2
fop	71.4	384	16.2
hsqldb	49.0	375	18.9
jython	83.4	498	32.5
luindex	91.6	429	10.6
lusearch	30.6	300	26.2
pmd	78.8	345	22.6
xalan	97.8	892	27.6
SpecJBB2005	63.0	339	14.8

Figure 14. Hot swaps performed by feedback-directed optimization during the executions from Figures 12 and 13 (averages).

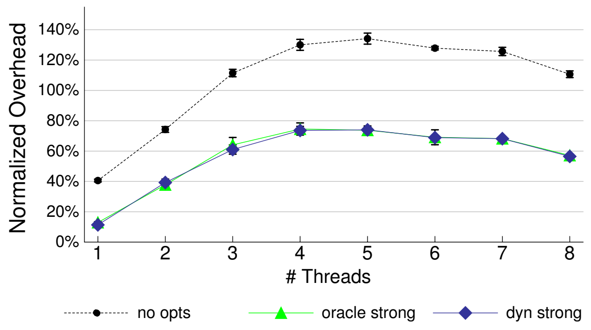


Figure 15. Overhead of strong isolation relative to weak isolation for SpecJBB2005 modified to use atomic regions and lazy transaction log construction. Increases in average business op execution time are shown, normalized to a weakly isolated STM with the same thread count.

The 'oracle strong' overheads represent an execution in which the system uses persistent profiling data to correctly select the initial optimization hypotheses, as described in Section 6.2. For many situations the oracle behavior is feasible even in deployed environments. The average overhead for DaCapo in this case was 16% and for fully threaded SpecJBB2005 (8 threads for 8 processors) it was 7%. We tested the stability of this overhead by measuring the 15$^{\text{th}}$ DaCapo iteration instead of the third; no improvement was seen. 'Oracle strong' demonstrates that isolation barriers can be efficiently optimized while preserving the ability to deoptimize.

The average difference between 'weak+safepoints' and 'oracle strong' (counting SpecJBB2005 peak scores as a single datum) is 2.2%. This indicates that HotSpot does an excellent job of inlining barriers optimized for ⟨NAIT|NAIT⟩. The average overhead from method splitting and metadata in object instances is 5.3%. Adding swap safepoints introduces an additional 8.5% overhead. Tightly coupling the STM and JVM would reduce all of these overheads.

The 'dynamic strong' values show that feedback-directed barrier optimization is practical for applications with a warm up period. The average overhead for the third DaCapo iteration was 24% higher than oracle OH. Reasonably, hot swap's impact is largest for programs with the shortest execution (`fop` is shortest, followed by `luindex` then `antlr`). All benchmarks except `fop` recovered their swapping costs during the first iteration. `fop`'s first iteration with feedback-directed optimization was 16% slower than the first iteration using unoptimized isolation barriers. Dynamic optimization results in a peak SpecJBB2005 score within 1% of the oracle's score.

Figure 14 shows average counts and elapsed times for the 'dynamic strong' executions. On average, 5.9 fields had their optimization hypothesis changed during each hot swap. 'Dynamic strong' demonstrates that the feedback-directed optimization quickly provides most of the benefits of an oracle static analysis, and that the extra costs incurred by hot swap can be recovered.

8.3 Optimizations in a Mostly-Transactional Program

To evaluate the cost of providing strong isolation for a program that has both transactional and non-transactional execution, we replaced some of SpecJBB2005's `synchronized` blocks with atomic regions. The benchmark simulates a multi-tier system by performing synchronized transactional and then performing an unsynchronized presentation phase. These phases are contained in the `process` and `processTransactionLog` portions, respectively. The presentation workload consists of creating XML and simulating a text-based client terminal using many character arrays. We modified the benchmark so that the setup and `process` portions of each benchmark transaction executed in an STM transaction and the `processTransactionLog` remained non-atomic. This split causes the benchmark to spend between 60% and 85% of its non-startup CPU time in atomic regions.

SpecJBB2005 constructs the character arrays that model the client terminal (the 'transaction log') inside `process` and fills those arrays in `processTransactionLog`. Our initial naive transactification thus causes `char[]` to receive an optimization hypothesis of ⟨ANY|ANY⟩, preventing acceleration for the most common barrier type. Our goal was not to evaluate automatic parallelization or lock elision, so we moved the offending construction code outside of the atomic section by using the lazy initialization pattern. After this transformation `char[]` allows an OH of ⟨UATX|ANY⟩. In this use case lazy initialization would also be required to optimize using NAIT or dynamic escape analysis.

$\mathbf{TC} = 0$

NT Read **14.5%**	(%)	**NT Write** **3.8%**	(%)	**TX Read** **14.9%**	(%)	**TX Write** **0.6%**	(%)
⟨UATX|ANY⟩	45	⟨UATX|ANY⟩	56	⟨ANY|ANY⟩	54	⟨ANY|ANY⟩	100
⟨UATX|UATC⟩	13	⟨UATX|NAIT⟩	13	⟨UATX|NAIT⟩	33		
⟨NAIT|ANY⟩	12	⟨ANY|ANY⟩	13	⟨RO|UATC⟩	12		
⟨RO|UATC⟩	11	⟨NAIT|ANY⟩	12				
⟨UATX|NAIT⟩	8	⟨NAIT|UATC⟩	6				
⟨NAIT|UATC⟩	7						

$\mathbf{TC} = 1$

NT Read **28.8%**	(%)	**NT Write** **6.7%**	(%)	**TX Read** **26.1%**	(%)	**TX Write** **4.6%**	(%)
⟨RO|UATC⟩	45	⟨ANY|ANY⟩	41	⟨ANY|ANY⟩	39	⟨NAIT|UATC⟩	67
⟨UATX|ANY⟩	22	⟨UATX|ANY⟩	38	⟨RO|UATC⟩	28	⟨ANY|ANY⟩	12
⟨ANY|ANY⟩	13	⟨NAIT|ANY⟩	21	⟨NAIT|UATC⟩	19	⟨ANY|ANY⟩	6
⟨UATX|UATC⟩	12			⟨UATX|UATC⟩	6	⟨UATX|UATC⟩	6
⟨NAIT|ANY⟩	6			⟨NAIT|ANY⟩	5		

Figure 16. Relative barrier invocation counts for 'dynamic strong' execution of SpecJBB2005 modified to use atomic regions and lazy transaction log creation. OH percentages are relative within the corresponding column. Hypotheses accounting for $< 5\%$ of a column are not shown.

Figure 15 shows the performance of SpecJBB2005 modified to use atomic regions, normalized to single-threaded performance with weak isolation and no barrier customization. We observe that as in the previous experiments, the feedback-directed analysis performs almost identically to the oracle analysis once the system has warmed up. Customization of the isolation barriers improves the peak score by 34% compared to unoptimized strong isolation. Some of this improvement results from acceleration of the isolation barriers and some from acceleration of the transactional barriers. Because all barriers must work together to verify the OH, it is not possible to apply these optimizations separately. Figure 16 shows the barrier invocation counts from a 'dynamic strong' configuration. The relatively large performance difference between weak and strong isolation in this experiment comes from write isolation barriers. When we examine the left half of the 'NT Write' hypotheses in the upper half of the table (TC $= 0$) and the right half of the hypotheses for the lower half, we see that almost all write isolation barriers must use locking.

8.4 Acceleration of Transactions

Although we are primarily motivated to produce a practical strongly isolated STM, we can also use our feedback-directed barrier optimization to accelerate a weakly isolated STM. No changes to the feedback or OH selection algorithm are required, because if no isolation barriers are inserted then the system observes only transaction barriers and confines itself to optimizing those. Dynamic optimization improved the peak SpecJBB2005 score by 31% over the weakly isolated transactional execution. 55% of the read barrier invocations and 75% of the write barrier invocations went to barriers customized for ⟨RO|UATC⟩. This OH is almost entirely responsible for the speedup, as the only other hypothesis with substantial use was ⟨ANY|ANY⟩.

9. Related Work

Dynamic NAIT – Recently published concurrent work by Schneider et al. explores a nearby point in the design space of strongly isolated STMs [19]. This work adds a dynamic not-access-in-transaction analysis (D-NAIT) to an STM-enabled JVM. This is the same JVM used by Shpeisman et al. to evaluate static whole-program NAIT analysis [21].

D-NAIT considers only the removal of barriers, rather than selection among alternate non-empty implementations. This restriction, along with the cooperation of the JIT (and the negligible cost of NOP instructions on a modern architecture) allow for barrier patching at a lower cost than in our evaluation system. The reduced decision space and the reduced cost of rescue lead them to prefer an initially tight hypothesis (⟨NAIT|NAIT⟩) that is reactively relaxed as needed. The reactive strategy increased AJ's swap time by 48% for the transactified SpecJBB2005.

Dynamic escape analysis plays a similar role in the D-NAIT system to the TC and HA bits in ours, allowing short-cut execution of barriers for instances that meet a dynamic safety property. Because dynamic escape analysis marks objects as escaped before they are shared, it does not need to synchronize access to an instance's state. This allows acceleration of both read and write isolation barriers for unescaped instances, as there is no potential race between observing that an instance has not escaped and a subsequent write to a field. ⟨UATX|*⟩ or ⟨*|UANT⟩ are potentially more accurate, detecting actual sharing instead of potential sharing and operating at field granularity. The tradeoff, however, is that homogeneous write barriers must perform extra work to guard against concurrent setting of the HA bit. HA bits need only be read or updated for optimization hypotheses involving UATX or UANT, while dynamic escape analysis must be performed for all objects.

Schneider et al. stop the world during barrier patching, while we propose a two-swap scheme that only blocks threads if they access a field whose optimization hypothesis is changing. D-NAIT handles optimization failures for all barriers of a method on entry, which reduces the number of swaps required but slightly decreases the effectiveness of the analysis. Per-method generation of optimization failures is not possible in our system, at least not for any optimization hypotheses that reference the TC or HA bits. D-NAIT's tight integration with the JVM (in particular with the GC, which already has code to efficiently quiesce all threads) reduces the performance penalty of stopping the world. Our two-swap scheme would benefit similarly from a tight coupling, and may have scaling advantages.

Static NAIT – The analysis of Shpeisman et al. can produce smaller and faster code than dynamic barrier customization, even when barrier customization is seeded with the oracle OH [21]. The static analysis is conservative, so it does not need to retain the ability to revert to full isolation barriers. Shpeisman et al. also optimize for fields that are not written in a transaction. This access pattern is not accepted by UAHA, the most general optimizable OH we consider. Shpeisman et al.'s dynamic escape analysis plays a similar role in their system to the TC and HA bits in ours, providing for a partial speedup based on an instance's dynamic history.

Automatic Data Partitioning – The family of optimization hypotheses we present are brittle when applied to arrays. `char[]` is particularly important because its use is ubiquitous via `String` and `StringBuilder`. The coupling between type and OH in our system is a degenerate form of pointer analysis, in which we assert that accesses to different fields cannot alias. More sophisticated analysis may ease the difficulties we encountered in Section 8.3.

Riegel et al.'s automatic data partitioning approaches the aliasing problem from the analysis end, constructing data partitions using pointer analysis that include multiple types in each partition [16]. This results in tens of partitions per application, rather than the thousands created our system. Their dynamic selection of the STM used for each partition is similar to our choice of optimization hypothesis, but they dynamically dispatch to the currently chosen STM in each barrier instead of swapping in customized barriers, and they only attempt to accelerate transactional barriers. Interestingly, they consider not only the accesses allowed by an STM specialization but also the relative costs. To extrapolate this idea to our system we would construct multiple barrier implementations for ⟨ANY|ANY⟩, with implementations tailored for high and low contention and for read-mostly or write-heavy access pattern.

Biased Locking – Our family of optimization hypotheses is unable to provide acceleration for fields that are written both inside and outside a transaction, even if those accesses don't break strong isolation. The rebiasing techniques of biased locking might be used to switch the allowed accesses without requiring synchronization by each barrier, because the rebias causes a happens-before relationship to exist between accesses with different biases [17].

10. Conclusion

Many of the simplicity and safety advantages of memory transactions are lost if strong isolation is not provided, but the advantages are also lost if the performance penalty of isolation barriers is too high. We have described how to dynamically optimize isolation barriers based on feedback gathered during execution, and we have shown that the overhead of performing these optimizations at runtime is reasonable. Our customized barriers take responsibility for guaranteeing strong isolation, allowing optimizations to be performed in the presence of dynamic class loading. Dynamic barrier customization is also effective at accelerating STM's transactional execution under weak and strong isolation.

We reduced the overhead imposed by strong isolation from 505% to 38% for 11 non-transactional benchmarks, and then used persistent feedback data to further reduce that overhead to 16%. We accelerated a benchmark with mixed transactional and non-transactional execution by 31% under weak isolation and 34% under strong isolation.

Acknowledgments

This work was supported by the Stanford Pervasive Parallelism Lab, by Dept. of the Army, AHPCRC W911NF-07-2-0027-1, and by the National Science Foundation under grant CNS–0720905.

References

[1] Intel C++ STM Compiler, Prototype Edition. `http://software.intel.com`.

[2] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

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

[4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems*, 2002.

[5] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th International Conference on High-Performance Computer Architecture*, February 2006.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC'06: Proceedings of the 20th International Symposium on Distributed Computing*, March 2006.

[7] D. Grossman. The transactional memory / garbage collection analogy. *SIGPLAN Not.*, 42(10):695–706, 2007.

[8] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *MSPC: Workshop on Memory Systems Performance and Correctness*, October 2006.

[9] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[10] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[11] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC: Workshop on Memory Systems Performance and Correctness*, October 2006.

[12] *Java Specification Request (JSR) 133: Java Memory Model and Thread Specification*, September 2004.

[13] V. Luchangco. Against lock-based semantics for transactional memory. In *SPAA '08: Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 98–100, New York, NY, USA, 2008. ACM.

[14] V. Menon, S. Balensieger, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *SPAA '08: Proceedings of the 20th ACM Symposium on Parallel Algorithms and Architectures*, 2008.

[15] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} Server Compiler. In *JVM '01: Proceedings of the Java^{TM}Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[16] T. Riegel, C. Fetzer, and P. Felber. Automatic data partitioning in software transactional memories. In *SPAA '08: Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 152–159, New York, NY, USA, 2008. ACM.

[17] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 263–272, New York, NY, USA, 2006. ACM.

[18] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

[19] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, October 2008. ACM.

[20] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Transactions and privatization in delaunay triangulation. In *PODC '07: Proceedings of the 26th ACM symposium on Principles of Distributed Computing*, pages 336–337, NY, USA, 2007. ACM.

[21] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[22] C. Unkel and M. S. Lam. Automatic inference of stationary fields: a generalization of java's final fields. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 183–195, New York, NY, USA, 2008. ACM.

[23] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.