

Dynamic Optimization for Efficient Strong Atomicity

Florian T. Schneider
Department of Computer Science
ETH Zurich, Switzerland
florian.schneider@inf.ethz.ch

Vijay Menon
Google
Seattle, WA 98103
vsm@acm.org

Tatiana Shpeisman
Ali-Reza Adl-Tabatabai
Intel Corporation
Santa Clara, CA 95054
{tatiana.shpeisman,ali-reza.adl-
tabatabai}@intel.com

Abstract

Transactional memory (TM) is a promising concurrency control alternative to locks. Recent work [30, 1, 25, 26] has highlighted important memory model issues regarding TM semantics and exposed problems in existing TM implementations. For safe, managed languages such as Java, there is a growing consensus towards strong atomicity semantics as a sound, scalable solution.

Strong atomicity has presented a challenge to implement efficiently because it requires instrumentation of non-transactional memory accesses, incurring significant overhead even when a program makes minimal or no use of transactions. To minimize overhead, existing solutions require either a sophisticated type system, specialized hardware, or static whole-program analysis. These techniques do not translate easily into a production setting on existing hardware.

In this paper, we present novel dynamic optimizations that significantly reduce strong atomicity overheads and make strong atomicity practical for dynamic language environments. We introduce analyses that optimistically track which non-transactional memory accesses can avoid strong atomicity instrumentation, and we describe a lightweight speculation and recovery mechanism that applies these analyses to generate speculatively-optimized but safe code for strong atomicity in a dynamically-loaded environment. We show how to implement these mechanisms efficiently by leveraging existing dynamic optimization infrastructure in a Java system. Measurements on a set of transactional and non-transactional Java workloads demonstrate that our techniques substantially reduce the overhead of strong atomicity

from a factor of 5x down to 10% or less over an efficient weak atomicity baseline.

Categories and Subject Descriptors D.1.3 [Programming techniques]: Concurrent Programming—Parallel Programming; D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Optimization, Run-time environments

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

1. Introduction

Transactional memory (TM) [18] has been suggested as an alternative to lock-based synchronization. Locks place many burdens on the programmer: Locks provide only indirect mechanisms for enforcing atomicity and isolation, and to make a program scale using locks, a programmer has to reason about complex details such as fine-grain synchronization and deadlock avoidance. TM promises to remove these burdens from the programmer and to automate them. Recent work [14, 2] has shown how TM can be tightly integrated into mainstream languages and how TM can provide scalability that competes with fine-grain locks.

But TM has arguably been a step back in the semantics it provides to the programmer. Most software transactional memory (STM) systems implement *weak atomicity* [3]: they provide no ordering or isolation guarantees between transactions and non-transactional memory accesses, and they cannot guarantee serializability of transactions in presence of potentially conflicting non-transactional accesses. Weak atomicity has surprising semantic pitfalls and in some cases, leads to incorrect execution for programs that are correctly synchronized under locks[30].

In contrast, an STM system that implements *strong atomicity* [3, 30] guarantees serializability of transactions in the presence of non-transactional memory accesses. Strong atomicity avoids the pitfalls of weak atomicity and provides cleaner semantics. The primary obstacle to strong atomicity is the performance overhead required to enforce it. To implement strong atomicity, an STM system must instrument

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
Copyright © 2008 ACM 978-1-60558-215-3/08/10...\$5.00

Thread 1	Thread 2
<pre>foo() { while (..) { ... = f.x; f.x = ...; } }</pre>	<pre>bar() { atomic { ... = f.x; } }</pre>

Figure 1. Strong atomicity example

non-transactional memory accesses with read or write barriers, incurring significant overhead even when a program makes minimal or no use of transactions. Past work has shown how to implement strong atomicity efficiently using static whole-program analyses, but such analyses are impractical in a dynamic language environment, and without these optimizations, the high overheads of strong atomicity — up to 5x on some workloads — make it nonviable for mainstream adoption.

This paper presents a dynamic optimization framework that reduces strong atomicity overheads and makes strong atomicity practical for dynamic language environments such as Java. Our approach speculatively eliminates strong atomicity barriers based on incremental analyses performed by the JIT. The JIT can later detect mis-speculation and recover by dynamically patching mis-speculated accesses. The approach can take advantage of existing dynamic profile-guided optimization and recompilation infrastructure to improve its speculative optimization. In contrast to prior work on optimizing strong atomicity, our approach does not depend on static whole-program analysis; instead, it takes advantage of existing dynamic optimization infrastructure inherent in Java VMs.

We will use the example program in Figure 1 to illustrate our approach. Suppose that Thread 1 initially invokes method `foo` for the first time and triggers the JIT to compile it. Because `foo` executes outside of a transaction, the JIT would normally generate strong atomicity read and write barriers for the two accesses to `f.x`. Using our approach, the JIT optimistically generates code with no strong atomicity barriers since it has not yet seen any transactional accesses to `f.x` that could cause a conflict. If Thread 2 now invokes method `bar` for the first time, the JIT will compile it and discover a transactional read of `f.x` that may conflict with non-transactional writes to that field. Before `bar` is compiled and executed, the JIT must invalidate code based upon now incorrect assumptions. The JIT stops Thread 1 and patches the write to `f.x` in `foo` so that it performs a full write barrier. It does not patch the read to `f.x` in `foo` as there are still no transactional writes to `f.x`. Once patching is done, Thread 2 can safely execute `bar` without violating strong atomicity.

We make the following contributions:

- We present a dynamic *not-accessed-in-transaction* analysis (D-NAIT) that optimistically and incrementally tracks the memory locations read or written inside transactions. We augment D-NAIT with an incremental alias analysis that optimistically tracks whether reference fields are unique or can alias.
- We present a low-overhead mechanism called *phantom barriers* that allows the JIT compiler to eliminate strong atomicity barriers speculatively based on D-NAIT’s analysis. Phantom barriers convert to standard barriers via dynamic patching if D-NAIT’s incremental analysis later invalidates the speculation. We demonstrate how to reduce the cost of mis-speculation by leveraging existing profile-guided optimization and recompilation infrastructure available in most high-performance Java VMs.
- We measure the performance of our approach in the context of a Java STM system on a set of transactional and non-transactional benchmarks. We show that our approach can significantly reduce the overhead of strong atomicity from a factor of 5 to less than 10% in most cases.

2. Background

Prior work [13, 30, 1, 26] provides detailed arguments in favor of strong atomicity in STM. In this section, we summarize the motivations for strong atomicity, the current solutions towards its implementation in STM, and the open performance challenges that remain.

2.1 Motivation for strong atomicity

From a programmer’s perspective, strong atomicity provides stronger and more intuitive semantics than weak atomicity. In Figure 2, for example, Thread 1 tests a safety property (i.e., that the object `x` is non-null) and executes an instruction that may otherwise raise an exception. Under strong atomicity, this code is properly isolated and will never fail. But under weak atomicity (or locks), malicious or buggy code can alter that safety property in the middle of Thread 1’s transaction and trigger an exception. More generally, this example represents the commonly known *Time-Of-Check-To-Time-Of-Use* (TOCTTOU) [4] vulnerability in system code and motivates strong atomicity semantics in high security or reliability situations.

Although the above example can break for both weak atomicity and locks, recent work [30] has shown that weak atomicity actually provides weaker isolation and ordering guarantees than locks. Under weak atomicity, privatization and consistency issues [1, 25, 26] can introduce incorrect behavior even in correctly synchronized programs that are free of data races. Additionally, memory models for safe languages such as Java provide strong behavioral guarantees even in the presence of data races. In Figure 3, a weakly

```

Thread 1          Thread 2
atomic {
  if(x != null)
    x.f = ...;
}
← x = null;

```

Figure 2. TOCTTOU example: Under weak atomicity, malicious or buggy code can introduce a fault in Thread 1.

Initially x==0 and y==0	
Thread 1	Thread 2
atomic {	
if (y==0)	if (x==1)
x = 1;	y = 1;
/*abort*/	
}	
Can x==0?	

Figure 3. SDR example from [30]: Under weak atomicity, x can be 0. Under strong atomicity or locks it cannot.

atomic STM with in-place updates can produce a final result of $x == 0$, even though it is impossible in the equivalent lock-based program under the Java memory model.

Weakly atomic STM implementations can be designed to provide *as-strong-as-lock* semantics [25] without strong atomicity even in safe languages such as Java. Nevertheless, this requirement severely curtails the STM design space. Current results show a significant overhead and scalability cost to this approach [25].

Alternatively, one can avoid the pitfalls of weak atomicity by making sure that a program never accesses the same data both inside and outside a transaction. Recent work has proven that segregating data in this manner in a weakly atomic STM gives the same semantics as strong atomicity [26]. This requires either programmer conventions or a special type system that segregates data as was done in the STM for Concurrent Haskell [15]. Such conventions, however, increase the programmer’s burden. Moreover, it’s unclear how to retrofit such a segregated type system into mainstream languages.

Finally, in addition to its cleaner semantics, a general requirement for strong atomicity semantics promises more consistent behavior between hardware TM (HTM) implementations (which are already strong) and STM implementations (which can be modified to be strong).

2.2 Implementing strong atomicity

Recent work has shown that an STM can implement strong atomicity with no adverse effects on scalability [30]. The primary challenge with strong atomicity is overhead it imposes in STM.

Implementing strong atomicity requires tracking non-transactional memory accesses to detect conflicts with transactional code. By relying on existing cache coherency mechanisms, many HTM systems implement strong atomicity naturally. In contrast, STM systems must instrument non-transactional accesses to implement strong atomicity, incurring a significant overhead even when a program makes minimal or no use of transactions. In native code, providing strong atomicity in an STM may be impractical as it requires recompiling the whole application, including pre-existing libraries. In managed code, however, virtual machines commonly add extra checks to memory accesses to support type safety (e.g., null, type, or bounds checks) or garbage collection (e.g., read or write barriers).

JIT compiler optimizations and runtime techniques [30] can reduce strong atomicity overheads, but their effectiveness has been mixed. For certain workloads, strong atomicity slows down execution by a factor of 5 even with these optimizations.

Static whole-program analysis has proven most effective in reducing strong atomicity overheads but such analysis is not practical in dynamic environments. In particular, the not-accessed-in-transaction analysis (NAIT) [30, 20] statically analyzes the whole program to determine which memory locations accessed outside of a transaction are also never accessed inside transactions, allowing the compiler to skip strong atomicity barriers for those locations. Static whole-program analyses, however, are not practical in a production setting where modular distribution of code and dynamic class loading are the norm. As described, NAIT cannot be performed in a production Java setting.

3. Dynamic NAIT analysis

In this section, we present a dynamic not-accessed-in-transaction (D-NAIT) analysis and optimization for STM. The fundamental observation behind NAIT [30, 20] is that a memory location that is not accessed inside a transaction requires no barriers to enforce strong atomicity. More precisely, a NAIT analysis categorizes memory locations as follows:

- The *TxNone* category contains memory locations that are not accessed inside a transaction. Non-transactional accesses to these locations do not require any barriers.
- The *TxRead* category contains locations that are only read inside a transaction. Only writes to these locations require barriers in non-transactional code (to avoid non-repeatable reads [30] inside transactions).
- The *TxAll* category contains locations that are written and possibly read inside a transaction. These locations require both read and write barriers in non-transactional code.

The NAIT analysis described in [30, 20] is a static, whole-program analysis where the NAIT property is summarized by the containing object type (and, for non-arrays,

Type descriptor	State		Type descriptor	State
F.x	TxNone	→	F.x	TxRead
...

(a) After foo compiled (b) After bar compiled

Figure 4. Simplified D-NAIT state for Figure 1

the particular field). Once the entire program has been properly analyzed, the compiler can eliminate strong barriers accordingly.

In contrast to whole-program NAIT analysis, D-NAIT is dynamic and does not require whole-program analysis. D-NAIT builds its not-accessed-in-transaction information optimistically and incrementally as the JIT compiles the program at runtime. In particular, it exploits the fact that our JIT (described in [2]) lazily compiles two different versions of a method depending on whether it was invoked inside or outside a transaction.

Initially, the JIT assumes that all memory locations are never accessed transactionally. It maintains a global state table where memory locations (summarized by type descriptors) are initially set to *TxNone*. As the JIT compiles a new method, it takes the following steps:

1. The JIT analyzes each method (via a linear scan), inspects each transactional load and store, and, if necessary, updates the global state table based on the corresponding type/field of the memory address operand. Note, there is no need to analyze thread local data. Only accesses to shared data on the heap (instance fields or array elements) or to static data (class fields) need to be analyzed. Figure 4 illustrates the global state for the example in Figure 1.
2. The JIT inspects each non-transactional load and store in the method. It generates non-transactional barriers based on the current global state of the analysis, as shown in Table 1. Where barriers appear unnecessary, it speculatively generates a lightweight *phantom barrier* that has minimal runtime cost.
3. Finally, before emitting machine code and executing, the JIT determines if any previously compiled phantom barriers are now invalidated by the new global state and patches them in a thread-safe manner to execute a standard strong barrier sequence instead.

Once these steps are done, the JIT may safely emit and execute the new method. In the remainder of this section, we describe this process in more detail, and we discuss how to modify it to leverage a profile-guided recompilation infrastructure.

3.1 D-NAIT analysis

D-NAIT analysis incrementally computes not-accessed-in-transaction state and barrier patching requirements. It uses

Category	Non-transactional Read	Non-transactional Write
TxNone	Phantom barrier	Phantom barrier
TxRead	Phantom barrier	Strong barrier
TxAll	Strong barrier	Strong barrier

Table 1. D-NAIT categories

```

patchList = EmptySet;
for all txn accesses A {
  if A accesses thread-local object
    continue;
  [state, phantomReads, phantomWrites] =
    stateTable.getInfo(A.typeDescriptor);
  if (A is load) {
    if (state == TxNone) {
      state = TxRead;
      patchList += phantomWrites;
      phantomWrites = EmptySet;
    }
  } else { // A is store
    if (state == TxNone) {
      patchList += phantomWrites;
      phantomWrites = EmptySet;
    }
    if (state != TxAll) {
      patchList += phantomReads;
      phantomReads = EmptySet;
    }
    state = TxAll;
    phantomWrites = phantomReads = EmptySet;
  }
  stateTable.setInfo(A.typeDescriptor, state,
    phantomReads, phantomWrites);
}

```

Figure 5. Pseudo-code for basic D-NAIT algorithm

containing type and field information to summarize information about individual memory accesses¹. Effectively, it relies on type-based aliasing to group memory accesses into disjoint alias classes, such that accesses in different classes are guaranteed to refer to different memory locations. The JIT maintains a global barrier state table with three entries for each type descriptor (i.e., alias class) — its NAIT state (*TxNone*, *TxRead* or *TxAll*), the list of read phantom barriers assuming that state, and the list of write phantom barriers assuming that state.

Each time the JIT compiles a method, it performs a linear scan over the method and updates the barrier state table when it encounters a transactional read or write operation. If any state changes occur, the JIT also computes a list of corresponding phantom barriers to patch (as described in the next

¹For instance field accesses, we always use the most general containing type that defines the corresponding field. Similarly, for element accesses to an array of references, we always use `Object[]`.

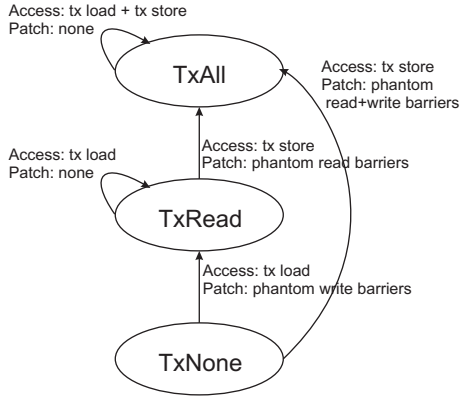


Figure 6. State transition diagram

two subsections). Figure 5 shows the D-NAIT analysis algorithm. Figure 6 shows the corresponding transition diagram for the NAIT states.

The analysis optimistically assumes that a memory location is not accessed inside a transaction until it encounters a potentially conflicting transactional access, so each type descriptor T initially starts in $TxNone$ state. A load operation transitions T to the $TxRead$ state unless it is in the $TxAll$ state. A store operation transitions T to the $TxAll$ state. This pass skips over transactional accesses where the JIT has already locally proven that an STM barrier is unnecessary (e.g., for transaction-local objects [2]); by definition, such accesses cannot conflict with other threads.

When the NAIT state of a type descriptor T changes during a method compilation, the JIT must patch any existing phantom barriers corresponding to T in order to ensure correctness. Transitioning the state of T from $TxNone$ to $TxRead$ requires patching phantom write barriers associated with T . Transitioning the state from $TxRead$ to $TxAll$ requires patching phantom read barriers associated with T . Finally, transitioning the state from $TxNone$ to $TxAll$ requires patching of both write and read barriers.

Consider the example code in Figure 7. In this code, there are four different type descriptors: three for each field of A and one for the elements of $\text{int}[]$. Table 2 illustrates the global state as each method is compiled. After only the constructor $A()$ and the method $m1$ have been compiled, the JIT has encountered no transactions, and the computed state entry for each type descriptor is $TxNone$. The JIT has, however, encountered non-transactional accesses and optimistically suppressed strong atomicity barriers. The global state table records the corresponding phantom read and phantom write barriers generated for the given type descriptor. Once $m2$ is compiled, the JIT will process a transactional write to $\text{int}[]$ and set the state of the type descriptor to $TxAll$. All phantom reads ($S3$, $S5$) and writes ($S4$, $S6$) corresponding to $\text{int}[]$ are patched and removed from the table. Similarly, the JIT will record a transactional read of the field $A.x$ in $m2$. In this case, the state is set to $TxRead$, and only phan-

```

class A {
    int[] x, y, z;

    A() {
        S0: x = new int[N];
        S1: y = new int[N];
        S2: z = new int[N];
    }

    void m1() {
        ...
        S3: ... = y[i];
        S4: z[i] = ...;
        S5: ... = x[i];
        S6: y[i] = ...;
    }

    void m2(int[] tmp) {
        atomic {
            S7: ... = tmp[i];
            S8: x[i] = ...;
        }
    }

    int[] m3() {
        S9: return y;
    }
}
  
```

Figure 7. D-NAIT example. Assume methods are initially invoked and compiled in order: $A()$, $m1$, $m2$, $m3$.

tom writes ($S0$) are patched and removed. After $m3$ (which has no effect) is compiled, several patched barriers remain.

3.1.1 Context-sensitive D-NAIT analysis

The basic D-NAIT approach works fine as long as common types and fields are not accessed both inside and outside of transactions. In some cases, however, more precision can be acquired by considering additional context information. Consider again the example in Figure 7. In this case, only the array x is accessed transactionally, but, as a result, all $\text{int}[]$ accesses are presumed to require barriers.

However, if the JIT can establish that reference field x points to a unique array, it can confine the effects of the transactional write to $x[i]$ and still speculatively use phantom barriers to access other $\text{int}[]$ arrays. Uniqueness of reference fields, like NAIT, can be tracked by the JIT in an optimistic fashion. In our implementation, we use a simple dynamic approach. As with NAIT, we track uniqueness by type descriptor. When a class is first loaded, the JIT optimistically assumes that its reference fields are all initially marked `unique`. On each method compilation, a uniqueness analysis pass updates the uniqueness information based on the following conservative rule: A reference field F is unique if it points to a freshly allocated object, and the object reference is stored only into F and does not escape. If the field F

	(NAIT State, PhantomReads, PhantomWrites)	
	After A() & m1()	After m2() & m3()
A.x	(TxNone, {S5}, {S0})	(TxRead, {S5}, ∅)
A.y	(TxNone, {S3,S6}, {S1})	(TxNone, {S3,S6,S9}, {S1})
A.z	(TxNone, {S4}, {S2})	(TxNone, {S4}, {S2})
int[]	(TxNone, {S3,S5}, {S4,S6})	(TxAll, ∅, ∅)

Table 2. D-NAIT analysis results for Figure 7

is used in any manner inconsistent with this rule, it is marked as *aliased*. A reference escapes if it is returned from a method, passed as a parameter in a method invocation, stored into a field other than F, or thrown as an exception. These rules provide a conservative approximation of unique fields.

To exploit context information, we make two modifications to the global state table, illustrated in Table 3. First, we extend each type descriptor with an additional level of context. For example, `A.x::int[]` represents the `int[]` only reachable from `A.x`. When the context is unknown, ambiguous, or non-unique, we use the notation `*::int[]` to represent the generic aliased context. In our scheme, the memory locations represented by `A.x::int[]` and `*::int[]` are disjoint. If we cannot establish that `A.x` points to a unique array, the former descriptor must be merged with and replaced by the latter descriptor. Second, we introduce a forwarding pointer to facilitate merging. In Table 3, after `m3` is compiled, the entry for `A.y::int[]` is a forwarding pointer to the aliased context. For type descriptors with a precise context, the presence of a forwarding pointer indicates that the context is aliased. The lack of a forwarding pointer indicates that the context is unique.

Table 3 illustrates the effect of context for the example in Figure 7. Here, we use additional context to maintain distinct entries for `A.x`, `A.y`, and `A.z`. As the constructor and `m1` are compiled, phantom barriers are generated. Because all initialization of the fields are to fresh memory and the memory never escapes, each extended descriptor is regarded as unique and has its own entry. When `m2` is compiled, it observes a transactional write to an `int[]` array. In this case, however, the write is in a unique context, and only `A.x::int[]` is updated to `TxAll` (triggering a patch on `S5`). There is also a read to an `int[]` of unknown context, and so `*::int[]` is set to `TxRead`. This, however, triggers no patching, as `A.y` and `A.z` still point to unique data that can not alias with the argument in `m2`. On the other hand, once `m3` is compiled, `A.y` escapes and the entry for `A.y::int[]` must be merged into `*::int[]`. This, in turn, triggers a patch on the write in `S6`. Overall, the use of context has allowed the system to maintain two extra phantom barriers: `S3` and `S4`.

Figure 8 illustrates the algorithm for managing additional context in D-NAIT. The JIT employs a simple pass based on the rules described above to determine when an extended type descriptor transitions from unique to aliased. When a

```
mergeWithAliased(TypeDesc [C1::C2]) {
    patchList = EmptySet;
    [state,phantomReads,phantomWrites] =
        stateTable.getInfo([C1::C2].typeDescriptor);
    [state*,phantomReads*,phantomWrites*] =
        stateTable.getInfo([*::C2].typeDescriptor);
    // Using TxnNone < TxnRead < TxnAll
    state* = max(state, state*);
    phantomReads* += phantomReads;
    phantomWrites* += phantomWrites;
    if (state* != TxnNone) {
        patchList += phantomWrites*;
        phantomWrites* = EmptySet;
    }
    if (state* == TxnAll) {
        patchList += phantomReads*;
        phantomReads* = EmptySet;
    }
    stateTable.setInfo([*,C2], state*,
        phantomReads*,phantomWrites*);
    stateTable.setForwardingPointer([C1,C2],[*,C2]);
    return patchList;
}
```

Figure 8. Pseudo-code for merging barrier info

transition occurs, it must merge the extended type descriptor `C::T` into the corresponding aliased descriptor `*::T` and install a forwarding pointer. The new state is the more restrictive of the two states prior to merging. The phantom read and write lists are merged, and, depending on the new state, then patched and removed.

3.2 Phantom barrier generation

As the JIT compiles each method, it speculatively removes strong atomicity barriers based on the current global D-NAIT state. In their place, it generates lightweight *phantom barriers* that introduce minimal runtime overhead but can be later converted to standard barriers. These phantom barriers are similar to speculatively devirtualized calls in [22]. During normal execution, the original load or store is executed without a strong atomicity barrier. If the state of the field changes and the speculation is invalidated, the phantom barrier is patched by the compiler to convert it to a standard strong atomicity barrier.

	(NAIT State, PhantomReads, PhantomWrites)			
	After A() & m1()	After m2()	After m3()	Final
*::A.x	(TxNone, {S5}, {S0})	(TxRead, {S5}, ∅)	...	(TxRead, {S5}, ∅)
*::A.y	(TxNone, {S3,S6}, {S1})	...	(TxNone, {S3,S6,S9}, {S1})	(TxNone, {S3,S6,S9}, {S1})
*::A.z	(TxNone, {S4}, {S2})	(TxNone, {S4}, {S2})
A.x::int[]	(TxNone, {S5}, ∅)	(TxAll, ∅, ∅)	...	(TxAll, ∅, ∅)
A.y::int[]	(TxNone, {S3}, {S6})	...	→ *::int[]	→ *::int[]
A.z::int[]	(TxNone, ∅, {S4})	(TxNone, ∅, {S4})
*::int[]	(TxNone, ∅, ∅)	(TxRead, ∅, ∅)	(TxRead, {S3}, ∅)	(TxRead, {S3}, ∅)

Table 3. Context-sensitive D-NAIT analysis results for Figure 7

Figure 9 shows the pseudo-code for a phantom write barrier used to protect an add operation that updates the heap. A phantom barrier consists of two parts:

1. The original load or store instruction. If additional space is necessary to accommodate a patch, we can take one of two strategies. We can insert nops after the access instruction, or we can duplicate instructions following the original access if possible (i.e., they do not also have phantom barriers).
2. A barrier code block, unique for each load or store, that performs the standard strong atomicity barrier followed by any duplicated instructions. In Figure 9, <next statement1> is duplicated in the barrier block. Note, the barrier block is only executed if the method is patched. It may be generated lazily by the JIT upon patching.

In the intermediate representation, the compiler maintains an additional phantom conditional jump in front of the original instruction. This phantom jump, not emitted in the final assembly code, is necessary to ensure that the normally unreachable barrier code is connected to the control flow graph. The barrier code itself is marked as infrequently executed (cold) code.

In general, the compiler is able to leverage existing optimizations to efficiently generate phantom barriers. For example, profile-guided code layout places all phantom barriers together at the end of a method’s generated code along with other blocks that are deemed cold. To minimize the effect of phantom barriers on register allocation, we also explicitly generate spill code in the cold barrier block as shown in Figure 9. This prevents the cold barrier block from interfering with live register ranges on the hot path.

Before the compiler emits a method with speculative phantom barriers, it records all information necessary to test and, if necessary, invalidate those barriers. The following information is maintained in the global state table:

1. For each type descriptor, the compiler retains a list of instruction pointers to all corresponding loads and stores where phantom barriers have been generated.

```

...
// Original           // Patch
add [base+offset], 1 <= jmp barrier_block
<next statement1>

next_block:
<next statement2>
...
ret

barrier_block:
<save live registers>
<stm lock base>
add [base+offset], 1
<stm unlock base>
<restore live registers>
<next statement1>
jmp next_block

```

Figure 9. Phantom write barrier for an add operation that stores to the heap in IA-32 pseudo-code. On patching, the add is overwritten with a jump to the barrier block. If the instruction to patch is smaller than the jump, the instruction following the memory access is duplicated in both the original block and the barrier block.

2. For each load or store instruction (identified by its instruction pointer), the compiler records the displacement to its corresponding barrier code block. (This is elided in Tables 2 and 3 for conciseness.)

3.3 Phantom barrier invalidation

If JIT analysis on a method has triggered an access state transition on a type descriptor, it may invalidate previously generated phantom barriers. Before emitting code that invalidates earlier assumptions, the compiler must safely convert incorrect phantom barriers to regular strong barriers. It does this by taking following steps:

1. The compiler accesses the above tables to determine the set of memory accesses to patch and, for each access, a corresponding jump instruction to insert in its place. If

this set is empty, no further work is necessary, and the compiler jumps to step 6.

2. The compiler invokes a stop-the-world-mechanism and signals all currently executing threads to pause at the next safe point.
3. The compiler waits until all other threads have reached a safe point.
4. The compiler patches each instruction, installing a jump to the corresponding barrier block. Due to variable instruction length on IA-32, the compiler must maintain the size of each patched instruction to generate the correct NOP padding. As described earlier, the compiler reserved enough space for a jump instruction that branches to the barrier block. For our platform, it reserves 5 bytes for a 32-bit branch to the barrier code².
5. The compiler signals each paused thread to execute a serializing instruction (CPUID on our IA32 platform) and continue.
6. The compiler emits code for the newly compiled method. At this point, all previously generated code has been updated and made visible to other threads.

The safety of the above process relies on two other constraints. First, only one thread of compilation may access and update the global tables at a given time. In our system, this is enforced in a coarse manner: only one thread may perform compilation at a time. Second, safe points may not overlap with phantom barriers. Once all other threads are at safe points, the compiler is assured that no thread is concurrently executing a phantom barrier as it is being patched.

The above algorithm respects the general guidelines for thread-safe cross-modifying code for the IA-32 architecture [21]. In particular, the CPUID (or other serializing) instruction is necessary to flush the instruction cache. To stop-the-world, we simply use existing VM and JIT mechanisms to support stop-the-world garbage collection (including safe points).

Note, our invalidation process does not need to abort ongoing transactions. Instead, it is sufficient to pause transactions at safe points (as any other thread), and to let them continue from that point once patching is complete. The above mechanisms ensure that conflicts between transactional and non-transactional accesses occur only after patching is completed and visible to all threads.

Table 3 shows the sequence of patches applied when executing our running example (Figure 7) using context information. For simplicity, we focus on `int []` accesses. In `m1`, the JIT encounters no transactions and installs phantom barriers for `int []` accesses in `S3`, `S4`, `S5`, and `S6`. In `m2`, the JIT encounters a transactional write and patches `S5`. In `m3`, the

JIT encounters a uniqueness state change and patches `S6`. `S3` and `S4` remain unpatched. Phantom barriers to the fields of `A` are maintained and patched similarly.

3.4 Exploiting profile-guided recompilation

The cost of misspeculation in our system is potentially quite high. Patched jumps to out-of-line barriers are not as efficient as normally generated strong barriers, suffering from additional instructions, poor code layout, and suboptimal register allocation. Stopping the world temporarily halts all progress. To explore reducing misspeculation, we leverage infrastructure for dynamic recompilation to employ phantom barriers more selectively.

Our runtime environment performs dynamic profile-guided optimization and has two compilers built-in (`O1` and `O2`). The `O1` compiler only performs a minimal set of optimizations and instruments the generated code to collect edge profile information. Hot methods are selected using the profile information and recompiled with the aggressively optimizing `O2` compiler.

We use this framework to improve optimization of strong atomicity barriers. We experimented with two recompilation strategies. Both exploit the fact that recompilation provides our analyses time to converge and allows the compiler to make better decisions for hot methods.

1. Phantom barriers in `O1+O2` code: We generate phantom barriers during initial `O1` compilation and speculate that there will be no patching. During `O2` recompilation of hot methods, we replace misspeculated phantom barriers with the standard inlined barriers.
2. Phantom barriers only in `O2` code: We generate standard strong barriers during `O1` compilation. During `O2` recompilation, we insert speculative phantom barriers more selectively in hot methods. Overall, we expect to see less patching activity with this approach.

We study the effects of both approaches in the next section.

4. Performance

We evaluate the effectiveness of our dynamic optimizations using both transactional and non-transactional workloads. Our underlying STM, described in [2], uses an eager-versioning, in-place update scheme (similar to that also described in [16]). On top of this, our underlying system also supports strong atomicity as described in [30].

For our workloads, we focus on those evaluated in [30]. For the standard, non-transactional SPEC JVM98, earlier dynamic techniques often left substantial strong atomicity overhead. Only static whole-program optimization was generally effective at removing this overhead. Similarly, for the transactional TSP (a traveling salesperson problem solver), earlier dynamic optimizations alone were ineffective, leaving a roughly 3x overhead. In contrast, the transactional ver-

²A 2-byte short jump has a range of only +127/-128. Our code layout typically puts barrier blocks at the end of methods which is usually too far away.

sions of SpecJBB and OO7 in [30] already show little overhead due to strong atomicity. As expected, our optimizations do not have a noticeable effect on these workloads, and we do not discuss them further. Instead, we study two different versions of SpecJBB that exhibit significant strong atomicity overhead: the unmodified synchronized version and a partially transactionalized version that spends considerable time in non-transactional code.

In our experiments, we use object-level conflict detection granularity in our STM, and we enable all JIT compiler optimizations described in [2] and [30], as well as dynamic escape analysis [30]. We do not use any static whole-program analyses [30] as our focus is on a production Java setting. In our experiments, we measure the performance of the baseline strong atomicity mode without our new optimizations (labeled Strong (Base)), strong atomicity using D-NAIT (Strong (D-NAIT)), strong atomicity with context-sensitive D-NAIT (Strong (+Context)), and finally strong atomicity with context-sensitive D-NAIT on hot methods only and full strong atomicity barriers on cold methods (Strong (+Hot Method)).

We conduct all our experiments on an Intel Clovertown system with two 2.66 GHz quad-core processors for a total of eight hardware threads and 3.25 GB of RAM running Microsoft Windows XP Professional with Service Pack 2.

4.1 SPEC JVM98

We first measure the effect of D-NAIT on the unmodified SPEC JVM98 benchmark suite [31]. These benchmarks are non-transactional and, with the exception of mtrt, single-threaded.

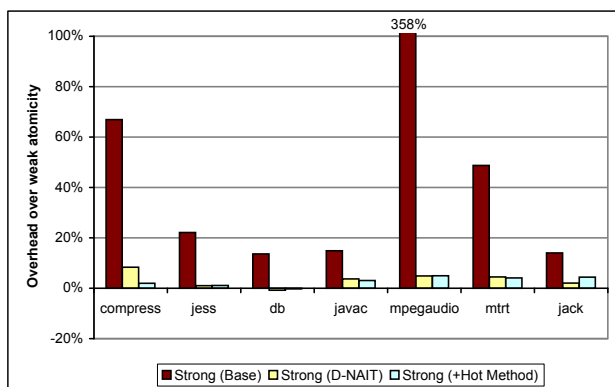


Figure 10. Overhead of strong atomicity on SPEC JVM98

Ideally, we would like to see no STM overhead for non-transactional workloads. As Figure 10 shows, the baseline strong atomicity mode incurs significant overheads — ranging from a minimum of 14% on db to as much as 358% on mpegaudio — because it inserts non-transactional barriers for reads and writes. D-NAIT reduces these overheads considerably. When phantom barriers are only used in hot methods, the overheads range from unnoticeable on db to about

5% on mpegaudio. Because the JIT never encounters any transactions, it always speculatively inserts phantom barriers in lieu of standard barriers and never patches the phantom barriers. This figure does not show the effects of adding context information to D-NAIT — since the JIT does not insert any strong atomicity barriers context information has no effect on the results. Using phantom barriers on hot methods has only a negligible effect with one exception. For compress, overhead is lowered from 8% to 2%. Compress has unusually few very hot methods, and, in this case, it appears beneficial to only generate phantom barriers in those methods.

All remaining overhead from D-NAIT in Figure 10 is solely due to the cost of inactive phantom barriers; these costs include execution of extra nop instructions, cache and TLB effects of code-size expansion from the barrier blocks, and the effects of modeling phantom barriers in the JIT’s IR on optimization phases. Our current implementation is also conservative in some respects: we do not duplicate instructions as described in Section 3.2 to reduce nops on the hot path, and we preallocate additional space in a method’s stack frame to support saving and restoring registers in barrier blocks. In general, we believe that with further tuning — including generating the barrier blocks on demand and removing the branches to the barrier blocks in the IR — we can bring these costs down even closer to zero for workloads such as SPEC JVM98 that make no use of transactions.

4.2 Traveling Salesperson Problem

TSP is a multithreaded Java implementation of the traveling salesperson problem used in earlier research on Java concurrency [36, 19]. Each thread evaluates different portions of the search space in a largely independent manner. Small critical sections, implemented as transactions in our version, are used to share intermediate work and track the currently optimal solution. From the perspective of strong atomicity, TSP is particularly interesting as a number of data structures are accessed both transactionally and non-transactionally as data computed by one thread is shared with others if it represents a potentially optimal solution. Moreover, it contains a benign data race where the `MinTourLen` field (shown in Table 2), representing the current best, may be read non-transactionally as other threads transactionally update it. The correctness of the program relies on this field decreasing monotonically — a property that weakly atomic STMs may violate. Thus this program may theoretically execute incorrectly under weak atomicity.

Figure 11 shows the results for TSP. On a single thread, D-NAIT reduces the overhead of strong atomicity relative to weak atomicity from roughly 2.9x to 11% when using context information (and phantom barriers on all methods). Without context information, the overhead is 47%, and so the additional precision is very effective on this workload. All versions scale well up to 8 threads. At 8 threads, D-NAIT is within 10% of the weak atomicity baseline. The

	Standard Strong Barriers	Correctly Speculated Phantom Barriers	Mis-speculated Phantom Barriers	Stop-The-World Invocations
Baseline Strong	851	0	0	0
D-NAIT	62	741	48	4
+Context	36	762	53	4
+Hot Method	760	91	0	0
+Context & Hot Method	746	105	0	0

Table 4. Generated Barrier Counts for TSP

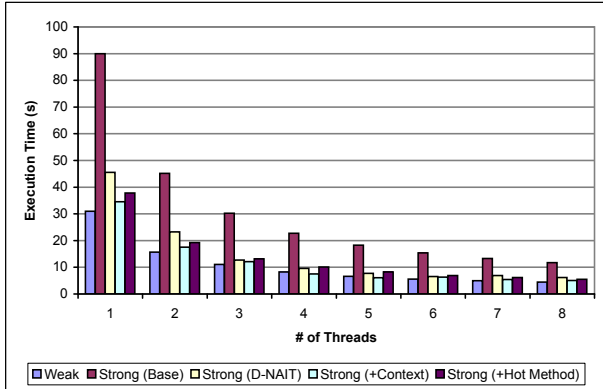


Figure 11. TSP

overhead of D-NAIT is due to real contention (mostly on the field `MinTourLen`) and due to a few non-transactional write barriers on array stores that context-sensitive D-NAIT cannot remove.

When phantom barriers are only applied to hot methods, D-NAIT is slightly less effective. Table 4 shows the static barrier counts for TSP: Context information increases the number of correctly speculated phantom barriers and reduces the number of standard strong barriers. Phantom barriers for hot methods reduce the amount of mis-speculation and stop-the-world invocations. There is a trade-off between less mis-speculation and an increased number of standard strong barriers in O1-compiled code. For TSP, it appears beneficial to insert these barriers early. Even at 8 threads, the cost of the stop-the-world invocations does not negatively impact execution time.

4.3 SpecJBB2000

Finally, we investigate D-NAIT on two different variations of the SpecJBB2000 workload [32]. In this multithreaded Java server benchmark, each thread operates on an independent warehouse data set for a fixed period of time. Performance is measured in terms of business transactions per second.

Figure 12 shows the results for the standard, synchronized version of SpecJBB2000. This version has no transactions and all its critical sections are protected by locks. The baseline strong atomicity decreases throughput consid-

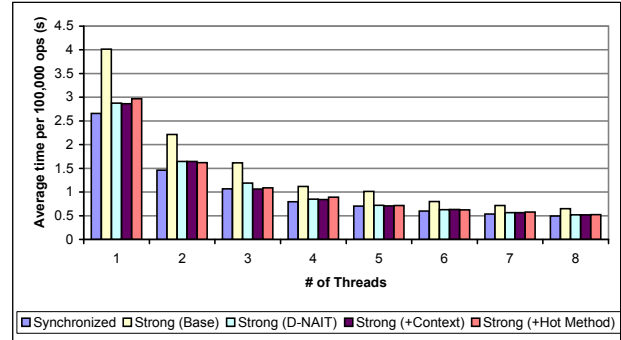


Figure 12. Standard, synchronized SpecJBB2000

erably compared to the plain synchronized version. This is unsurprising as all memory operations are non-transactional and the JIT thus instruments them with strong atomicity barriers. With D-NAIT, however, we have perfect speculation, and we see throughput within a few percent of the plain synchronized version. The throughput of D-NAIT is within 7% of the synchronized version whereas standard strong atomicity is up to 34% slower than synchronized version. Adding context and focusing D-NAIT on hot methods only has little effect on this workload.

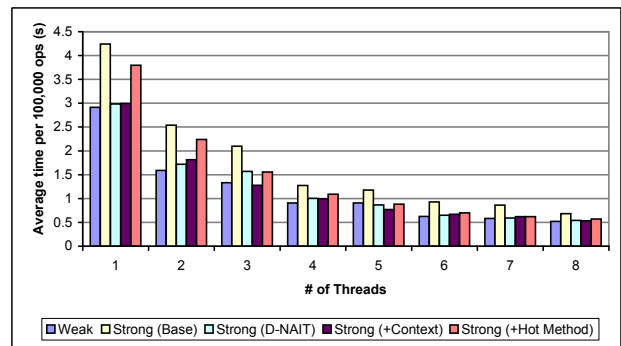


Figure 13. Synchronized SpecJBB2000 with Transactional B-Tree

The results for the second version of SpecJBB, shown in Figure 13, evaluate a partially transactional version of SpecJBB. This version retains all of the original synchronization with the exception of the core B-Tree data structure

	Standard Strong Barriers	Correctly Speculated Phantom Barriers	Mis-speculated Phantom Barriers	Stop-The-World Invocations
Baseline Strong	15253	0	0	0
D-NAIT	1860	13165	228	11
+Context	1739	13253	256	15
+Hot Method	9765	5481	0	0
+Context & Hot Method	9721	5532	0	0

Table 5. Generated Barrier Counts for Synchronized JBB with Transactional B-Tree

that is now fully transactional. Our intent here is to simulate realistic settings where TM is introduced piecemeal into applications rather than ubiquitously at once. In contrast to the standard version (where the B-Tree is not fully synchronized), the transactional B-Tree in this version is thread-safe. Accordingly, we see a slight overhead in weak atomicity mode over the synchronized version in Figure 12. On average the base strong atomicity configuration is 30% slower than weak atomicity. With D-NAIT the throughput gets within a few percent of the weak atomicity baseline. On average D-NAIT reduces the overhead of strong atomicity to 5% of weak atomicity for the partially transactional version of SpecJBB.

Table 5 shows the barrier counts for SpecJBB. Generating phantom barriers only for hot methods (Strong +Context & Hot Method) eliminates mis-speculated phantom barriers, but does not help in terms of performance. All the patching in this benchmark already occurs during the warm-up phase preceding the measurement phase. Therefore, mis-speculation of phantom barriers is not a factor here and the increased number of standard barriers weighs more. This explains why inserting phantom barriers only in O2-compiled code (Strong +Context & Hot Method) performs slightly worse with an average overhead of 13%.

5. Related work

5.1 Incremental points-to and escape analysis

There have been a number of approaches for incremental analysis of programs. With growing importance of dynamic compilation, incremental program analysis becomes more and more important. We distinguish two categories: optimistic and pessimistic analyses. An incremental, optimistic analysis makes optimistic assumptions about unanalyzed part of the program. If optimizations are performed based on these assumptions, the system requires an invalidation and recovery mechanism should an assumption fail later. Pessimistic approaches make conservative assumptions about unknown parts of the program. Therefore they do not require handling for mis-speculation. Our work is incremental and optimistic, and we provide a code patching mechanism to recover from mis-speculation.

5.1.1 Optimistic analysis

Optimistic inter-procedural analysis has been shown useful for type analysis [28]. That approach targets the optimization of virtual method calls. The system includes an event notification to recompile methods when an optimistic assumption is invalidated. The JIT compiler tracks dependencies between alias sets and compiled methods to recompile affected methods in case an optimistic assumption is invalidated.

Object inlining is another optimization that can benefit from an optimistic analysis. Wimmer et al. [37] perform dynamic optimistic object inlining [12, 24] in a dynamic compiler. The preconditions necessary to successfully inline objects are similar to our concept of unique references and are checked by runtime guards. If an inlined object reference is overwritten, the system de-optimizes all methods that access invalidated inlined objects. The de-optimization process, however, is unspecified.

Code patching has been used to enable speculative devirtualization of virtual method calls [22] in a dynamic environment. An optimistic version of class hierarchy analysis (CHA) [11], based upon currently loaded classes, is used to find candidates for devirtualization.

This style of speculative optimization and deoptimization differs from ours in two respects. First, speculative CHA assumptions are re-verified at dynamic class loading, where as D-NAIT assumptions can be verified later at method recompilation. Second, and more subtly, deoptimization of devirtualized calls is simpler in a multithreaded setting. The compiler may concurrently patch a call site (via an atomic instruction) even as another thread is invoking that call. Loading the new class does not invalidate with concurrently executing devirtualized calls. In contrast, in our system, compiling a new method may invalidate existing phantom barriers. Converting a phantom barrier to regular one requires a stronger mechanism for correctness (e.g., forcing all threads to a safe point).

5.1.2 Pessimistic analysis

Incremental escape analyses have been proposed in the context of ahead-of-time compilation [7, 34] and also for JIT compilers [23]. In escape analysis, the compiler has correct information at each step and can optimize each method safely without support for speculative optimization. Typical

applications for escape analysis are stack allocation, scalar replacement and synchronization removal.

Our optimization is essentially also a kind of synchronization removal. We also operate in a purely dynamic setting where whole-program analysis is not an option. In contrast to existing incremental escape analyses, D-NAIT is optimistic, and we eliminate barriers speculatively. The optimistic nature of D-NAIT potentially provides more optimization opportunities over static analysis which is conservative about the unknown parts of the program.

5.2 Ownership types

The concept of unique references and restricting aliasing among references has been addressed in several ways: Ownership types [10] are static type systems that restrict aliasing and limit the visibility of object references. Confined types [33] pursue a similar goal. The confinement property is useful to specify security constraints and is statically checked at compile time. Applications of such ownership models range from correctness [5, 17] to security [9, 33] and optimizations [6]. All those approaches have in common that they are static and usually require either programmer annotation or static (whole-program) analysis for inferring aliasing properties.

Our approach borrows the idea of unique references to use context information in D-NAIT, but does not rely on any form of annotations or whole-program or modular analysis. Instead, we dynamically infer ownership properties. We optimistically assume no aliasing for freshly allocated objects until proven otherwise. The dynamic approach has the advantage that methods that are never actually executed do not contribute to the analysis result.

5.3 Data race detection

Strong atomicity barriers may be viewed as a mechanism to detect data races. There are several static [36, 27] and dynamic approaches [29, 35, 8] to detect data races. Static analyses are usually sound, but more conservative (more false positives than dynamic analyses). On the other hand, dynamic techniques add significant runtime overhead because they require expensive instrumentation.

There are several ways of reducing this runtime overhead: The Eraser algorithm [29] uses a thread ownership model to use heavy-weight barriers only on object that are actually shared. Dynamic escape analysis [30] pursues a similar idea, but it tracks object visibility instead of actual shared accesses. This way the systems makes sure that it finds potentially shared objects “early” enough for the STM system to recover in case of an actual conflict. Praun et al. [36] use static analysis eliminate instrumentation overhead caused by dynamic race checking. Using thread-local information obtained by the static analysis the compiler can only instrument access to potentially shared objects.

Our work on strong atomicity STM goes into a similar direction by optimistically eliminating barriers for object that can never conflict. However, since we work in a purely dy-

namic setting (without whole-program information) we also need a backup procedure to recover from mis-speculation in our analysis. Another difference is that we do not have complete information about thread-locality, but instead track which objects are accessed within transactions as an approximation.

6. Conclusions

Strong atomicity provides a well-defined and intuitive semantics for transactional memory but has been challenging to implement efficiently in a dynamic language environment. Past work has shown how to implement strong atomicity efficiently using static whole-program analyses, but such analyses are impractical in a dynamic language environment, and without these optimizations, the high overheads of strong atomicity make it nonviable for mainstream adoption.

In this paper, we have presented new dynamic optimizations that significantly reduce strong atomicity overheads and are suitable for dynamic language environments such as Java. Our new dynamic D-NAIT analysis, augmented with context information from an incremental alias analysis, optimistically tracks which non-transactional memory accesses can avoid strong atomicity barriers. Our new phantom barrier mechanism uses these results to generate optimized code that speculatively avoids strong atomicity barriers but can be patched to recover from mis-speculation if D-NAIT later refines its optimistic results based on new dynamically compiled code. We showed how to implement these mechanisms efficiently by leveraging existing dynamic infrastructure in a Java system, and we demonstrated that our techniques substantially reduce the overhead of strong atomicity over an efficient weakly-atomic baseline. The dynamic optimizations we have presented now make strong atomicity practical in a dynamic language environment.

References

- [1] ABADI, M., BIRRELL, A., HARRIS, T., AND ISARD, M. Semantics of transactional memory and automatic mutual exclusion. In *POPL 2008*.
- [2] ADL-TABATABAI, A.-R., LEWIS, B. T., MENON, V. S., MURPHY, B. R., SAHA, B., AND SHPEISMAN, T. Compiler and runtime support for efficient software transactional memory. In *PLDI 2006*.
- [3] BLUNDELL, C., LEWIS, E. C., AND MARTIN, M. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters* 5, 2 (Nov. 2006).
- [4] BORISOV, N., JOHNSON, R., SASTRY, N., AND WAGNER, D. Fixing races for fun and profit: how to abuse atime. In *SSYM 2005* (Berkeley, CA, USA, 2005), USENIX Association, pp. 20–20.
- [5] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 2002*.

- [6] BOYAPATI, C., SALCIANU, A., WILLIAM BEEBEE, J., AND RINARD, M. Ownership types for safe region-based memory management in real-time Java. In *PLDI 2003* (2003).
- [7] CHOI, J.-D., GUPTA, M., SERRANO, M. J., SREEDHAR, V. C., AND MIDKIFF, S. P. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.* 25, 6 (2003).
- [8] 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 *PLDI 2002*.
- [9] CLARKE, D., RICHMOND, M., AND NOBLE, J. Saving the world from bad beans: deployment-time confinement checking. In *OOPSLA 2003*.
- [10] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership types for flexible alias protection. In *OOPSLA 1998*.
- [11] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP 1995*.
- [12] DOLBY, J., AND CHIEN, A. An automatic object inlining optimization and its evaluation. In *PLDI 2000*.
- [13] GROSSMAN, D., MANSON, J., AND PUGH, W. What do high-level memory models mean for transactions? In *MSPC 2006*.
- [14] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *OOPSLA 2003*.
- [15] HARRIS, T., MARLOW, S., JONES, S. P., AND HERLIHY, M. Composable memory transactions. In *PPoPP 2005*.
- [16] HARRIS, T., PLESKO, M., SHINNAR, A., AND TARDITI, D. Optimizing memory transactions. In *PLDI 2006*.
- [17] HEINE, D. L., AND LAM, M. S. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI 2003*.
- [18] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In *ISCA 1993*.
- [19] HINDMAN, B., AND GROSSMAN, D. Atomicity via source-to-source translation. In *MSPC 2006*.
- [20] HINDMAN, B., AND GROSSMAN, D. Strong atomicity for Java without virtual-machine support. Tech. Rep. UW-CSE-06-05-01, May 2006.
- [21] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide*.
- [22] ISHIZAKI, K., KAWAHITO, M., YASUE, T., KOMATSU, H., AND NAKATANI, T. A study of devirtualization techniques for a Java just-in-time compiler. In *OOPSLA 2000*.
- [23] KOTZMANN, T., AND MÖSSENBOCK, H. Escape analysis in the context of dynamic compilation and deoptimization. In *VEE 2005*.
- [24] LHOTÁK, O., AND HENDREN, L. Run-time evaluation of opportunities for object inlining in Java. In *JGI '02: Proc. of the conf. on Java Grande*.
- [25] MENON, V., BALENSIEFER, S., SHPEISMAN, T., ADL-TABATABAI, A.-R., HUDSON, R. L., SAHA, B., AND WELC, A. Practical weak-atomicity semantics for Java STM. In *SPAA 2008*.
- [26] MOORE, K. F., AND GROSSMAN, D. High-level small-step operational semantics for transactions. In *POPL 2008*.
- [27] NAIK, M., AIKEN, A., AND WHALEY, J. Effective static race detection for Java. In *PLDI 2006*.
- [28] PECHTCHANSKI, I., AND SARKAR, V. Dynamic optimistic interprocedural analysis: a framework and an application. In *OOPSLA 2001*.
- [29] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP 1997*.
- [30] SHPEISMAN, T., MENON, V., ADL-TABATABAI, A.-R., BALENSIEFER, S., GROSSMAN, D., HUDSON, R. L., MOORE, K. F., AND SAHA, B. Enforcing isolation and ordering in STM. In *PLDI 2007*.
- [31] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC JVM98, 1998. See <http://www.spec.org/jvm98>.
- [32] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC JBB2000, 2000. See <http://www.spec.org/jbb2000>.
- [33] VITEK, J., AND BOKOWSKI, B. Confined types. In *OOPSLA 1999*.
- [34] VIVIEN, F., AND RINARD, M. Incrementalized pointer and escape analysis. In *PLDI 2001* (2001).
- [35] VON PRAUN, C., AND GROSS, T. R. Object race detection. In *OOPSLA 2001*.
- [36] VON PRAUN, C., AND GROSS, T. R. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI 2003*.
- [37] WIMMER, C., AND MÖSSENBOCK, H. Automatic feedback-directed object inlining in the Java Hotspot virtual machine. In *VEE 2007*.