Early Experience on Transactional Execution of Java™ Programs Using Intel® Transactional Synchronization Extensions

Richard M. Yoo[†] Sandhya Viswanathan[‡] Vivek R. Deshpande[‡]
Christopher J. Hughes[†] Shirish Aundhe[‡]

[†]Intel Labs [‡]Software and Services Group
Intel Corporation
{richard.m.yoo, sandhya.viswanathan, vivek.r.deshpande, christopher.j.hughes, shirish.aundhe}@intel.com

Abstract

This paper applies Intel® Transactional Synchronization Extensions (Intel® TSX) to elide Java™ monitor locks. By utilizing the transactional execution capability inside a Java Virtual Machine (JVM), applications running on top of the JVM transparently benefit without source or bytecode modifications.

While others have previously examined transactional execution of Java programs, we find that the absence of a guarantee on eventual transactional commit requires a software fallback, and present a novel design where the fallback code harmoniously coexists with the JVM locking mechanism. We also find that if not used judiciously, a real implementation of hardware transactional memory can significantly degrade application performance. Therefore, we propose an adaptive algorithm that selectively deoptimizes transactional lock elision where necessary.

Evaluation of the Intel TSX-enabled JVM with various Java workloads shows promising results. For workloads that utilize a significant number of coarse-grained monitors, transactional lock elision provides a substantial performance improvement: up to 1.18x on a standard Java benchmark suite. In contrast, on workloads with monitor locks that are hard to elide, our adaptive deoptimization approach prevents significant performance degradation from lock elision overheads.

1. Introduction

The landscape of computing is changing. Due to limits in technology scaling, designs that exploit instruction-level parallelism to improve single-thread performance now provide diminishing returns. As a result, computer architects now rely on thread-level parallelism to obtain sustainable performance improvement [24]. More cores are being added to the same die, and both academia [14] and industry experts [15] predict hundreds, if not thousands, of cores on a single chip.

The task to efficiently program and utilize these cores, however, is left to the software programmer. To aid in the task, some modern programming languages incorporate thread-level parallelism as a key language feature [5, 7, 25]. In addition to the native support for threading, these languages also provide synchronization constructs to orchestrate accesses to shared data.

Java™ is one of these languages. From its inception, Java has provided language-level support for threading. While it initially supported only monitors [20], later versions added the Java Concurrency Framework (`java.util.concurrent`) to provide full-fledged synchronization constructs such as locks and atomic operations, and concurrent data structures.

However, inter-thread synchronization is often a significant performance overhead for threaded applications. For example, applications that use coarse-grained monitors may see limited scalability, since the execution of lock-protected monitors is inherently serialized. Applications that use fine-grained locks, in contrast, generally provide good scalability, but see increased locking overheads, and sometimes contain subtle bugs.

To provide high-performance synchronization, computer architects have proposed various schemes for speculative parallel execution [13, 26, 28], which allow the threads to optimistically execute without synchronization, and enforce synchronization only when necessary. In particular, transactional memory [13] provides software the notion of a *transaction*, a code region that *appears* to execute atomically; underneath, the hardware speculatively executes multiple threads inside the region, and rolls back the execution only if it detects any true atomicity violations.

Such decoupling allows programmers to write simple, coarsely-synchronized codes, while the hardware enforces fine-grained synchronization. Academic designs have shown significant performance potential [1, 11, 23], and more recently, commercial designs have become available [17, 31]. Starting with 4th Generation Intel® Core™ processors, Intel also provides a transactional memory implementation through Intel® Transactional Synchronization Extensions (Intel® TSX) [16].

Nevertheless, language-level adoption of transactional memory has been slow, since the introduction of a transactional construct may incur other pragmatic issues [34]. Hence, while a few mission-specific languages do support transactional memory [12], mainstream languages, e.g., C/C++ and Java, have yet to adopt transactional memory at the language level [29].

We observe that Java's use of an intermediate instruction set, combined with language-level support for synchronization constructs, provides an opportunity to utilize transactional memory without introducing new language constructs. A Java executable is a series of *bytecodes*, where a bytecode is an abstract machine language that the language runtime, a *Java Virtual Machine (JVM)*, dynamically maps to the host machine's instruction set. By modifying the JVM to re-map synchronization-related bytecodes to transactional memory operations, a Java application can utilize transactional memory. In addition, since JVMs are also used to execute programs written in other languages (e.g., Scala, Clojure, etc.), the benefit is not limited to Java.

This paper applies Intel TSX to a JVM and evaluates its performance. Specifically, we use Intel TSX to *elide Java monitor locks*. Elision is performed inside the JVM, and Java applications transparently utilize Intel TSX without any source or bytecode changes.

During the process, we identify two key challenges that need addressing to accelerate a JVM with a real-world transactional memory implementation. (1) As with other commercialized transactional memory systems [17, 31], Intel TSX does not guarantee that a transactional execution will eventually succeed; thus, we need a *software fallback to guarantee forward progress*, and a policy on when to employ that fallback. At the same time, this fallback mechanism should *correctly interact with the existing JVM monitor lock implementation*. (2) On a real transactional memory implementation, performance degradation is a distinct possibility: e.g., in monitor-protected code where transactional execution almost always fails. Thus, we need a mechanism to detect where transactional execution is hurting performance, and *selectively disable speculation* for that code.

None of the prior work that evaluates hardware transactional memory on Java [4, 6, 8, 17] addresses nor describes solutions to these challenges. Those efforts were either on simulated and/or non-commercialized transactional memory systems [4, 8]—which did not face these real-world challenges—or were on proprietary JVMs whose modifications were not disclosed [6, 17].

In this paper, we provide a detailed description of how a production-quality JVM should be modified to utilize Intel TSX—a widely available transactional memory implementation. However, our techniques are generic, and can be adapted to other JVMs using other hardware transactional memory systems, as well. We also quantify Intel TSX performance on a wide range of Java workloads; compared to previous work that only utilized microbenchmarks [8, 17], our workloads include regular Java benchmark suites. To the best of our knowledge, this is the first paper to evaluate the performance of applying Intel TSX to Java.

In the next section we summarize the Intel TSX programming interface and its implications. In Section 3, we provide a brief introduction to Java synchronization mechanisms. Then, in Section 4, we describe our JVM modifications in detail; we provide the performance results in Section 5. Section 6 discusses previous work, and Section 7 concludes.

2. Intel® Transactional Synchronization Extensions

Developers can use the Intel TSX instruction set interface to mark code regions for transactional execution [16]. The hardware executes these developer-specified regions speculatively, without performing explicit synchronization and serialization. If speculative execution completes successfully (i.e., the transaction is *committed*), then memory operations performed during the speculative execution appear to have occurred atomically at the time of commit, when viewed from other processors. However, if the processor cannot complete its speculative execution successfully (i.e., the transaction is *aborted*), then the processor discards all speculative updates, restores architectural state to before the region started, and resumes execution. The execution may then need to serialize through locking, to ensure forward progress. The mechanisms to track transactional states, detect data conflicts, and commit or roll back transactional execution are all implemented in hardware.

2.1 Programming Interface

Intel TSX provides two programming interfaces. With the Hardware Lock Elision (HLE) interface, a programmer marks the memory operations that acquire and release a lock variable with prefixes. The write operation performed to acquire the lock is elided, and the hardware transactionally executes the following instructions, until a matching HLE-prefixed write operation that releases the lock variable is encountered. While legacy compatible, HLE provides little degree of freedom over the fallback; when a transaction aborts, HLE automatically re-executes the region non-transactionally and without elision.

Restricted Transactional Memory (RTM), in contrast, provides a more flexible interface. A programmer specifies a transactional region through the XBEGIN and XEND instructions; when a transaction aborts, architectural state is recovered, and the execution restarts non-transactionally at the fallback address provided with the XBEGIN instruction. Inside the fallback handler, the programmer can determine whether to retry the hardware transaction or to fallback to software. In addition, a programmer can explicitly abort a transaction by executing the XABORT instruction, and the XTEST instruction can be used to query whether the processor is currently in transactional execution mode. Nested transactional regions are flattened and treated as one monolithic region. Due to its generality, in this work we utilize the RTM interface.

2.2 Implications

To accommodate implementation constraints, Intel TSX may abort a transaction for numerous architectural and microarchitectural conditions. In addition to data conflicts, examples include exceeding buffering capacity for transactional states, and executing instructions that may always abort (e.g., system calls). Therefore, one cannot assume that a given instance of a transactional region will *ever* commit. Instead, the software fallback path must ensure forward progress, and it must be able to run successfully without Intel TSX. Additionally, the transactional path and the fallback path must co-exist without incorrect interactions.

In particular, Intel TSX can be used to convert existing lock-based critical sections into transactional regions; in this capacity, it is used for *lock elision*. For such cases, the natural fallback path would be to use the prior locking mechanism. To guarantee correctness, however, the transactional path must test the lock during the execution—to ensure correct interaction with another thread that may, or already has, explicitly acquired the lock non-transactionally—and should abort if not free. In addition, the software fallback handler should establish a policy on whether to retry transactional execution (e.g., if the lock is not free), or to non-transactionally acquire the lock.

In this regard, using Intel TSX within a runtime, e.g., a JVM, is preferable, since once the correct mechanisms are implemented, all the workloads utilizing the runtime can benefit. A runtime can also utilize dynamic information to further optimize performance.

3. Java Synchronization Overview

We now briefly describe how Java monitors provide synchronization, first at the language level, then how they are implemented within a JVM.

3.1 Programming Model

Java programmers apply the `synchronized` keyword to designate code segments that should be executed in a mutually exclusive

```
public class Counter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }
}
```

Listing 1: **Shared counter with synchronized methods.**

```
public class Counter {
    private int c = 0;

    public void increment() {
        synchronized(this) {
            c++;
        }
    }
}
```

Listing 2: **Shared counter with synchronized statements.**

fashion. The keyword can be used to implement either *synchronized methods* or *synchronized statements*. Listing 1 shows an example that uses *synchronized methods* to implement a shared counter. In Java, every object has a lock associated with it—a *monitor lock*. Invocations of synchronized methods on an object are serialized by implicitly acquiring the object's monitor lock.

In contrast, *synchronized statements* are used to provide mutual exclusion on a contiguous sequence of code that may be smaller than an entire method. Listing 2 implements the same functionality as in Listing 1, using synchronized statements. Unlike synchronized methods, synchronized statements must specify the object whose monitor lock is used; this example uses the lock from the current object, `this`.

Since monitors are associated with objects, *any* synchronized piece of code, whether it be a method or statement, that uses the monitor lock on the same object cannot be simultaneously executed by different threads.

In either case, a synchronized code segment is required to have a *block structure*; i.e., all Java statements in a synchronized segment must be in the same lexical scope. When compiled to bytecodes, the beginning and the end of a synchronized code segment is denoted by a `monitorenter` and `monitorexit` bytecode, respectively.

3.2 Java Virtual Machine Implementation

`monitorenter` and `monitorexit` only specify that the bytecodes in between the two should be executed mutually-exclusive; a JVM is free to choose how to implement the exclusion. We now describe a state-of-the-art, production-quality implementation [2, 27]. Our baseline JVM uses these well-known techniques.

To optimize for the case of uncontended locks, the JVM maintains two different locks to implement a single monitor lock: a *thin lock* and a *fat lock*. The *thin lock* is for the uncontended case; it is implemented as a lightweight, user-level lock. In contrast, the *fat lock* is a fallback if the JVM detects contention; it is implemented with a heavyweight OS mutex and a condition variable.

Initially, the JVM considers a monitor lock uncontended. A thread that tries to acquire the lock performs a compare-and-swap (CAS) on the monitor lock variable, to have it point to its *lock record*. If the CAS succeeds, the thread has acquired the lock. If the CAS fails, however, it means that some other thread acquired the lock, and the lock is *inflated*. The thread then tries another CAS operation to swing the monitor lock variable to point to a fat lock, and waits on the accompanying condition variable.

When a thread tries to release the monitor lock, it first performs a CAS on the monitor lock, which should still point to its lock record. If the CAS succeeds, there was no contention, and lightweight locking remains in effect. However, if the CAS fails, this means there was contention while the lock was held, and the thread executes the code path required to release a fat lock; this includes notifying waiting threads, and releasing the lock.

```
routine monitor_enter(monitor_lock) {

    // Handle thin lock
    if acquire_thin_lock(monitor_lock) is success,
        return;

    // Lock is inflated: try to elide fat lock
    for (int i = 0; i < retry_threshold; i++) {
        // Increment total # txns tried
        inc_total_count(monitor_lock);

        XBEGIN abort_handler;

        // Check if fat lock is already acquired
        if not fat_lock_occupied(monitor_lock),
            // Proceed to execute the critical section
            return;

        else
            // Fat lock is acquired by another thread
            XABORT;

abort_handler:
        adjust_elision(monitor_lock);
    }

    // Elision failed: explicitly acquire fat lock
    acquire_fat_lock(monitor_lock);
}
```

Listing 3: **Pseudocode for Intel TSX-enabled monitor lock acquisition.**

```
routine monitor_exit(monitor_lock) {

    // Handle thin lock
    if release_thin_lock(monitor_lock) is success,
        return;

    // Lock is inflated: did we elide the fat lock?
    if not fat_lock_occupied(monitor_lock),
        // Lock was elided: commit transaction
        XEND;

    else
        // Fat lock was acquired
        release_fat_lock(monitor_lock);
}
```

Listing 4: **Pseudocode for Intel TSX-enabled monitor lock release.**

Compared to a fat lock, a thin lock acquisition takes only a single CAS operation. For uncontended monitors, a lock will never be inflated, and the synchronization overhead (while unnecessary) will be small. Some implementations can even *bias* locks towards a thread [27], to further reduce the CAS overhead. However, once a monitor lock is inflated, it remains inflated, and subsequent lock acquisitions will follow the fat lock acquisition protocol. Also notice that this dual-lock implementation of Java monitors is incompatible with the HLE interface, which assumes a lock is implemented as a single in-memory variable.

4. Enabling Transactional Execution on a Java Virtual Machine

We now describe the modifications to our JVM to enable transactional execution. While our descriptions are in the context of Intel TSX and our JVM, our techniques are not specific to these—our ideas are applicable to other JVMs using other hardware transactional memory systems.

4.1 Basic Algorithm for Monitor Lock Elision

Since Intel TSX does not guarantee that a transaction will eventually succeed (see Section 2), we use the baseline monitor lock implementation of our JVM as the fallback, and opportunistically try to elide the lock. Our microbenchmark measurements [33] show that while Intel TSX is relatively lightweight, its overheads are larger than a single CAS operation. Combined with the fact that thin locks are uncontended, this means applying Intel TSX to elide thin locks would not be beneficial. Therefore, we keep the thin lock handling unmodified, and apply Intel TSX to elide fat locks only.

Listing 3 shows the pseudocode for our modified monitor lock acquisition routine. A JVM executes this routine when encountering the `monitorenter` bytecode.

The JVM first executes the regular thin lock acquisition protocol (line 4). If it succeeds, the JVM moves on to execute the first bytecode in the critical section. If the thin lock acquisition fails, the monitor lock is inflated, and the JVM tries to elide the fat lock.

To elide the fat lock, the JVM starts a transaction by executing XBEGIN (line 12), with the address (label) of the abort handler as the argument. Once it starts transactional execution, the JVM then checks the fat lock to see if it is occupied by another thread (line 15). As discussed in Section 2, this is necessary to guarantee correct interaction with (1) a thread that has non-transactionally acquired the lock, or (2) one that would non-transactionally acquire the lock in the future, before this thread exits the critical section. If the fat lock is occupied, we explicitly abort the transaction (line 21). If not, the JVM speculates the lock elision has succeeded, and moves on to transactionally execute the critical section. If, during the execution of the critical section, another thread non-transactionally acquires the lock, the hardware will automatically abort this transaction.

When a transaction aborts, either via an XABORT instruction or a hardware abort, execution resumes at the `abort_handler` label (line 23). Once some statistics are updated (line 24, discussed later), the JVM retries transactional execution as long as the retry count is below a certain threshold, i.e., `retry_threshold`. If the JVM fails to successfully elide the fat lock within the threshold, it gives up on transactional execution, explicitly acquires the fat lock (line 28), and then executes the critical section non-transactionally.

On the other hand, Listing 4 shows the pseudocode for our modified monitor lock release routine, which the JVM executes when it encounters a `monitorexit` bytecode.

First, the JVM checks whether the critical section was entered by acquiring a thin lock, and tries to release the thin lock (line 4). If it fails, the monitor lock was inflated, and the JVM checks whether it had transactionally elided the fat lock, by checking its occupancy (line 8). If the lock is not occupied, it means the lock was transactionally elided, so the JVM commits the transaction (line 10). If the lock is occupied, on the other hand, it means the JVM had explicitly acquired the fat lock. So the JVM releases the lock using the regular fat lock release protocol (line 14).

The correctness of the else-clause on line 12 relies on a non-obvious property; it works only if it is impossible for a thread that transactionally elided the fat lock in the `monitor_enter()` routine to find the lock to be occupied by a different thread in `monitor_exit()`. When a thread transactionally elides the fat lock in `monitor_enter()`, it checks the status of the fat lock (Listing 3, line 15), adding the lock variable to the read set of its transaction. If another thread non-transactionally acquires the fat lock later, it must write to the lock variable as part of the acquisition. As mentioned above, this conflict will be detected by the hardware and will abort the transaction[1].

[1] This assumes a transactional memory system with strong atomicity guarantee, which the cache coherence-based commercial designs trivially satisfy [8, 16, 17, 31].

```
1  routine adjust_elision(monitor_lock) {
2
3      // Increment the abort count
4      abort_count = inc_abort_count(monitor_lock);
5
6      // Get total # txns tried
7      total_count = get_total_count(monitor_lock);
8
9      // Is the monitor lock contended enough?
10     if abort_count > abort_threshold:
11
12         // Bias against lock elision?
13         if abort_count / total_count > abort_ratio:
14             // Deoptimize txn lock elision
15             set_elision_state(monitor_lock, NO);
16
17     // Bias towards lock elision?
18     else if total_count > lock_threshold:
19
20         // Always perform txn lock elision
21         set_elision_state(monitor_lock, YES);
22
23     // Perform txn scheduling
24     while fat_lock_occupied(monitor_lock) delay;
25 }
```

Listing 5: **Pseudocode for the adaptive tuning logic.**

4.2 Adaptive Tuning

In addition to our basic algorithm, we also implement performance features to dynamically adapt the JVM's lock elision behavior. Specifically, for each monitor lock, we maintain the number of transaction attempts and aborts, and selectively *deoptimize* lock elision when not profitable. Listing 5 shows `adjust_elision()`.

The first criteria to determine deoptimization is whether the number of transactional aborts is significant (line 10). For monitor locks with a sufficient number of aborts, if the transactional abort ratio is greater than a user-specified threshold (line 13), we deoptimize transactional lock elision. To keep track of this decision, for each monitor lock, the JVM maintains a variable that records the *lock elision state*: `MAYBE`, `YES`, or `NO`. When the state is changed to `NO`, the JVM will re-map the code so that when the `monitorenter` bytecode is next encountered, it will execute the original, non-transactional monitor lock acquisition code.

On the other hand, the JVM may decide to *bias for* transactional lock elision. In Listing 5, for monitor locks that have few aborts but are acquired frequently (line 18), the JVM sets the lock elision state to `YES`, to bias the locks so that transactional elision is always attempted. To further optimize performance, the JVM may disable the `adjust_elision()` routine entirely.

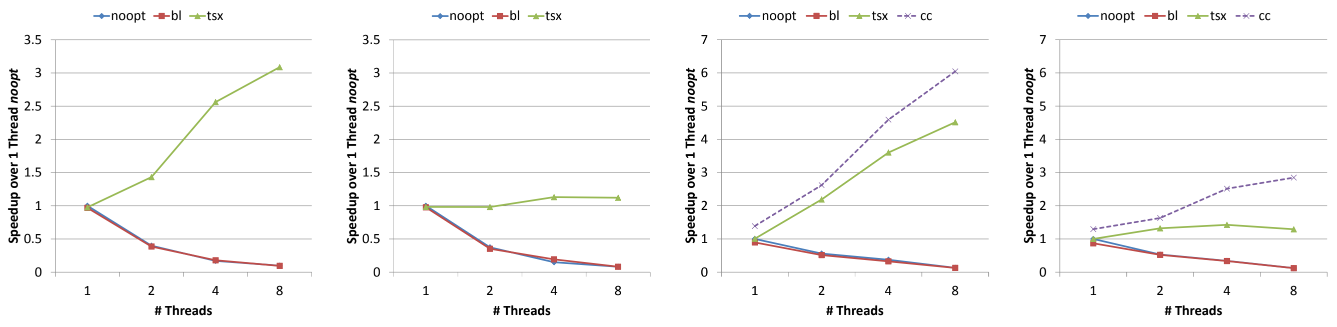
By default, a monitor lock is initialized to the `MAYBE` state—which is the same as `YES` but with the `adjust_elision()` routine enabled. Monitor locks in `NO` and `YES` states may periodically be reset to `MAYBE` to adapt to program phase changes.

In addition to deoptimization, the last part of `adjust_elision()` implements transaction scheduling (line 24) to further adapt the lock elision behavior. Here, we choose to spin-wait until the fat lock is free; other mechanisms (e.g., exponential backoff, adaptive scheduling [32], etc.) can also be implemented.

4.3 Discussion

Since Intel TSX implements closed nesting (see Section 2), this design trivially handles nested monitors. When a transaction aborts, the JVM will resume execution at the outermost abort handler. It may then re-attempt to elide the outermost monitor lock.

In cases of exceptions (`java.lang.Exception`) thrown in synchronized code, we conservatively abort. Assuming the exception recurs on every attempted transactional execution, the JVM will eventually acquire the monitor lock non-transactionally. At that point the exception will be raised again, and be handled by the



(a) `Hashtable` **(0p, 100g, 0r)** (b) `Hashtable` **(20p, 70g, 10r)** (c) `HashMap` **(0p, 100g, 0r)** (d) `HashMap` **(20p, 70g, 10r)**

Figure 1: **Microbenchmark performance results. Speedup is against** *noopt* **with single thread.** *p*, *g*, **and** *r* **denotes the percentage of** *put()*, *get()*, **and** *remove()* **operations, respectively.**

non-transactional, standard exception handling routine. We similarly abort a transaction when the execution flow leaves the scope of the JVM (e.g., via a use of the Java Native Interface (JNI)). Since Intel TSX detects conflicts at the hardware level, it is possible to transactionally execute JNI code and still maintain correctness. Nevertheless, we opted for a conservative design choice.

In addition, we do not modify condition variable routines (i.e., `wait()`, `notify()`, and `notifyAll()`). We find that condition variable use in Java is relatively rare [18]. Moreover, in the worst case, invocation of these methods will simply abort a transaction[2]: The thread will eventually acquire the monitor lock non-transactionally, and correct operation will be guaranteed through non-transactional execution. In the future, transactional execution-aware condition variables may be implemented [10].

4.4 Performance Debugging Support

To help users optimize the performance of an Intel TSX-enabled JVM, we also implement some performance debugging features. Specifically, a command-line flag tells the JVM to dump transactional statistics for each `monitorenter` bytecode at the end of execution. In addition to the stack trace, the dump includes the number of aborts, elision attempts, and a transactional abort code histogram. We found these extremely helpful in pinpointing scalability bottlenecks, and in further optimizing code for performance.

5. Performance Results

5.1 Experiment Settings

We use an Intel 4th Generation Core processor with Intel TSX support. The processor has 4 cores with 2 Hyper-Threads per core, for a total of 8 threads. Each core has a 32 KB L1 data cache, which is used to track transactional states. All tracking and conflict detection are done at cache line granularity, using physical addresses.

For those benchmarks that are distributed in source code, we modified the source to bind application threads so that as many cores are used as possible—e.g., a 4 thread run will use a single thread on each of the 4 cores, while an 8 thread run will also use 4 cores, but with 2 threads per core. We do not alter the thread binding of benchmarks distributed via binaries.

We use a production-quality JVM[3] for this study. The JVM has two modes of execution: interpreted and just-in-time (JIT) compiled mode. Initially, the JVM executes application code through the interpreter, but once it identifies frequently executed code sections, it selectively JITs them to generate native machine code. We apply our transactional lock elision algorithm to the JIT execution mode only; the interpreter does not perform lock elision. The JVM implements adaptive deoptimization and biasing for lock elision (see Section 4.2) by triggering the JIT compiler to re-generate necessary codes. Finally, to better adapt to dynamic execution behavior, when the JVM recompiles a synchronized segment (e.g., due to code path changes), it sets the lock elision state of the monitor lock to `MAYBE`, and re-evaluates transactional elision profitability.

We found that updates to monitor lock statistics counters, especially the number of transaction begins, incur synchronization overheads. To reduce this, we use a stochastic method [9]. Specifically, we maintain a multiplier, and on a transaction begin we increment the counter only if a hardware timestamp (e.g., `rdtsc`) modulo the multiplier is 0. When the JVM dumps the statistics, the counters are compensated with the multiplier to compute the estimated value. Abort counts are incremented precisely. Due to this design, successful transactions with few executions may not be detected, and the total number of transaction begins may be underrepresented.

For all experiments, we maintain the JVM heap size so that garbage collection does not dominate the execution time, yet some degree of garbage collection is present. For statistical fidelity, all data points are averaged over at least 10 executions.

5.2 Microbenchmark Results

In this section we use microbenchmarks to isolate Intel TSX-enabled JVM performance. Specifically, we use two benchmarks that perform concurrent updates on a shared hash table and a hash map, respectively. We can control the ratio of `put()`, `get()`, and `remove()` operations in these benchmarks. For the hash table benchmark, we use `java.util.Hashtable`, which already uses synchronized methods to provide thread safety. For the hash map benchmark, we use `java.util.HashMap`, which does not include synchronization; therefore, we enclose each call to access methods in a synchronized statement block.

Figure 1 shows the performance results. In the figure, **noopt** denotes the baseline thin lock/fat lock implementation described in Section 3. On the other hand, **bl** denotes the baseline with a biased locking optimization [27], which tries to bias a monitor lock towards a thread, and elides some CAS operations if the thread reacquires the lock. This optimization is widely regarded as the state-of-the-art Java synchronization optimization. Lastly, **tsx** shows the performance of the Intel TSX-enabled JVM, described in Section 4. Speedup is over the baseline (**noopt**) with a single thread. We discuss the other data series later in this section.

We see that performance with the non-transactional synchronization schemes (**noopt** and **bl**) does not scale with increasing number of threads; performance actually *worsens* with more

[2] This is not the case for the transactional memory system in [4], where transactional writes do not invalidate other transactions until commit. There, a `wait()` call can lead to a deadlock.

[3] The vendor of the JVM specifically asked us not to disclose the name.

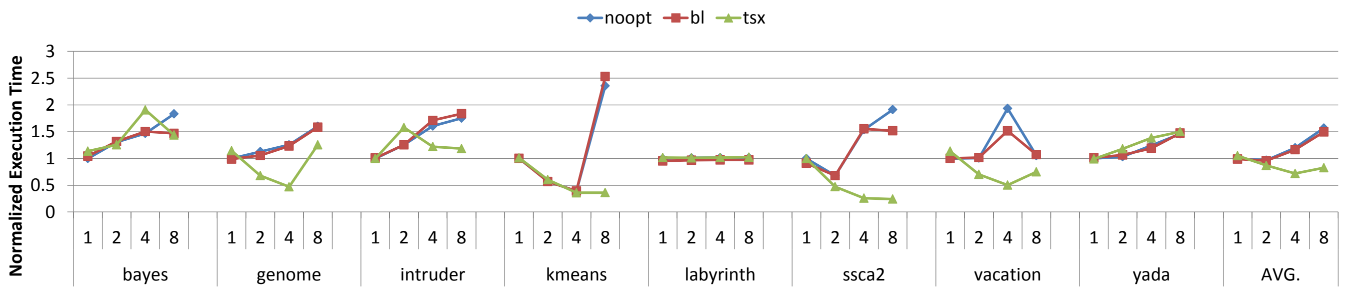


Figure 2: **Intel TSX-enabled JVM performance on Java STAMP workloads. Execution time is normalized to *noopt* with single thread.**

threads. Intel TSX, however, provides significant scalability, by exploiting the concurrency within a critical section. When the operations are read-only[4] (Figures 1a and 1c), scalability is close to linear. Even when there are a significant fraction of write operations (Figures 1b and 1d), Intel TSX performs better than the non-transactional schemes; at 8 threads, the speedup over **noopt** is 14x for `Hashtable` and 11x for `HashMap`, respectively.

As another reference point, for the hash map benchmark, we also evaluate the performance of a concurrent implementation—`java.util.concurrent.ConcurrentHashMap`. This version uses a non-blocking synchronization algorithm to implement the data structure. In Figures 1c and 1d, **cc** denotes the performance. For this concurrent implementation, we do not place the calls to access methods within a synchronized statement block.

When operations are read-only (Figure 1c), eliding the monitor lock with Intel TSX provides comparable scalability to the non-blocking implementation, albeit with somewhat higher overheads. When a significant fraction of operations are writes (Figure 1d), however, Intel TSX provides smaller benefits than the non-blocking implementation. Nevertheless, this performance is achieved with the original binary, rather than relying on a significant rewrite of the data structure.

5.3 Transactional Memory Benchmark Results

We now evaluate our Intel TSX-enhanced JVM on full-fledged workloads. Specifically, we examine STAMP [21], a benchmark suite extensively used by the transactional memory community. The benchmark suite is written in C; starting from [30], we ported the benchmark suite to Java. In this version, critical sections are expressed using synchronized statements, with the application class itself providing the monitor lock—this amounts to synchronizing on a single global lock.

Since the benchmark suite was specifically written to evaluate transactional memory implementations (and was originally written in C), the workloads rely heavily on their own data structure implementations, rather than implementations from a library. Hence, interactions with Java Runtime Environment (JRE) libraries are minimal, and the critical sections are mostly in the application code.

Figure 2 compares the performance of different synchronization schemes. Execution time is normalized to that of the baseline (**noopt**) with a single thread. Table 1 also provides the transactional statistics, measured by the JVM performance debugging feature (see Section 4.4). In the table, `begin` denotes the total number of transaction begins, and among those transactions, `abort` gives the abort ratio. As discussed in Section 5.1, due to the stochastic nature of our shared counter implementation, some samples may be missed, and numbers may not be precise. We nevertheless verified that all the workloads do execute Intel TSX instructions to perform transactional lock elision.

Workload	1 thread		2 threads		4 threads		8 threads	
	begin	abort	begin	abort	begin	abort	begin	abort
bayes								
genome	2.6E6	7%	4.6E6	44%	7.5E6	68%	6.2E7	99%
intruder			2.6E8	95%	2.8E8	95%	2.6E8	95%
kmeans	8.7E5	0%	4.4E6	2%	5.2E6	17%	5.9E6	34%
labyrinth	8.0E0	100%						
ssca2			2.6E7	0%	2.3E7	1%	3.0E7	28%
vacation	6.3E6	50%	1.7E7	78%	3.9E7	94%	1.5E8	102%
yada			3.8E7	91%	3.3E7	83%	3.2E7	56%

Table 1: **Java STAMP transactional statistics. Due to the stochastic counting scheme used [9], reported numbers may not be precise.**

In the figure, as in the microbenchmark results, **noopt** and **bl** represent the performance of the baseline and biased locking-enabled JVM, respectively. Overall, these schemes do not scale; the only data points that see a performance increase from more threads are **kmeans** at two and four threads, and **ssca2** at two threads.

In contrast, Intel TSX manages to improve scalability on many workloads. Specifically, **kmeans** and **ssca2** scale up to 8 threads, due in part to their low transaction abort ratio (see Table 1). Intel TSX also enables scaling of **genome** and **vacation** up to 4 threads, but not with 8 threads. These two workloads exhibit medium to large transactional footprints [21]; since Hyper-Threads share the L1 cache, increasing the thread count from 4 to 8 reduces the effective transactional buffering capacity for each thread, thus increasing the number of transactional aborts.

On the other hand, **intruder** has a critical section that is heavily contended. When Intel TSX-based lock elision is first performed (2 threads), performance suffers due to transaction overheads and abort penalties. However, the abort ratio does not increase with more threads; thus, adding more threads successfully exploits parallelism within the critical section.

Among the non-scaling workloads, **bayes** implements a randomized hill climbing algorithm, and is known to have large variance in execution time [21]. Our results match this known behavior. In contrast, manual code analysis of **labyrinth** shows that the average transaction footprint is about 14 MB, well larger than the L1 cache. Hence, while our stochastic statistics collection failed to collect samples, we know that the transactional executions consistently fail, leading to flat performance as we increase thread count. Finally, the execution time of **yada** is dominated by the garbage collector. The workload inserts/deletes nodes in a graph, resulting in high garbage collector activity. Still, the Intel TSX-enabled version provides similar performance to the other versions.

5.3.1 Sensitivity Study: Transaction Retry Threshold

In Section 4, we introduced the transaction retry threshold, within which a JVM retries transactional execution to elide a monitor lock. In this section we vary the user-specified threshold from 20 to 80,

[4] `java.util.Hashtable` increments a counter even for `get()` operations. We move the increment towards the end of the method to reduce transactional conflicts.

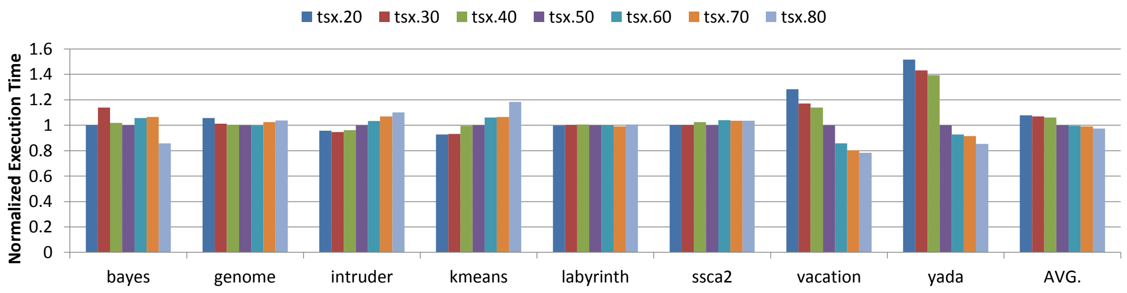


Figure 3: **Java STAMP workloads sensitivity to transaction retry threshold. Execution time is normalized to the case where the threshold is 50.**

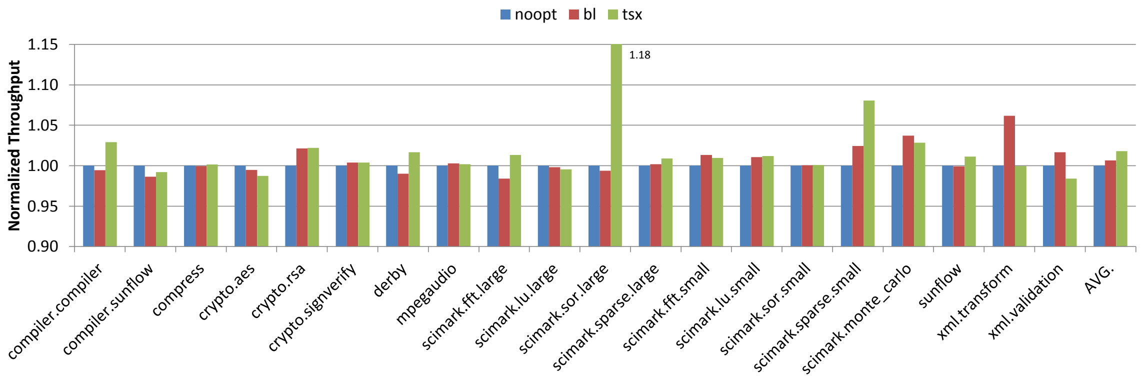


Figure 4: **SPECjvm2008 performance results. 8 threads are used. Performance is normalized to that of *noopt* (higher is better).**

and observe its impact on the Intel TSX-enabled JVM performance. Figure 3 shows the results.

Workloads exhibit varying sensitivity to the threshold—different workloads exhibit different trends, and the best performing retry threshold differs across workloads. Data presented in Figure 2 are from the best performing configuration for each workload.

We see that applications with very few aborts (e.g., **ssca2**, see Table 1) and/or long transactions (e.g., **labyrinth** [21]) are relatively insensitive to the retry threshold. On the other hand, applications with high abort ratios (e.g., **intruder**, see Table 1) and/or short transactions (e.g., **kmeans** [21]) do better with a smaller threshold. The remaining applications benefit from an increased retry threshold up to an inflection point, after which the retry overhead outweighs any increase in the transactional commit ratio.

Based on the above findings, we believe an auto-tuning approach that heuristically adjusts the transaction retry threshold should be quite effective; we leave such a scheme for future work.

5.4 Regular Benchmark Results

In this section we evaluate the Intel TSX-enabled JVM on regular Java benchmark suites: specifically, SPECjvm2008 and DaCapo [3]. Compared to STAMP, a benchmark suite written to evaluate transactional memory systems, these benchmark suites do not make as much use of critical sections, have more complicated monitor use, and exhibit an increased dependence on the JRE library.

5.4.1 SPECjvm2008 Benchmark Suite Results

We executed all of the SPECjvm2008 workloads on our Intel TSX-enabled JVM. All execution results, comprising both transactional and non-transactional executions, passed the benchmark result verification routines.

Even with the baseline monitor lock implementation, we find that many SPECjvm2008 workloads already scale relatively well, diminishing the potential benefit due to transactional lock elision.

Workload	begin	abort	library	lib_abort	cap_abort
compiler.compiler	8.3E2	3%	100%		9%
compiler.sunflow	3.9E6	12%	100%		0%
crypto.aes	9.6E2	46%	55%	66%	8%
crypto.rsa	1.5E7	86%	100%		6%
derby	1.4E9	8%	0%		16%
sunflow	5.3E4	42%	2%	80%	4%
xml.transform	4.8E6	54%	5%	31%	1%
xml.validation	5.6E8	11%	11%	27%	1%

Table 2: **SPECjvm2008 transactional statistics.** ***library*** **denotes the percentage of monitor locks that are in JRE libraries, and** ***lib_abort*** **denotes the abort ratio for those library locks.** ***cap_abort*** **denotes the ratio of aborts that were due to transactional buffering capacity overflow.**

Figure 4 summarizes the results when 8 threads are used. Performance is normalized to that of the baseline (**noopt**).

We see that not all workloads show sensitivity to the monitor lock implementation; many of them spend minimal time executing critical sections, and neither biased locking (average 1.00x over baseline) nor transactional lock elision (average 1.01x over baseline) provides tangible performance improvement.

However, some workloads do exhibit sensitivity to monitor lock performance, and performance improvement due to Intel TSX can be significant. Table 2 shows the transactional statistics collected for such workloads[5].

compiler.compiler uses `java.util.Hashtable`, for which Intel TSX-based lock elision can significantly improve performance (see Section 5.2). Similarly, **crypto.rsa** utilizes a synchronized hash map to implement a lookup cache, and benefits from lock elision. This shows that performing transactional lock elision on key JRE data structures can improve application performance.

Workloads that exercise monitor locks in the application code tend to benefit from transactional execution, as well (small values

[5] **scimark** workloads execute some transactions that rarely abort, and our performance debugging feature failed to collect samples.

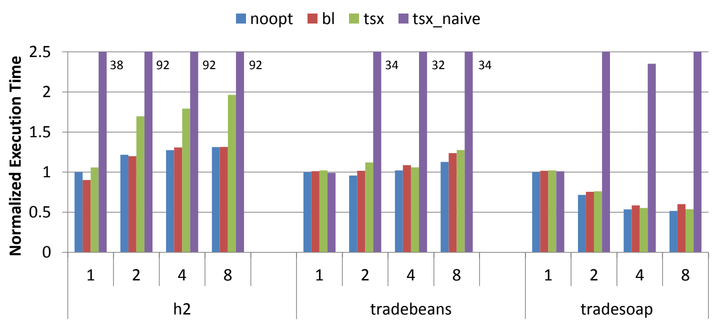


Figure 5: **Effect of performance deoptimization on DaCapo workloads. Execution time is normalized to *noopt* with 1 thread.**

in the `library` column in Table 2). **derby**, in particular, has most of the monitor locks in the application code, and transactional lock elision has few aborts. Similarly, **scimark** workloads use a custom, synchronized random number generator to initialize some of its data structures; transactionally eliding the monitor lock improves performance. **sunflow** workload implements a k-d tree, whose build phase benefits from transactional execution.

In contrast, eliding the more complicated monitors in some JRE libraries turns out to be difficult. In Table 2, the `lib_abort` column shows the transaction abort ratio for monitor locks that are in the JRE. When compared to the overall transaction abort ratio (the `abort` column), we can see that the JRE transaction abort ratio is considerably higher. Inspection of the library codes reveals possible sources of transactional contention, such as debugging counter increments and object allocations (e.g., `new`) within a monitor. Restructuring the codes to pull such segments outside a monitor will better prepare the libraries for transactional execution.

Table 2 also shows the ratio of transactional aborts that are due to transactional buffering capacity overflow (the `cap_abort` column). Across the workloads the ratio is relatively low, demonstrating that Intel TSX buffering capacity (32 KB on our system) is sufficient to transactionally execute standard Java workloads.

Overall, in Figure 4, we can see that **tsx** rarely performs worse than **bl**, the state-of-the-art monitor lock optimization scheme. This indicates that transactional lock elision can be safely applied without worrying too much about performance pitfalls.

5.4.2 DaCapo Benchmark Suite Results

Compared to SPECjvm2008, none of the DaCapo workloads exhibited significant potential for performance improvement through transactional lock elision. Roughly, the workloads could be categorized into two classes, and each class exhibited different reasons for minimal performance improvement.

For the first class, the number of workload threads is determined by the input. These workloads significantly oversubscribe the system—for example, **avrora** maintains on average 30 threads throughout the execution on a 4-core machine. While the Intel TSX-enabled JVM manages to elide a significant number of monitor locks, execution is dominated by OS thread management.

In contrast, in the second class, the number of threads is user-controllable. However, most of these workloads scale well even on the baseline JVM, leaving little room for improvement through transactional lock elision. The remaining workloads do not scale at all, but for reasons other than synchronization [18].

Since the DaCapo workloads have little potential for speedup from Intel TSX, we rather use these to demonstrate that our use of transactional execution does not negatively impact performance. In particular, three workloads, **h2**, **tradebeans**, and **tradesoap**, exhibit poor scalability on the baseline JVM, and naively applying Intel TSX greatly reduces performance.

Figure 5 shows the performance with various JVM implementations. In the figure, **tsx_naive** shows the performance from the Intel TSX-enabled JVM, but with the deoptimization feature disabled.

As can be seen, on **h2** and **tradebeans**, **noopt** and **bl** provide worse performance with more threads. Naively applying Intel TSX to elide the monitor locks (**tsx_naive**) aggravates the situation, and results in a significant slowdown.

These workloads are among the most monitor-intensive workloads in the DaCapo suite [18]. Debugging dumps for these workloads also show that those monitors rarely can be elided; in fact, almost all the transactions abort. Our performance deoptimization feature (see Section 4.2) can detect this situation, and dynamically removes the transactional lock elision code. In the figure, when this feature is enabled (**tsx**), performance is much closer to **noopt** or **bl**.

6. Related Work

As discussed in Section 1, previous work exists on exploring how to apply hardware transactional memory to Java. The biggest difference of our work stems from the *best effort* property of Intel TSX [16]; the system does not *guarantee* that a transaction will eventually commit. While such a design is common for commercial systems [8, 17, 31], it requires a software fallback. As demonstrated, when applied to a JVM for monitor lock elision, this requires careful handling of the existing thin lock/fat lock mechanism. In addition, if not used judiciously, real implementations of hardware transactional memory may reduce performance; thus, a JVM that leverages such support should include adaptive tuning of the fallback, such as our performance deoptimization feature. To the best of our knowledge, no previous work discusses these issues.

Some researchers examined using software transactional memory to support transactional execution of Java programs. Since a JVM does not expose the memory location of an object, these approaches similarly require modifying the JVM [19, 22]. Specifically, these schemes allow a programmer to designate a code section as `atomic`, which the JVM instruments to keep track of transactional accesses. This instrumentation can take place at class load time [19] or at compile time [22]. Since Intel TSX does not require instrumentation, however, such overheads can be avoided.

7. Conclusion

Applying Intel Transactional Synchronization Extensions to elide Java monitor locks shows promising results. By devising a software fallback algorithm that coexists with the underlying JVM locking mechanism, and by implementing an adaptive tuning approach that selectively deoptimizes transactional lock elision where necessary, we obtain performance improvements on varying ranges of workloads. On a standard Java benchmark suite, we observe up to 1.18x performance improvement. Proposed modifications are confined to the JVM, and applications running on top of a JVM transparently benefit without source or bytecode modifications.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We also thank Konrad Lai, Ravi Rajwar, Justin Gottschlich, and Tatiana Shpeisman for their help and feedback on this project.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[2] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 258–268.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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 *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[4] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Executing Java programs with transactional memory. *Sci. Comput. Program.*, 63(2):111–129, 2006.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 519–538, 2005.

[6] C. Click. Azul's experience with hardware transactional memory. In *HP Labs Bay Area Transactional Memory Workshop*, 2009.

[7] Cray Inc. Chapel Language Specification 0.796, 2010.

[8] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.

[9] D. Dice, Y. Lev, and M. Moir. Scalable statistics counters. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 43–52, 2013.

[10] P. Dudnik and M. M. Swift. Condition variables and transactional memory: Problem or opportunity? In *the 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[11] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, 2004.

[12] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[14] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[15] Intel Corporation. From a few cores to many: A tera-scale computing research review. White Paper, 2006.

[16] Intel Corporation. Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions. 2012.

[17] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36, 2012.

[18] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 335–354, 2012.

[19] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive software transactional memory in Java. *Transactions on HiPEAC*, 5(2), 2010.

[20] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.

[21] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.

[22] M. Mohamedin, B. Ravindran, and R. Palmieri. ByteSTM: Virtual machine-level Java software transactional memory. In *the 8th ACM SIGPLAN Workshop on Transactional Computing*, 2013.

[23] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.

[24] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1996.

[25] Oracle. The Fortress Language Specification Version 1.0, 2008.

[26] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 294–305, 2001.

[27] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 263–272, 2006.

[28] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58–71, 1993.

[29] Transactional Memory Specification Drafting Group. Draft specification of transactional language constructs for C++, version: 1.1. 2012.

[30] University of California Irvine. Java STAMP benchmark suite version 0.5, `http://demsky.eecs.uci.edu/software.php`.

[31] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136, 2012.

[32] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, 2008.

[33] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 19:1–19:11, 2013.

[34] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 129–154, 2008.

Notice and Disclaimers