Hybrid STM/HTM for Nested Transactions on OpenJDK

Keith Chapman[†] Antony L. Hosking[†*] J. Eliot B. Moss[$]

[†]Purdue University, USA [*]Australian National University / Data61, Australia [$]University of Massachusetts, USA
keith@cs.purdue.edu antony.hosking@anu.edu.au moss@cs.umass.edu

Abstract

Transactional memory (TM) has long been advocated as a promising pathway to more automated concurrency control for scaling concurrent programs running on parallel hardware. Software TM (STM) has the benefit of being able to run general transactional programs, but at the significant cost of overheads imposed to log memory accesses, mediate access conflicts, and maintain other transaction metadata. Recently, hardware manufacturers have begun to offer commodity hardware TM (HTM) support in their processors wherein the transaction metadata is maintained "for free" in hardware. However, HTM approaches are only *best-effort*: they cannot successfully run all transactional programs, whether because of hardware capacity issues (causing large transactions to fail), or compatibility restrictions on the processor instructions permitted within hardware transactions (causing transactions that execute those instructions to fail). In such cases, programs must include failure-handling code to attempt the computation by some other software means, since retrying the transaction would be futile. Thus, a canonical use of HTM is *lock elision*: replacing lock regions with transactions, retrying some number of times in the case of conflicts, but falling back to locking when HTM fails for other reasons.

Here, we describe how software and hardware schemes can combine seamlessly into a hybrid system in support of transactional programs, allowing use of low-cost HTM when it works, but reverting to STM when it doesn't. We describe heuristics used to make this choice dynamically and automatically, but allowing the transition back to HTM opportunistically. Our implementation is for an extension of Java having syntax for both open and closed nested transactions, and boosting, running on the OpenJDK, with dynamic injection of STM mechanisms (into code variants used under STM) and HTM instructions (into code variants used under HTM). Both schemes are compatible to allow different threads to run concurrently with either mechanism, while preserving transaction safety. Using a standard synthetic benchmark we demonstrate that HTM offers significant acceleration of both closed and open nested transactions, while yielding parallel scaling up to the limits of the hardware, whereupon scaling in software continues but with the penalty to throughput imposed by software mechanisms.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—code generation, compilers, incremental compilers, run-time environments

General Terms Design, Experimentation, Languages, Measurement, Performance

Keywords transactional memory, nested transactions, hardware transactional memory, software transactional memory, Java

1. Introduction

Transactional memory (TM) allows programmers to group memory operations into *transactions* that appear to execute atomically: no transaction sees the intermediate states of other transactions executing in other threads, and all work of a transaction either happens (the transaction *commits*) or not (the transaction *aborts*). Transactional memory is more abstract than locking, and avoids many of the problems encountered with locks, such as deadlock, priority inversion, convoying, pre-emption, and reduced concurrency.

Transactional memory systems track memory read and write operations performed against disjoint memory units. When two transactions access the same memory unit and at least one of the accesses is a write then there is a *conflict*: one of the transactions must abort (discarding its pending writes) and restart. The transaction system must also manage atomicity: either all of a transaction's writes occur, or none of them, and to other transactions the writes appear to occur all at a single instant in time.

Software transactional memory (STM) systems track memory accesses in software, usually at the logical level of fields or objects of a host programming language. The overhead of this software instrumentation results in loss of throughput for

memory accesses. Nevertheless, STM systems can still scale better than non-transactional synchronization schemes (such as locking) because of increased concurrency.

In contrast, hardware transactional memory (HTM) systems track memory accesses in hardware at the physical level of bytes, words, or cache lines, with little or no throughput overhead. However, current HTM proposals and implementations such as Intel's Transactional Synchronization Extensions (TSX), IBM's System Z, and AMD's ASF, offer only *best-effort* hardware transactions: they can fail even when there are no conflicts [4, 6, 9, 15, 26, 27]. Such reasons for failure include lack of hardware capacity to track accesses, compatibility restrictions on instructions permitted to execute with transactions, page faults, and other hardware interrupts. As a result, HTM systems require software to take over when a hardware transaction cannot profitably be retried. For example, *hardware lock elision* (HLE) replaces lock regions with transactions, retrying some number of times in the case of conflicts, but falling back to lock acquisition when HTM otherwise fails [7, 25].

Proponents of transactional memory have long devised various models for aggregating nested execution of atomic actions into larger transactions. Most systems (including existing commercial HTM) simply fold the operations of nested transactions into the top-level outermost transaction, forming one large *flat* transaction. In this case, any conflict arising in a nested transaction will cause the top-level transaction to abort, discarding all of its effects. However, some systems allow nested transactions to abort independently of the parent while preserving the parent's atomicity, avoiding the loss of work performed by the parent due to a conflict by the child. *Closed* nesting [21] still aggregates the effects of nested transactions into their parent on commit, but allows retry of a nested transaction when it aborts, without necessarily aborting the parent. Other work [22, 23] has proposed *open* nesting as an extension to closed nesting, allowing improved concurrency at the cost of some programmer effort. This approach relies on having programmers annotate open transactions with an abstract *undo* action that can be used by the parent to revert the effects of the child if the parent aborts. The nested atomic actions still execute as transactions, with the usual conflict detection for their memory operations to ensure atomicity, but when they commit their memory effects become permanent and globally visible. The undo action allows their effects to be rolled back if the parent aborts. Open nesting also requires *abstract concurrency control*, so as to detect abstract conflicts between transactions that occur at the level of the abstract operations encapsulated by the open nested transactions.

Our focus in this paper is the extent to which these alternatives for nesting transactions can be accelerated by using HTM where possible, to avoid the respective overheads of their STM implementations, while allowing fallback to STM execution when the hardware fails to provide. In particular, we desire a system that presents a full-blown general transactional programming framework for Java, while automatically and dynamically choosing when to use HTM versus STM, and where hardware and software variants can execute concurrently and seamlessly while preserving transaction semantics.

Our Contributions Include:

- a full-fledged implementation of extended Java language abstractions for *nested* transactions (where children can fail independently of parents, as opposed to the flattening of other systems) to gain improved concurrency; we call this system XJ (for transactional Java);
- a hybrid HTM and STM scheme for nested (and boosted) transactions in Java, allowing HTM and STM to execute concurrently and compatibly;
- use of HTM where possible with adaptive seamless reversion to STM where not, for good performance;
- execution under optimized compilation with the high-performance (HotSpot-based) OpenJDK for Java, with minimal modification to the HotSpot compilers to add HTM intrinsics;
- demonstrating via experiments using an established benchmark that HTM can significantly boost throughput and that falling back to STM does not compromise scalability; and
- showing that open nesting increases the envelope of concurrency and transaction sizes that can be accommodated in hardware.

The broader implications of our work are that programmers can easily make use of transactional programming abstractions to build scalable concurrent data structures without needing to devise complicated implementations using low-level synchronization primitives. Moreover, these transactional implementations can benefit from hardware acceleration on current hardware for modest transaction sizes and degrees of concurrency. We also suggest that HTM would be even more useful if its capacity were higher.

2. Background

Researchers and implementers have explored a number ways in which transactions might be nested. A natural form of nesting for transactional constructs in a programming language is *linear nesting*, which allows a *parent* transaction to invoke a sequence of sub-operations, some of which may themselves also execute as *child* subtransactions. How these subtransactions are managed may vary, so long as the atomicity of the parent transaction is preserved. If the parent transaction aborts and its effects are discarded, then the effects of its committed children must also be discarded. Nesting is desirable when aggregating atomic operations against underlying data structures into larger transactions. For example, a transaction transferring a balance from one bank account to another needs to debit from one account while crediting the other,

both operations ideally appearing to occur simultaneously, perhaps to avoid arbitrage. The debit and credit operations must themselves be implemented as atomic operations. Performing the transfer as a transaction that executes the nested debit and credit actions (in either order, it does not matter) satisfies the requirement that the balance be seen to be in one account or the other at all times. Linear nesting matches well both static nesting of program blocks in one another and the dynamic nesting patterns of calls and returns. Hence our transactionalized version of Java uses linear nesting.

Approaches to handling linear nested transactions that we consider in this paper include *flattening*, *closed and open nesting*, and *boosting*.

2.1 Flattening

Flattening ignores the nesting structure and runs the operations of any nested transaction as part of its parent. If a nested transaction aborts, then the entire top-level transaction also aborts. Thus, all the work of the top-level transaction must be discarded and retried. Flattening means that no metadata for nested transactions needs to be maintained, other than a simple counter to track nesting depth—entering a nested transaction increments the counter, and committing decrements the counter. When the counter decrements to zero the top-level transaction commits its writes and they become globally visible.

Current HTM implementations, such as Intel's TSX, flatten hardware transactions. As a result, they are susceptible to failure if they run for a long time (increasing the likelihood of conflicts or interrupts) or touch a large amount of memory (exceeding capacity).

2.2 Closed Nesting

Closed nesting allows a nested transaction to abort independently of its parent. A closed nested transaction can successfully commit, in which case its reads and writes accrue to its parent. If the child aborts then its writes are discarded and the nested transaction can be retried. After some number of unsuccessful retries the parent itself may be aborted (or the parent might attempt some other action). Closed nesting sometimes avoids the need to discard the accumulated effects of a parent. On the other hand, as the write sets of a series of linear nested transactions accrue to the parent, its chances of failure due to conflict with other transactions will increase, because the write sets are larger and held longer.

Two *nested* transactions conflict as before (if they both access the same memory unit and at least one of them writes it), excepting that a child never conflicts with its ancestors. Thus, writes by children override writes of their ancestors without conflicting. Similarly, reads by children do not conflict with writes of their ancestors (but need to see the value most recently written by ancestors and previously committed descendants).

2.3 Open Nesting

Open nesting allows *further* increases in concurrency [23], by releasing concrete resources (e.g., memory reads and writes) earlier and applying conflict detection (and roll back) at a higher level of abstraction. For example, transactions that increment and decrement a shared memory location would normally conflict, since they write to the same location. But, since increment and decrement commute as abstract operations, they can be implemented correctly with open nesting. An increment (say) does: read, add-one, write. The open nested transaction would be over, its writes made globally visible, and the updated field would not be part of the parent transaction's read or write set. Instead, if the parent later aborts, it must run a *compensating* decrement to undo the logical effect of its committed open nested child.

The only difference between open and closed nesting with respect to memory accesses concerns what happens when a transaction commits. When an open nested transaction commits then its writes become permanent and *globally* visible; they do not accrue to its parent. Moreover, for each of its writes any corresponding read by its ancestors from the same location is also forgotten (so that its ancestors can no longer have conflicts on that location).

Instead of conflict detection being performed on the concrete level of memory units, when a committing open nested transaction releases its concrete reads and writes, it must typically claim some (set of) *abstract* resource(s) ("abstract locks") and provide a corresponding abstract compensation operation (e.g., the decrement in the earlier example) for use by its ancestors if they need to abort and undo the child.

If we view transaction conflicts and rollback in terms of *operations*, we can see greater similarity between closed and open nesting and highlight better the essential difference. Closed nesting works in terms of *read* and *write* operations, with the usual conflict rules on those operations. The undo of a *write* is a corresponding *write* that installs the original value of the memory unit. In the open nesting case we have a programmer-defined set of operations, with programmer-defined conflict rules and programmer-supplied rollback operations for each forward operation. So the essential difference when viewed from "outside" the transaction is the set of operations over which the transaction operates.

However, the more abstract[1] transactions provided by open nesting—which offer increased concurrency because abstract concurrency control captures the essential semantic conflict while read/write level conflict detection over-estimates conflicts—must be built from *something*, and the individual operations must still appear to execute atomically. More precisely, they must be *linearizable* [13]: they must appear to occur at a single instant of time. Transactions are one way to achieve that linearizability, so it is natural to *implement* open

[1] We mean "abstract" in that conflicts don't occur at the physical level.

nesting using much the same mechanism as for closed.

Interestingly, because open nested children discard their physical reads and writes they are particularly amenable to acceleration using hardware, even when their parent runs in software. All that needs to be done is to ensure that the necessary abstract locks are acquired before the hardware open nested child commits. By storing abstract lock meta-data in a carefully-implemented (non-transactional) concurrent data structure the abstract locks can simply be acquired before entering the open nested hardware transaction (so avoiding placing the burden of managing the locks on the hardware transaction, and leaving it only to track application-level memory accesses).

2.4 Boosting

Transactional boosting [12] recognizes that *how* linearizability is achieved does not matter, and thus naturally supports an approach where existing *non-transactional* (but otherwise thread-safe (linearizable)) code is extended with transactional wrappers. For example, given a thread-safe data structure such as Java's `ConcurrentHashMap`, where concurrent operations to manipulate the map are linearized using low-level non-blocking primitives, linearizability of the *composition* of sets of these operations can be achieved using the same abstract concurrency control mechanisms as for open nesting. Instead of using transactions to linearize the sub-operations (say, adding and removing from the map), transactions are used only to linearize aggregations of those sub-operations.

For example, an aggregate operation that adds two elements to a `ConcurrentHashMap` can be linearized with respect to other operations on the map by locking the visibility of those elements until the aggregate operation completes, and providing a compensating action that removes the elements if the aggregate must be rolled back.

The advantage of transactional boosting is that it removes the need to manage low-level conflicts using transactions, which in the case of software transactional memory can have significant overhead. Instead, the underlying data structures support linearizability of their operations using other means, such as low-level hardware atomic operations. Software transaction mechanisms come into play only when it comes to aggregating these operations, capturing their resource reservations in the form of abstract locks.

2.5 Related Work

We now briefly discuss other related work, before describing in later sections how to present our abstractions to the programmer, how they can be implemented so as to be compatible with and amenable to HTM acceleration, and experiments showing the impact they have on performance.

There are previous hybrid STM/HTM implementations, such as HyTM [5, 18]. Their approach is similar to ours, where they generate separate software paths for HTM and STM with instrumentation to check the needed metadata. HyTM supported two simple back-off schemes to transition from HTM to STM in the face of failures. In the "immediate fail-over" scheme a transaction failing in HTM retries itself in STM immediately. In the "back-off" scheme, a transaction failing in HTM retries for 10 times before retrying under STM. Since their transactions were very short and with small memory footprint, their simple approach of trying HTM first for every transaction was a successful policy. Matveev and Shavit [20] describe a similar back-off policy.

PhTM [19] took an alternative approach, running transactions in phases. Under this scheme transactions cannot run in HTM and STM concurrently: it is either all HTM or all STM. This works well when all transactions succeed under HTM, but incurs major overheads if even one HTM transaction fails.

Recent work by Diegues and Romano [8] explores a self-tuning retry policy developed with reinforcement learning to decide when to use the fallback path for TSX. This work falls in the space of lock elision rather than full-blown transactions.

We are the first to consider nesting in both closed and open forms with respect to seamless combination of transactions executing under both HTM and STM. Unlike these previous approaches, we also avoid trying hardware when past history shows that the transaction is unlikely to succeed.

3. Nested Transactions for Java

We build upon the work of Ni et al. [23] and Chapman et al. [1] to allow expression of closed nested, open nested, and boosted transactions in Java programs. The overall system is called XJ (for transactional Java), comprising language design, translation framework from syntax to bytecode, and run-time system. In this section, we reprise the language extensions of Chapman et al. [1] and describe their current implementation, which differs from Chapman et al. [1] in several key aspects.

3.1 Closed Atomic Blocks/Methods

A block (or method) is designated closed atomic by writing **`xatomic`** (meaning "transactional atomic") wherever the **`synchronized`** keyword is allowed. These execute as closed atomic transactions. Their *effects* include assignments to all declared variables and fields, as well as the effects of nested transactions that they execute.

3.2 Open Atomic Classes and Abstract Locking

Open nesting naturally applies to *classes* that implement abstract data types rather than individual methods, because all operations of the abstract data type need to cooperate in providing suitable abstract concurrency control and recovery. A class declared **`openatomic`** indicates that each of its public methods executes as an open transaction, and that its fields can be accessed only during execution of those methods.[2] Each public method may also record a compensating action

[2] It is possible to have other closed atomic blocks and non-public methods, but the open atomic methods need to be designed together to preserve the abstract state of the open atomic class.

in the form of an **onabort** clause to undo its effects in the case of abort by an enclosing (ancestor) transaction. It may also have a **locking** clause to express abstract locks that it must acquire before it can commit. Only these two accrue[3] to an enclosing parent on commit, while the physical operations do not. The abstract locks typically protect the ability of an aborting parent transaction to roll back the effects of the method by running the **onabort** clause as a transaction, and to protect against conflicting abstract operations.[4]

Aborting a transaction rolls back its (accrued) physical operations (including those from closed nested children) and also runs the accrued **onabort** clauses of its open nested children (as open nested transactions). These rollback operations are executed in reverse order from which they accrued.

Following Chapman et al. [1] we allow users to construct rich abstract locking protocols. The locking framework relies on the notions of *locks*, *lock spaces*, *lock shapes*, and *lock modes*. The **locking** clause of an open atomic method requests locks of particular shapes in particular modes from lock spaces. The type signatures of these are illustrated in Listing 1. (The metaphor here is of possibly overlapping geometric shapes within some space. A shape indicates *what* is being locked, while a lock *mode* describe *how* it is being accessed.)

An instance of an open atomic class will typically have some number of lock tables in which to record abstract locks held by active transactions against the *abstract* state of the instance. Lock tables record locks and the mode in which they are held, along with the transaction holding the lock. Locks come in multiple shapes, as defined by a lock space, allowing a single lock to cover a range of locked values.

As an example, consider the design of an open atomic class `OrderedSet<T>` implementing `java.util.SortedSet<T>`. A suitable lock space for an ordered set is the one-dimensional set of all possible `T` instances, having a total order (`OneDSpace`). Within this space one can imagine a number of lock shapes:

Point(x): lock a single "point" object, associated with a particular `T` instance x, which mathematically could be considered the range $[x,x]$;

GT(x): lock upward "rays" starting at x, meaning $(x,\infty]$;

LT(x): lock downward "rays" starting at x, meaning $[-\infty,x)$;

Range(x,y): lock ranges defined on values x and y where $x \leq y$ in the total order, meaning (x,y), etc.

We will use two lock mode classes here, `SXMode`, shown in Listing 2, and `PCMode`, shown in Listing 3. `SXMode` provides

[3] The language extensions also support **oncommit**, **ontopcommit**, and **onvalidate** clauses, not discussed here, which also accrue.

[4] Notice that a given object will be used either in an open atomic way or not, and no public access is allowed to its fields. This prevents any access patterns by which mixed accesses in closed and open atomic modes to the same object can lead to deadlock when trying to execute compensating actions while aborting an open transaction.

```
public interface LockTable
  <LT extends LockTable<LT>> {
    public void acquireLock
      (LockShape lockShape, LockMode mode,
       TxnDescriptor desc) throws
          LockConflictException;
    public void releaseLock(Lock lock);
}

public interface Lock
  <S extends LockShape, M extends LockMode> {
    public S getLockShape();
    public M getLockMode();
    public TxnDescriptor getTxnDescriptor();
    public LockTable getLockTable();
}

public interface LockSpace
  <M extends LockMode<M>, LS extends
      LockSpace<M,LS>>
  extends LockTable<LS> {}

public interface LockShape
  <M extends LockMode<M>, LS extends
      LockShape<M,LS>>{}
```

Listing 1. Lock tables, spaces, shapes, and modes

```
enum SXMode implements LockMode<SXMode> {
  S { public boolean conflictsWith(SXMode
      other)
       { return other != S; } },
  X { public boolean conflictsWith(SXMode
      other)
       { return true; } },
}
```

Listing 2. Shared/eXclusive lock modes

```
enum PCMode implements LockMode<PCMode> {
  P { public boolean conflictsWith(PCMode
      other)
       { return other != P; } },
  C { public boolean conflictsWith(PCMode
      other)
       { return other != C; } },
}
```

Listing 3. Pin/Change lock modes

```
openatomic class OrderedSet<T> ... {
  private final
    LockSpace<SXMode,OneDSpace<SXMode,T>>
    eltSpace;
  private final
    LockSpace
      <PCMode, UnitSpace<PCMode,OrderedSet<T>>
    setSpace;
  public boolean add(T elt) locking
    (eltSpace : point(elt) : SXMode.X),
    (setSpace : get() : PCMode.C) ...
  public boolean remove(T elt) locking
    (eltSpace : point(elt) : SXMode.X),
    (setSpace : get() : PCMode.C) ...
  public int size()
    locking (setSpace : get() : PCMode.P) ...
  public boolean contains(T elt)
    locking (eltSpace : point(elt) : SXMode.S)
        ...
  ...
}
```

Listing 4. `OrderedSet` lock tables

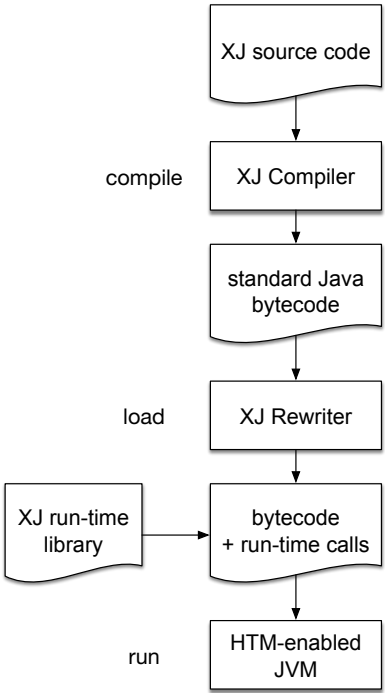


Figure 1. XJ Tool Chain

`S` (share) and `X` (exclusive) locks (also often called read/write locks). `S` and `X` modes are used concerning the presence/absence of individual elements of a set. For the set as a whole, we can *pin* the state of the set using `P` mode, or indicate some *change* to the set using `C` mode: operations like `size` would use `P` mode, and `add`/`remove` operations would use `C` mode on the set (plus `X` mode on individual elements). Note that `C` conflicts with `P` but not with `C`. The two mode classes `SXMode` and `PCMode` are strictly different.

Lock modes are naturally implemented using Java **`enum`** classes that implement the `LockMode` interface.

An `OrderedSet<T>` might then have two lock tables, one for the set of individual elements and one for gross statistics (current size, total number of insertions/deletions, etc.) about the set as a whole, as in Listing 4. In this example, `UnitSpace` is a space that allows locking just one object, in this case the set as a whole.

For simplicity in our later examples we consider only unordered sets, whose lock space for their elements is a simple point space (`PointSpace`) supporting just one `point` shape for locking distinct elements (not ranges). The operations on the set use locking clauses to protect their abstract state changes. Thus, for example adding/removing an element locks the presence/absence of that element in the set by taking an `X` lock on that element, whereas `contains` takes an `S` lock.

3.2.1 Boosting

The framework for abstract locks extends naturally to boosting. In place of **`openatomic`** we allow a class to be declared **`boostedatomic`**. This presumes that the implementations of its methods are inherently linearizable, such as by being implemented using non-blocking hardware primitives instead of executing as open nested transactions.

3.3 Implementation

Our implementation extends that of Chapman et al. [1], comprising four main components: (1) a compiler front-end based on OpenJDK's `javac`, (2) a minimally modified version of OpenJDK to support hardware transactional memory using TSX, (3) a Java agent for load-time bytecode rewriting to inject transaction support, and (4) a run-time library to manage the dynamics of transactions and abstract locking. Figure 1 shows how these components fit together. The rewriter generates an interface for each class and moves the implementation into a separate class. These transformations are not currently compatible with the default methods of Java 8 because a default method cannot be invoked on an interface implemented indirectly. Due to this limitation we currently work only with Java 7.

3.3.1 Java Compiler

Our modified `javac` compiler supports the new `boostedatomic` keyword, in addition to **`xatomic`**, **`openatomic`**, and **`onabort`** clauses. It generates compliant Java bytecode compatible with any standard Java virtual machine. The compiler treats **`boostedatomic`** just like **`openatomic`** but the load-time rewrites for **`boostedatomic`** classes differ significantly since the methods do not execute as transactions (though they generally acquire abstract locks and have **`onabort`** clauses).

```
public final class Unsafe {
...
    public static int beginHWTxn() { return 0;
        }
    public static void endHWTxn() {}
    public static void abortHWTxn(int flag) {}
...
}
```

Listing 5. HTM methods added to sun.misc.Unsafe.java

3.3.2 OpenJDK Modifications

In order to make use of the new TSX instructions to support HTM we need a modified Java virtual machine capable of injecting them into compiled code. We augmented version 7u40-b23-2013-08-26 of OpenJDK. The TSX specification provides two different interfaces to programmers. While both interfaces make use of the underlying TM hardware, their purpose is quite different. The Hardware Lock Elision (HLE) interface is used to implement hardware lock elision techniques while the Restricted Transactional Memory (RTM) interface resemble classic TM proposals. We use RTM since it is more amenable to implementing HTM.

We modify the non-standard `sun.misc.Unsafe` class of OpenJDK as shown in Listing 5 to provide methods that begin, end, or abort a hardware transaction. We do not provide any concrete implementations of these methods here, but instead provide their implementations via HotSpot compiler and interpreter intrinsics [16]. We use `sun.misc.Unsafe` as a mere interface to communicate between the user code and the HotSpot compilers (both C1 and C2) and interpreter. Providing intrinsic implementations of these methods avoids the overhead of calling them as native code routines. These intrinsics were the only extensions we made to HotSpot.

The `beginHWTxn` method uses the new `XBEGIN` instruction. If the transaction completes successfully it returns -1; in the failure case it returns the corresponding error code stored in the `EAX` register. This code can be used to diagnose the reason for the hardware transaction's failure. The `endHWTxn` method uses the new `XEND` instruction, which indicates the end (commit point) of a hardware transaction. The `abortHWTxn` method can be called if the transaction needs to be aborted explicitly. This method uses the `XABORT` instruction and takes an **`int`** flag as an argument, which fills in part of the `XBEGIN` result code in `EAX`, allowing the caller of `abortHWTxn` to convey a few bits of information outside the aborting transaction.

In our initial experiments, many simple hardware transactions surprisingly failed due to conflicts even when there were no writes involved. We diagnosed this issue using the Intel Software Development Emulator (SDE) and found the conflicts to occur on accesses to a bookkeeping field of the `Node` class in the tree data structure manipulated by our benchmark. These conflicts turned out to be due to false sharing because

```
// i and j are unique for each transaction
static int i = 1;
static int j = 0;
method (args) {
  if (j == 0) { method_htm(args, false); }
  else {
    j = j - 1;
    method_stm(args);
  }
}
```

Listing 6. Pseudo-code for the routing method

TSX operates at the granularity of a cache line. Java 8 introduced the `@Contended` annotation to be used to prevent such false sharing. We back-ported this feature to Java 7 and added suitable `@Contended` annotations to the benchmark code.

3.3.3 Bytecode Rewriter

The load-time bytecode rewriter is a Java agent built using the Java Virtual Machine Tool Interface (JVMTI). It runs as a separate process, and can rewrite all loaded classes (including those loaded by the bootstrap class loader). In contrast to Chapman et al. [1], we do not replace all classes with rewritten transactionalized versions. Instead, having a clean separation between application code that is rewritten for transactional execution and the run-time library code that *supports* transactions avoids entanglement and complexity. There is no need to produce code that must be made to serve in both the run-time and the application contexts, with the associated run-time overhead needed to distinguish the context. For bootstrap classes we generate a new version of the class under a different package name, while also preserving the original class. The rewritten class has no relationship to the original, other than that its source was the original class. Application classes (loaded by the application class loader) are rewritten to refer to the new bootstrap classes rather than the originals, while the transactional run-time library classes, being infrastructural in nature, continue to use the original versions. This creates a clean separation between the run-time library and the application.

There are situations where the separation does not always work, such as when the original class has native methods. We treat these classes specially by generating a transactionalized wrapper class for the application to use. The wrapper maintains the relationship between the original and wrapped instances, and serves as a transactional proxy for the original class. There are also a handful of classes that do not need transactional machinery (such as those whose instances are immutable). We do not rewrite these classes and refer to them directly from the rewritten classes.

For each of the generated transactionalized versions of the classes we apply a series of transformations. These transformations generate new methods, as well as transform

```
method_htm (args, boolean runInSW) {
  TD desc = TD.getNewDescriptor(); // TD =
      TxnDesc
  int numRetries = 0;
  while (true) {
    int txnStatus = TD.beginOpenHtm(desc,
        runInSW);
    if (txnStatus == -1) { // running in HTM mode
      // Method body goes here
      TD.commitOpen(desc);
      j = 0;
      return;
    } else if (WARMUP_PHASE) {
      if (TD.retryInSWMode(txnStatus,
          numRetries) {
        method_htm(args, true);
        return;
      }
      numRetries++;
      continue;
    } else { // HW transaction failed
      if (TD.retryInHW(txnStatus, numRetries) {
        numRetries++;
        continue;
      }
      // Back off to SW mode
      j = i;
      i = i * 2;
      method_stm(args);
      return;
    }
  }
}
```

Listing 7. Pseudo-code for the HTM version of a method

existing methods in order to add the transactional machinery. In general, transactional programs can run with or without HTM support. When the system is run with support for HTM, the bytecode instrumenter performs a series of additional method transformations. We generate a transactional version of the method for both HTM and STM. The STM version calls run-time routines that support STM, while the HTM versions call routines that support HTM. Examples of such run-time calls are `openForRead` (indicating that an object is about to be read), `openForWrite` (indicating that an object is about to be written), `beginTxn`, `endTxn`, etc. We also generate a *routing method* that direct the caller to either the STM or HTM version of the method, with the decision guided by two special variables we generate for each routing method. Listing 6 shows pseudo-code for the routing method, and Listing 7 for the HTM version of a method. Although these show `i` and `j` as static members of the class, in reality they are encapsulated in a separate object referenced from a **`static final`** field. This guarantees that updating of `i` and `j` will not cause false conflicts with other transactions. The idea behind the scheme is that we first try to run a transaction in HTM. In the face of HTM failures we back off to STM aggressively, yet try HTM once in a while. If a transaction succeeds in HTM, then we will keep trying it in HTM.

This simple scheme worked well when the number of threads was low, but failed to yield its true potential as the number of threads increased. It was backing off too aggressively and not attempting enough times in HTM, which limited the throughput that we could achieve. To remedy this, we introduced thread-local counters so that most counting down occurs per-thread rather than against shared counters. We call the thread-local counters `decrementCounter` and `updateCounter`, and they are initialized to 10. Each time the shared `j` would have been decremented, we check if the thread-local `decrementCounter` is at 0, and if so we decrement the shared `j` and reset `decrementCounter` to 10. Otherwise we just decrement the thread-local `decrementCounter`. The same goes for updating `i` and `j` when backing off from STM to HTM, using `updateCounter` in place of `decrementCounter`. This scheme ensures that the back-off rate does not change drastically as the number of threads increases. The scheme we use is much simpler than that of Diegues and Romano [8], who used a reinforcement learning technique to decide when to use the fallback path for TSX.

One of the main issues we encountered early on with using HTM was that many transactions failed with result code 0 (i.e., no specific reason given). Using the Intel SDE, we found these aborts to be caused by execution of instructions that are incompatible with TSX [14]—`FXRSTOR` and `FXSAVE` (perhaps among others)—and which are compiled into HotSpot's run-time stubs used to control dynamic optimization and linking, and to resolve Java static and virtual method calls. By design, HotSpot patches these call sites at run time [24]. Thus our hardware transactions always failed, *and* those failures preventing triggering of the patching mechanism. Our workaround was to devise a mechanism to "warm" the system up in STM mode before attempting any hardware transactions. However, so that the compiler's optimizations will be triggered appropriately, and so that linking/patching will occur, these STM transactions must follow the same code path (except for not using the `XBEGIN` instruction, etc.) as HTM transactions do. We use a global flag to indicate whether we are in the software-only warm up phase. The `i` and `j` values described above are used only after warming up.

The bytecode rewriter generates code that preserves many important invariants related to possible transitions between STM and HTM code. We follow a few simple rules. The HTM version of a method always calls the nested HTM version of other methods. The nested HTM version of a method is much simpler than the one presented in Listing 7. Since nested HTM methods will always be called from an HTM

context, they do not need to begin a new transaction, and thus they contain only the instrumented method body. On the other hand, the STM version of the method calls the method with the original name. Thus, if the method being called is a transactional method, it will call the routing method. This enables the new transaction to run under *either* HTM or STM. There is one caveat though: a parent transaction running under STM should not create a nested closed hardware transaction (since it will not gather locks and log records and accrue them to the parent).[5] In contrast, if the parent transaction is running under STM then an *open* nested child transaction can safely run under HTM. This is because the open nested transaction will acquire abstract locks and undos and can release all physical locks (making HTM possibly profitable in this case). We acquire the abstract locks and log undos before starting the hardware transaction, and release/revert them if the hardware transaction fails. We further optimize the case where an *open* nested action runs in hardware under a top-level hardware transaction. Such a child does not need actually to acquire locks or log undos, since they will be immediately discarded on either success or failure of the hardware transaction. However, to detect conflicts, the child must check that it *could* have acquired the locks—i.e., that there are no conflicting locks held by other transactions.

Summary of Method Versions: For methods not marked atomic, we generate a non-transactional version and three closed nested versions: a routing method and STM and HTM versions. For methods marked closed atomic, there are top-level routing, STM, and HTM versions, and nested routing, STM, and HTM versions, and likewise for open atomic and boosted methods (except there are no HTM versions under boosting). Thus a given method has four to seven versions. In principle, these could be generated on demand to avoid code bloat, but we have focused on benchmarks whose code is not particularly large and have not had code size problems. Thus, HTM is not enabled only at leaf transactions, though it is true that HTM forces flattening. Top-level (parent) transactions can start in hardware and run their children (both open and closed) as subsumed (flattened) HTM transactions. If this succeeds (because both parent and children fit in hardware) then we get the full benefit of hardware acceleration, as our results confirm. Open nested transactions can run in hardware even when their parent is in software because open nesting supports full undo of the abstract operations; closed nested cannot because there is no way to undo their commits if the parent fails.

3.3.4 Run-Time library

The run-time library provides the dynamic support needed for transactional execution. It supports both closed and open nested transactions, running under HTM or STM simultaneously, as well as boosting. Thus a program can make use of all styles of transactional execution. Our experiments also configure the run-time library for modes of execution that support only one of closed, or boosted transactions, so as to isolate the overheads for each mode. For example, the data structures needed for tracking the reads and writes of open/-closed nested transactions are not needed for boosting and there is no need to instantiate them in that case.

The run-time library offers both HTM and STM versions of all important methods. As previously explained, the HTM version of the method takes an additional **`boolean`** argument indicating whether it should run in "software mode." The run-time library also maintains statistics in a thread-local manner, avoiding false conflicts in the statistics collection process.

The library performs conflict detection at the level of objects, and tracks writes at the level of fields using an undo log. Each transactionalized object carries an extra field, which holds the lock for writes, and otherwise contains a version number for the object, which is incremented upon commit. In our implementation HTM and STM can safely co-exist simultaneously. Thus the two mechanisms need to play well with each other. In general, we adopt pessimistic concurrency control for writes, and optimistic concurrency control for reads. When running under STM, writes acquire a lock on the object. Reads proceed optimistically, simply logging the value of this field (a version number), and the log is then processed at commit time to validate the transaction (if the logged version number does not match the current value and the owner of a locked object is not the current transaction (or an ancestor) then the transaction aborts). When running under HTM, writes simply increment the version number, thus invalidating conflicting STM readers and conflicting with HTM readers or writers. Reads under HTM perform a check to make sure that the object is not locked by a non-ancestor transaction, explicitly aborting if necessary. In sum, the lock/version word "glues" together the STM and HTM schemes into a coherent (and safe!) hybrid TM.

The implementation of `PointSpace` that we use in our experiments itself requires a concurrent data structure to store the lock metadata because multiple transactions can try to acquire abstract locks concurrently. We use the `NonBlockingFriendlyHashMap` of Crain et al. [2] for this purpose.

4. Benchmarks

Our workloads extend Synchrobench [11], which is a micro-benchmark suite for evaluating synchronization techniques on collection classes such as sets and maps. It provides implementations for a variety of differently synchronized data structures in Java (as well as C/C++). It defines abstract APIs comprising simple `add`, `remove`, `contains`, and `get` operations that the data structures must implement. Adding new implementations to the framework is simply

[5] Doing so is possible, but would mean the HTM version does all the work of the STM version, with the added overhead of starting and committing an HTM transaction.

a matter of making them conform to one of these APIs. The CompositionalIntSet interface abstracts sets, while CompositionalMap abstracts maps.

We extend Synchrobench for use with nested transactions in several ways. First, we provide open atomic, closed atomic, and boosted implementations of the CompositionalIntSet and CompositionalMap interfaces in our language dialect. These classes are compiled by our modified compiler. We also augment the Synchrobench driver to instantiate these implementations for measurement. Second, we reconfigure the driver to run *transactions* of various sizes, consisting of aggregate operations on the underlying data structures. This enables benchmarking for throughput while varying transaction granularity. Third, we reconfigure the driver to offer the ability to *pin* worker threads to specific cores. Finally, we make refinements to the manner in which the driver calculates throughput numbers. We now describe these modifications in more detail.

4.1 Open Atomic Workload

Listing 8 shows the OpenIntSet class, an open atomic implementation of CompositionalIntSet. OpenIntSet provides a concurrency-safe wrapper for unsynchronized implementations of CompositionalIntSet. Similarly, OpenMap provides a concurrency-safe wrapper for unsynchronized CompositionalMap implementations. Here we give more precise details of the implementation of OpenIntSet; OpenMap is derived similarly.

As in the earlier OrderedSet example, OpenIntSet defines two lock spaces: eltSpace manages abstract locks issued on points corresponding to elements in the set, and setSpace defines abstract locks for the set as a whole. The addInt method attempts to add the element elt to the set. Thus it needs an X lock on the point represented by element elt from the eltSpace lock space, and a C lock for the set as a whole from the setSpace lock space.

Generally, **onabort** handlers are needed only for methods that change the abstract state of the set. One such method is addInt, which returns **true** if the element was added to the set and **false** if the element was already present. Thus its **onabort** handler must remove the element from the set only if it was not previously there. To achieve this, the **onabort** clause captures and uses the result of the committed body of the method. The other methods can be derived similarly. Our extended transactional Java syntax supports declarations for variables (like result) outside the body of the open atomic method that are visible to the body and the **onabort** clause.

4.2 Closed Atomic Workload

The ClosedIntSet class shown in Listing 9 provides a concurrency-safe wrapper, using closed nesting, for an unsynchronized CompositionalIntSet. The methods of ClosedIntSet execute the set operations (closed) nested mode.

```
public openatomic class OpenIntSet
    implements CompositionalIntSet {
  private final CompositionalIntSet intSet;
  private final
    LockSpace
      <SXMode,PointSpace<SXMode,Integer>>
    eltSpace
      = new PointSpace<SXMode,Integer>();
  private final
    LockSpace
      <PCMode,UnitSpace<PCMode,OpenIntSet>>
    setSpace
      = new UnitSpace<PCMode,OpenIntSet>();
  public
    OpenIntSet(CompositionalIntSet intSet)
    { this.intSet = intSet; }
  public boolean addInt (int elt)
    [boolean result = false]
    locking
      (eltSpace : point(elt) : SXMode.X),
      (setSpace : get() : PCMode.C)
    { return (result = intSet.addInt(elt)); }
    onabort
    { if (result) intSet.removeInt(elt); }
  // etc.
}
```

Listing 8. OpenIntSet class

```
public xatomic class ClosedIntSet
    implements CompositionalIntSet {
  private final CompositionalIntSet intSet;
  public ClosedIntSet(CompositionalIntSet
      intSet)
    { this.intSet = intSet; }
  public xatomic boolean addInt(int x)
    { return intSet.addInt(x); }
  // etc.
}
```

Listing 9. ClosedIntSet class

4.3 Boosted Workload

Boosted and open atomic classes look similar since they both must make use of abstract locks to protect the abstract state of the underlying data structure. Listing 10 shows BoostedMap as an implementation of the CompositionalMap interface. Unlike an open atomic class, a boosted class wraps a thread-safe implementation of the CompositionalMap interface. This is an important distinction.

4.4 Support for Varying Transaction Sizes

We extend the driver for Synchrobench to aggregate some number of underlying data structure operations nested within

```
public boostedatomic class BoostedMap<K,V>
    implements CompositionalMap<K,V> {
  private final ConcurrentMap<K,V> map;
  private final
    LockSpace<SXMode,PointSpace<SXMode,K>>
    keySpace = new PointSpace<SXMode, K>();
  private final
    LockSpace
      <PCMode,UnitSpace<PCMode,
          BoostedMap<K,V>>>
    mapSpace
      = new UnitSpace<PCMode,
          BoostedMap<K,V>>();
  public BoostedMap(ConcurrentMap<K,V> map)
    { this.map = map; }
  public V put(K key, V val)
    [V result]
    locking
      (keySpace : point(key) : SXMode.X),
      (mapSpace : get() : PCMode.C)
    { return (result = map.put(key, val)); }
    onabort {
      if (result == null) map.remove(key);
      else map.put(key, result);
    }
  // etc.
}
```

Listing 10. BoostedMap class

```
private xatomic void atomicDoOperation() {
  for (int i = 0;
       i < Parameters.groupSize;
       i++)
    doOperation();
}
```

Listing 11. Top-level transaction for nesting

a top-level closed transaction, parameterized by a new run-time flag g. We modified the worker threads of Synchrobench accordingly as shown in Listing 11. If the parameter g has a value greater than 0 then the operations are performed within a top-level closed transaction by marking atomicDoOperation as **xatomic**. Then doOperation will be nested/boosted accordingly within the top-level transaction. We also compare against Deuce [17], for which we use the corresponding method shown in Listing 12, to achieve the same effect.

4.5 Support for Thread Pinning

We update the driver for Synchrobench to accommodate the option of specifying a strategy for pinning worker threads. The new run-time flag ps can be used to specify this strategy.

```
@Atomic(metainf = "elastic")
private void deuceAtomicDoOperation() {
  for (int i = 0;
       i < Parameters.groupSize;
       i++)
    doOperation();
}
```

Listing 12. Top-level transaction for Deuce

```
@Atomic // API method
public boolean addInt(int x) ...
// Methods used by the maintenance thread
@Atomic(metainf = "maint")
private Node getChild(Node n, boolean left) ...
```

Listing 13. Deuce STM implementation of TFTreeSet

The value accepted is any combination of the characters C, S, and H. The character C represents core, S represents socket, while H represents hyperthread. These characters represent the 3 different dimensions that can be varied when pinning threads. The sequence of the characters specifies which aspect of these to vary most rapidly when pinning threads. For example, CSH means to vary the core first, then the socket, and finally hyperthreads of the same core.

4.6 Modified Transaction Friendly Data Structure

Synchrobench [11] contains transaction-friendly data structures that are "speculation-friendly" [3]. We took the transaction-friendly TreeSet binary search tree implementation and modified it to run with transactions. We refer to this as TFTreeSet. It uses a separate maintenance thread to keep the data structure properly balanced. Inserts are done at the leaf level, while deleting an element simply marks the node as deleted. The maintenance thread rebalances the data structure and removes deleted nodes. In the implementation for Deuce the maintenance thread performs its tasks inside small atomic methods as shown in Listing 13. The API methods are also marked as atomic methods.

Adapting these data structures for our transactional Java dialect is trivial. We mark those methods used by the maintenance thread as closed atomic using the **xatomic** method modifier as shown in Listing 14. This is reasonable because the maintenance methods are short, making only a quick modification. We do not include anything special on the API methods, but leave it to our open/closed wrapper classes to enforce atomicity. Hence, depending on the wrapper that is instantiated, the API methods may run closed or open.

We also performed some hand optimizations to the benchmark code that are important in the transactional setting. These optimizations could be performed by a bytecode rewriting optimizer, a task we leave to future work. Specifically, we found places where a field is often unconditionally updated

```
// API method
public boolean addInt(int x) ...
// Methods used by the maintenance thread
private xatomic
Node getChild(Node n, boolean left) ...
```

Listing 14. Transactional implementation of `TFTreeSet`

with the value it already contains. Such writes are cheap in the non-transactional case, but introduce needless conflicts in transactions. We made them conditional. We also specially mark `openForRead` and `openForWrite` calls that are redundant and `openForRead` calls that are always followed by an `openForWrite` on the same object. This substantially reduces the transactional instrumentation in the micro-benchmarks.

4.7 Modified Throughput Reporting for Accuracy

Previously, the Synchrobench driver thread worked as follows. It created all the worker threads, then recorded the system time, and finally started the worker threads individually. The main thread then slept for the duration of the benchmark. Upon being woken up, the main thread attempted to join all the worker threads and to record the system time again. The difference between the recorded system times is taken as the elapsed time for the benchmark iteration. Meanwhile each thread kept a record of the number of operations it executed. When reporting the results, Synchrobench divided the total number of operations completed by all the threads and divided by the elapsed time to calculate the throughput in units of operations per second. This mechanism works relatively well when running with a small number of threads, but when running on a multi-socket machine some flaws appeared. We noticed that the elapsed time when running with 48 threads was in the range of 5.5 seconds when the specified duration was 5 seconds. This had to do with the difference in the times at which each thread started (they are started one by one), and even more in the times when they stopped (after each operation, they look to see if their "stop" flag has been set; operation times vary as do the times when the "stop" flags are actually set). Thus some threads are actually idle for significant periods of time leading to an underestimate of throughput. Our remedy is to record the start and stop time of the individual worker threads. We divide the total number of operations completed by the total of the running times of the worker threads, and then multiply by the number of threads. This throughput value more accurately represents average throughput for large numbers of workers.

5. Experiments

Our experiments explore a range of structured transactions, namely flat, closed, open, and boosted, in STM-only mode and in self-tuning hybrid HTM/STM mode. We further compare against Deuce STM, running its efficient elastic mode transactions and configured as described in Section 4, as a reference point. We conducted all experiments using the extended version of Synchrobench described in Section 4 with the parameters shown in Table 1.

$i = 16K\|64K$	The initial number of elements added to the data structure before measurement begins.
$r = 32K\|128K$	The range of possible keys used in the data structure; keys are drawn uniformly at random.
$u = 0\|5\|50$	The percent of operations that are updates, each randomly chosen either to add or remove an element.
$n = 5$	The number of iterations of the benchmark.
$t = 1\|2\|4\|8...44\|48$	The number of spawned worker threads.
$W = 5$	The warm up time in seconds that the benchmark runs before starting measurement.
$d = 5000$	The duration of a single iteration of the benchmark in milliseconds.
$g = 1\|2\|4\|8\|16\|32$	The number of operations to perform in each transaction.
$ps = CSH$	The pinning strategy to use. We first pin threads to different cores on one socket, then on the next socket, before finally assigning threads to different hyperthreads of the same core. Exploratory experiments showed this strategy to be clearly the best.

Table 1. Synchrobench parameters for experiments

We perform three sets of runs across these parameters so as to space the sets of five iterations over time. Thus, we sample 15 measurements for each configuration.

Given a benchmark data structure, Synchrobench initializes the data structure to its initial size, drawing randomly from the indicated range of values. Once the data structure is initialized, Synchrobench performs operations at random, using the update percentage to decide if the operation should be "add/remove" or "get/contains". The collected statistics are cleared once the warm up period ends, and the benchmark runs for the specified duration after that. Then Synchrobench reports statistics for the benchmark run, including the throughput (operations/s).

When enabling HTM, we followed a more complex warm-up procedure. First, we ran for five seconds calling the HTM routing methods of transactions. Then we paused five seconds to allow the HotSpot compiler to compile (and possibly optimize) methods. We repeated this procedure to force proper linking of the resulting compiled methods. We then forced garbage collection (so that collections will not interfere with our timings) and started Synchrobench's warm-up run.

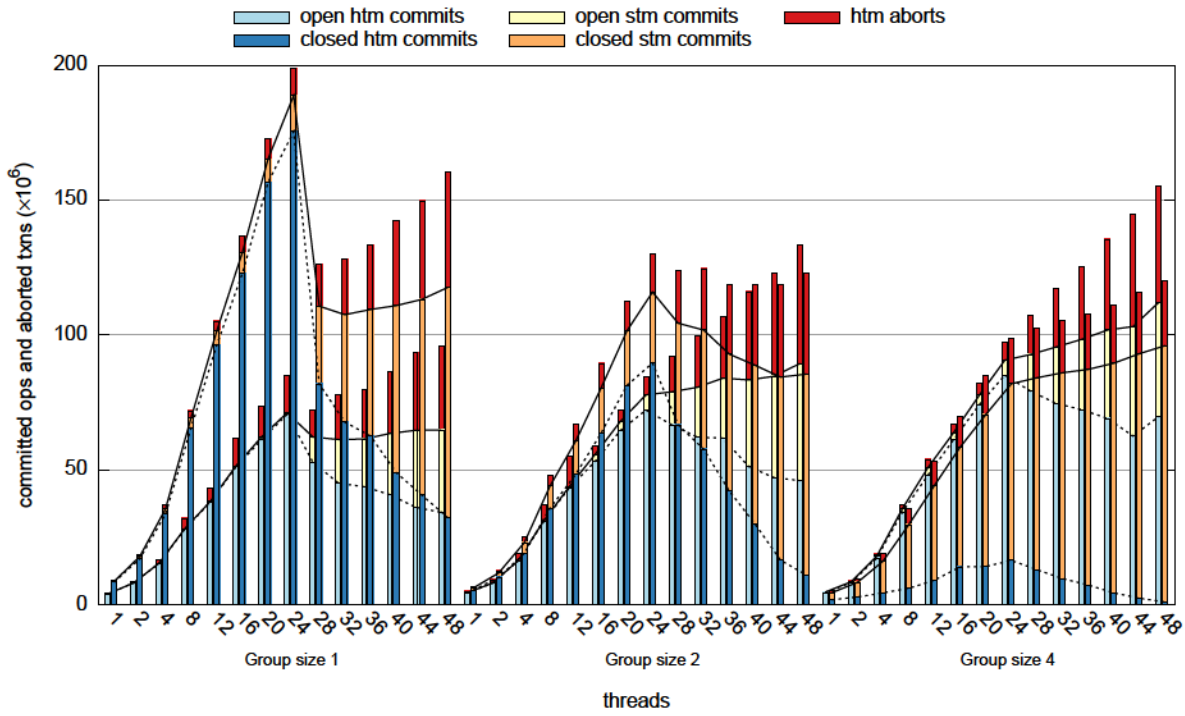


Figure 2. Committed operations versus aborts

We believe that in the future this warming up approach can be generalized for arbitrary transactional applications. Alternatively, deeper modifications could be made to the compilers to make them HTM aware.

The `OpenIntSet` and `ClosedIntSet` classes are initialized with the transactionalized version of `TFTreeSet`. For boosting, `BoostedMap` is initialized with `NonBlockingTorontoBSTMap` [10]. For benchmarks involving Deuce STM we run `TFTreeSet` under Deuce STM.

All benchmarks were run on a 48-way, x86-64 Intel Xeon E5-2690 v3 machine with 2 sockets of 12 hyperthreaded cores, with the clock frequency fixed to 2.4 GHz, and with TSX enabled. The machine was running CentOS Linux release 7.2.1511 and our modified version of OpenJDK.

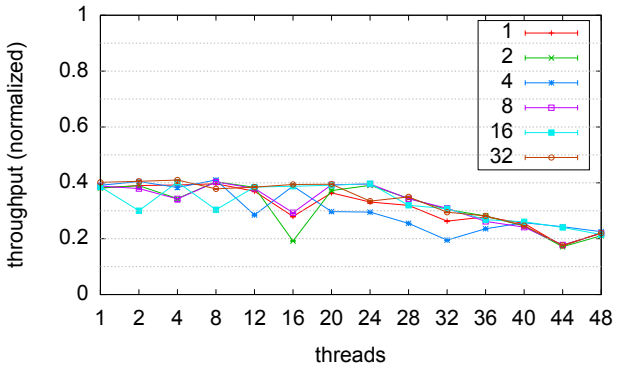
6. Results

We now present results for executing the workload under different transaction implementations. Our first set of results are for data structures initialized with 64K elements and a key range of 128K. All numbers reported in throughput graphs are normalized *per-thread* throughput. This implies that perfect scaling will appear as a horizontal line in the graphs. Our normalization is relative to the standard unsynchronized `java.util.TreeMap` (run with one thread, no synchronization). At each point we plot the median along with bars showing the 10th and 90th percentiles across the 15 total iterations we accumulated. A common theme in the results is that open nesting and boosting do not perform well when the transaction size is small. This is because these transaction forms carry a certain amount of overhead—prominent at transaction size 1, for example. Much of this overhead is in acquiring abstract locks. Also, for each nested operation, the inner transaction (which is open) needs to create an abort handler and log it. These costs become smaller in a relative sense as transaction size increases, giving these forms better performance and scaling at larger transaction sizes.

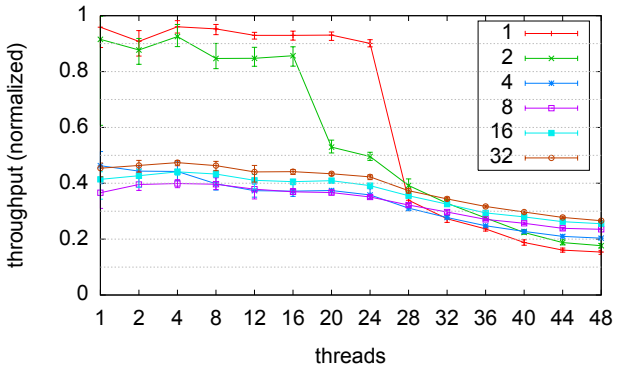
6.1 HTM versus STM

We first compare HTM and STM. Figure 2 shows three different transaction (group) sizes, 1, 2, and 4, from left to right. Within each group we have bars for each thread count (1, 2, ..., 48). The bars show the mean number of committed operations or hardware aborts per 5s benchmark iteration, breaking committed operations down by whether they ran in HTM or STM, with STM stacked on top of HTM. The left bar in each pair is for open nesting, the right bar for closed. Software aborts are so few as to be invisible in this graph. Finally, HTM abort counts are stacked on top (sometimes so few they are not visible). We connect HTM and total commits by lines, to help see the trends better. The bluer colors represent HTM, the yellower ones STM, and red represents aborts. These results are for update fraction 5%.

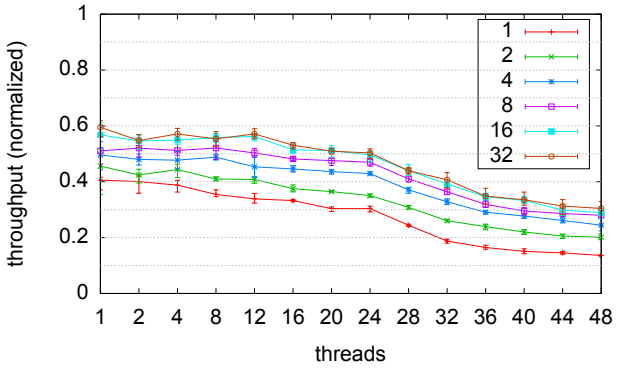
We find that open nesting performs relatively poorly due to the extra overhead of abstract locking and logging of undo operations, except at group size 4 where it outperforms closed nesting at all thread counts. This trend continues with higher



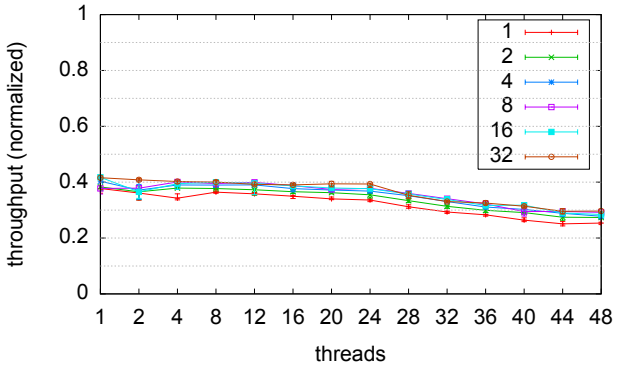
(a) Deuce (elastic)



(b) Closed

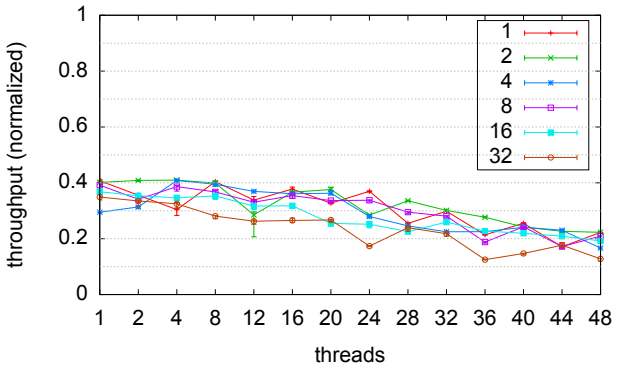


(c) Open

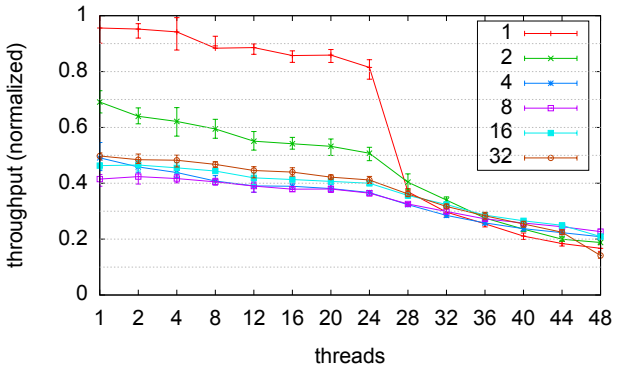


(d) Boosted (wraps NonBlockingTorontoBSTMap)

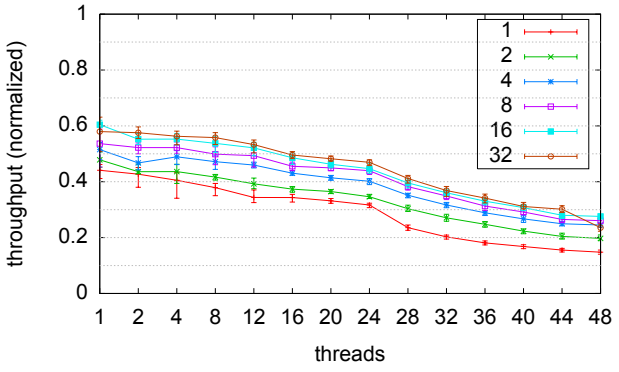
Figure 3. 0% updates (read-only), varying g



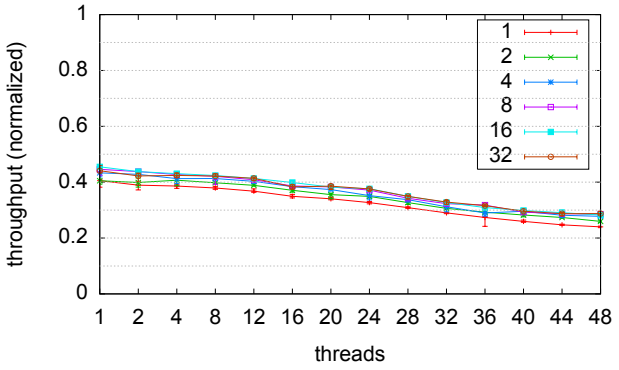
(a) Deuce (elastic)



(b) Closed

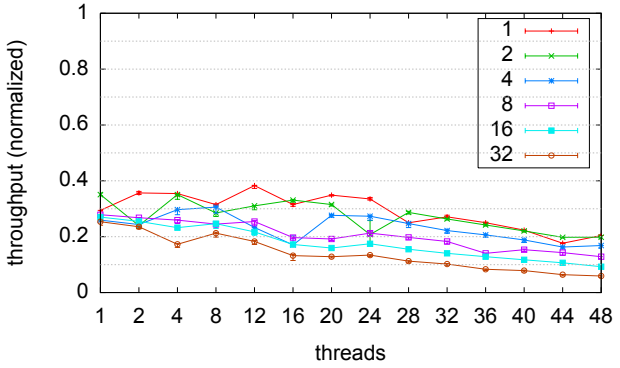


(c) Open

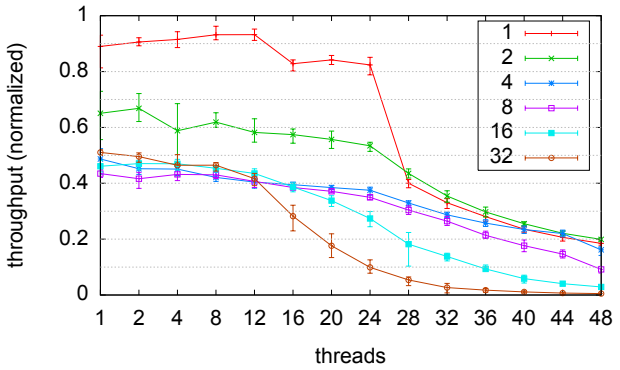


(d) Boosted (wraps NonBlockingTorontoBSTMap)

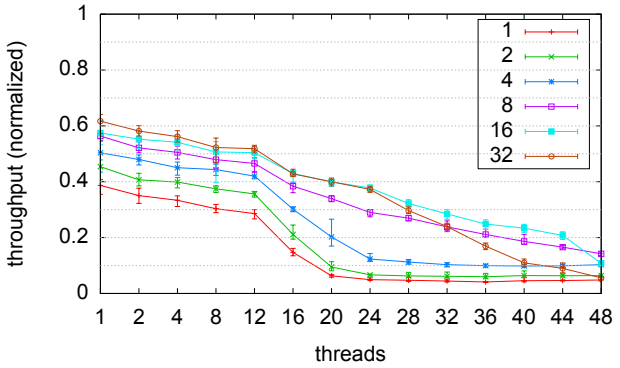
Figure 4. 5% updates, varying g



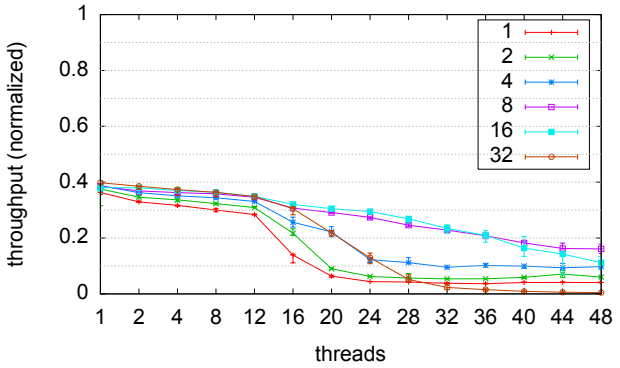
(a) Deuce (elastic)



(b) Closed

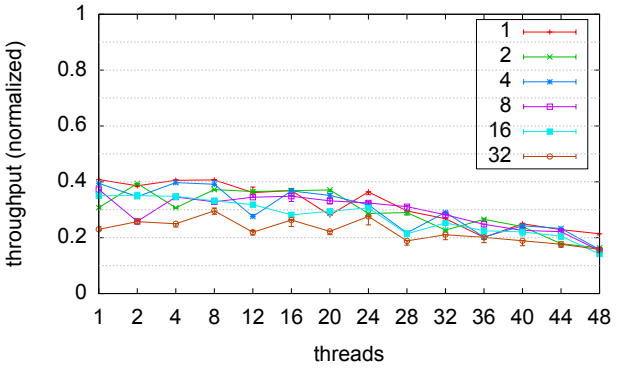


(c) Open

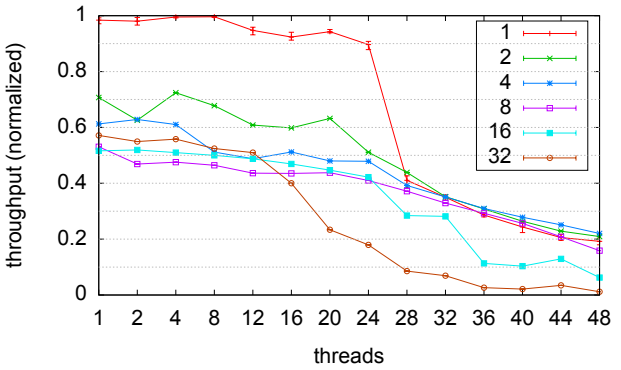


(d) Boosted (wraps NonBlockingTorontoBSTMap)

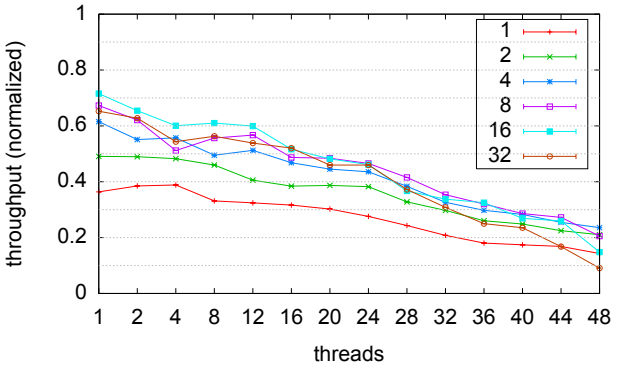
Figure 5. 50% updates, varying g



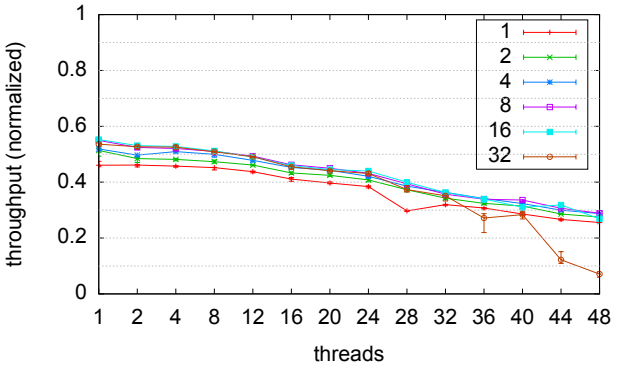
(a) Deuce (elastic)



(b) Closed



(c) Open



(d) Boosted (wraps NonBlockingTorontoBSTMap)

Figure 6. 5% updates, varying g for tree size of 16K

group sizes (not shown). For thread counts beyond 24, threads start to share the same core (hyperthreading), which results in poorer performance, especially for HTM since a core's hyperthreads share L1 and L2 caches, which are used as the transactional buffer by TSX. This is exhibited by the drop in HTM commits and increase in HTM aborts. We also see that closed HTM falls away quickly as we increase the group size. Closed HTM largely fails beyond group size of 4. This is more because of transaction footprint exceeding the buffer than because of increasing conflicts. However, open HTM is strong in group size 4 and beyond. This is because the top-level transactions here are in software and each HTM transaction handles just one operation. This keeps the HTM footprint small while amortizing the open nesting overheads. Even with open nesting we see a relative increase in STM versus HTM beyond 24 threads, as a result of hyperthreading. The overall shape of the graphs for other update percentages are similar to these, and hence we do not show them.

A theme here that we will see in other results as well is that there are portions of the parameter space where HTM works well and offers substantial speed up over STM (even with our hand optimization of STM). Likewise, there are portions of the space where open nesting works better than closed nesting, despite its higher overheads.

6.2 Closed, Open, and Boosted

Figures 3 to 5 show normalized throughput for update fractions 0%, 5%, and 50%, respectively. Each figure includes four graphs, showing performance for Deuce [17] (running its efficient elastic mode transactions), closed nesting, open nesting, and boosting. We include Deuce since it demonstrates that our system lies in the same general performance range as this mature system. We see that closed nesting does better than Deuce at small thread counts and the same or not quite as well at large thread counts. We also see that for smaller thread counts and group sizes 1 and 2, closed nesting achieves particularly good performance. This is because those cases run in HTM much of the time. We compared open and closed nesting above and these graphs are consistent with that analysis. Boosting is interesting to compare with open nesting since a boosted data structure is hand crafted to offer good throughput for individual operations, and our wrappers implement the same abstract locking and undo logging for both boosting and open nesting. Being hand-crafted, we expected boosting to do better, but not surprisingly open nesting tends to win up to 12 threads where HTM remains effective.

6.3 Smaller Data Structure Size

Figure 6 shows the impact due to increased chance of conflicts when using a smaller data structure, with 16K entries instead of 64K, key range of 32K, and update fraction 5%. For the same update fraction this smaller tree size results in more conflicts (both physical and abstract) than for larger trees, and the graphs clearly show how performance drops off with increasing group size since more transactions will conflict.

7. Conclusions

Our results demonstrate the utility of nesting as a means to achieving reliably scalable concurrent manipulation of data structures using open/closed nesting, without the need for hand-tuned and hand-coded non-blocking implementations. So long as the underlying data structure is friendly to transactions it can easily be nested.

Moreover, we demonstrate that HTM mechanisms can be exploited effectively to accelerate nested transaction schemes, while allowing software-only schemes to run safely alongside the HTM-accelerated executions.

Our results indicate the degree to which hyperthreading degrades performance of HTM schemes due to the need to share capacity between hyperthreads on the same core.

We also demonstrate the performance envelopes for each of the schemes, showing that there is a space in the workload spectrum where each is superior. As such, programmers must choose carefully which technique to employ, depending on the nature of their programs.

For programmers willing to wrap bespoke linearizable data structures, boosting works well at high thread counts where HTM degrades, because it does not pay the performance penalty of STM.

We have also shown how to integrate HTM features into OpenJDK such that the compilers can inline the HTM operations as intrinsics. In future work we plan to convince the Hotspot compilers to warm up more effectively and optimize the HTM code.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCF-1408896, CCF-1409284, CNS-1405939, CNS-1161237, and CNS-1162246.

References

[1] K. Chapman, A. L. Hosking, J. E. B. Moss, and T. Richards. Closed and open nested atomic actions for Java: Language design and prototype implementation. In *International Conference on Principles and Practice of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 169–180, Cracow, Poland, Oct. 2014. doi: 10.1145/2647508.2647525.

[2] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly methodology for search structures. Research report, INRIA, Feb. 2012. URL https://hal.inria.fr/hal-00668010.

[3] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–170, New Orleans, Louisiana, Feb. 2012. doi: 10.1145/2145816.2145837.

[4] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages

39–52, Newport Beach, California, Mar. 2011. doi: 10.1145/1950365.1950373.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, San Jose, California, Oct. 2006. doi: 10.1145/1168857.1168900.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, Washington, DC, Mar. 2009. doi: 10.1145/1508244.1508263.

[7] D. Dice, A. Kogan, and Y. Lev. Refined transactional lock elision. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19:1–19:12, Barcelona, Spain, Mar. 2016. doi: 10.1145/2851141.2851162.

[8] N. Diegues and P. Romano. Self-tuning Intel transactional synchronization extensions. In *USENIX International Conference on Autonomic Computing*, pages 209–219, Philadelphia, PA, June 2014. URL https://www.usenix.org/conference/icac14/technical-sessions/presentation/diegues.

[9] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Aug. 2014. doi: 10.1145/2628071.2628080.

[10] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–140, Zürich, Switzerland, July 2010. doi: 10.1145/1835698.1835736.

[11] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, San Francisco, California, Feb. 2015. doi: 10.1145/2688500.2688501.

[12] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Salt Lake City, Utah, Feb. 2008. doi: 10.1145/1345206.1345237.

[13] M. P. Herlihy and J. M. Wing. Linearizability: A correctness criterion for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, July 1990. doi: 10.1145/78969.78972.

[14] Intel. *Intel Transactional Synchronization Extensions (Intel TSX) Programming Considerations*. URL https://software.intel.com/en-us/node/582935.

[15] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM system Z. In *International Symposium on Microarchitecture*, pages 25–36, Dec. 2012. doi: 10.1109/MICRO.2012.12.

[16] A. Kasko, S. Kobylyanskiy, and A. Mironchenko. *OpenJDK Cookbook*. Packt Publishing, Jan. 2015. ISBN 1849698406.

[17] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Workshop on Programmability Issues for Heterogeneous Multicores*, Pisa, Italy, Jan. 2010. URL http://www.velox-project.eu/sites/default/files/multiprog10.pdf.

[18] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 209–220, Mar. 2006. doi: 10.1145/1122971.1123003.

[19] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Aug. 2007. URL https://www.cs.rochester.edu/meetings/TRANSACT07/papers/lev.pdf.

[20] A. Matveev and N. Shavit. Reduced hardware NORec: A safe and scalable hybrid transactional memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 59–71, Istanbul, Turkey, Mar. 2015. doi: 10.1145/2694344.2694393.

[21] J. E. B. Moss. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.

[22] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63:186–201, Dec. 2006. doi: 10.1016/j.scico.2006.05.010.

[23] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–78, San Jose, California, Mar. 2007. doi: 10.1145/1229428.1229442.

[24] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, Monterey, California, Apr. 2001. URL https://www.usenix.org/legacy/events/jvm01/full_papers/paleczny/paleczny.pdf.

[25] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *International Symposium on Microarchitecture*, pages 294–305, Austin, Texas, Dec. 2001. doi: 10.1109/MICRO.2001.991127.

[26] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 53–64, San Jose, California, June 2011. doi: 10.1145/1989493.1989501.

[27] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 19:1–19:11, Denver, Colorado, Nov. 2013. doi: 10.1145/2503210.2503232.