Open Nesting in Software Transactional Memory

Yang Ni   Vijay Menon   Ali-Reza Adl-Tabatabai   Antony L. Hosking[1]

Richard L. Hudson   J. Eliot B. Moss[2]   Bratin Saha   Tatiana Shpeisman

Programming Systems Lab	[1]Dept. of Computer Science	[2]Dept. of Computer Science
Intel Corporation	Purdue University	University of Massachusetts
Santa Clara, CA 95054	West Lafayette, IN 47907	Amherst, MA 01003

{yang.ni,vijay.s.menon,ali-reza.adl-tabatabai,rick.hudson,bratin.saha,tatiana.shpeisman}@intel.com, hosking@cs.purdue.edu, moss@cs.umass.edu

Abstract

Transactional memory (TM) promises to simplify concurrent programming while providing scalability competitive to fine-grained locking. Language-based constructs allow programmers to denote atomic regions declaratively and to rely on the underlying system to provide transactional guarantees along with concurrency. In contrast with fine-grained locking, TM allows programmers to write simpler programs that are composable and deadlock-free.

TM implementations operate by tracking loads and stores to memory and by detecting concurrent conflicting accesses by different transactions. By automating this process, they greatly reduce the programmer's burden, but they also are forced to be conservative. In certain cases, conflicting memory accesses may not actually violate the higher-level semantics of a program, and a programmer may wish to allow seemingly conflicting transactions to execute concurrently.

Open nested transactions enable expert programmers to differentiate between physical conflicts, at the level of memory, and logical conflicts that actually violate application semantics. A TM system with open nesting can permit physical conflicts that are not logical conflicts, and thus increase concurrency among application threads.

Here we present an implementation of open nested transactions in a Java-based software transactional memory (STM) system. We describe new language constructs to support open nesting in Java, and we discuss new abstract locking mechanisms that a programmer can use to prevent logical conflicts. We demonstrate how these constructs can be mapped efficiently to existing STM data structures. Finally, we evaluate our system on a set of Java applications and data structures, demonstrating how open nesting can enhance application scalability.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Concurrent programming structures

General Terms Design, Languages, Performance

Keywords transactional memory, nested transactions, open nesting, abstract locks

1. Introduction

With the recent advent of multi-core architectures, the need for better concurrent programming methodologies has intensified. In particular, careful reasoning about shared data is crucial to both scalability and reliability of concurrent programs. Transactional memory (TM) simplifies concurrent programming by providing the programmer with a simpler, more sequential, semantic universe in which to reason and program. Recent work has shown that TM implementations can match the scalability of fine-grained synchronization while retaining the program-level simplicity of coarse-grained locks. Moreover, unlike locks, transactions are composable and deadlock-free, significantly reducing programmer burden.

The ideas behind TM borrow greatly from the wide body of literature on transaction processing, mostly in the context of databases. In particular, memory transactions have been designed to respect the first three fundamental ACID properties: atomicity, consistency, and isolation. One important idea in the database context is that of *open nested transactions*, where a programmer can exploit higher-level semantics to enhance concurrency. Moss and Hosking have recently suggested that open nested transactions can be used in the context of transactional memory as well [13, 14, 15]. To understand the benefits of open nesting, we must distinguish *physical* serializability and *abstract* serializability. We call a concurrent execution of a set of transactions *physically serializable* if the resulting *state of memory* is consistent with some serial execution of those transactions. On the other hand, we call a concurrent execution of a set of transactions *abstractly serializable* if the resulting *abstract view of data* is consistent with some serial execution of those transactions.

We illustrate the difference in the example of Figure 1, which uses a set s that is implemented as a linked list. Suppose that s is initially empty and that it always inserts a new element at the end of its internal linked list. Then, physically serializable executions of these transactions must result in either the order (x, y, z) or the order (z, x, y). In particular, (x, z, y) is not a valid result as it would violate the atomicity of Transaction 1.

However, the programmer may realize that list ordering is not relevant to the abstract state of the set. Set membership tests, etc., will produce the same result regardless of the order of the elements on the internal linked list. Thus, the order (x, z, y) is consistent with

Transaction 1:	Transaction 2:

```
atomic {                    atomic {
  s.insert(x);                s.insert(z);
  s.insert(y);              }
}
```

Figure 1. Example 1: Physical versus abstract serializability

Transaction 1:

```
atomic {
  s.insert(x);
  if (!s.contains(z))
    s.insert(y);
}
```

Transaction 2:

```
atomic {
  s.insert(w);
  if (!s.contains(y))
    s.insert(z);
}
```

Figure 2. Example 2: Physical versus abstract serializability

an *abstractly* serializable execution: both transactions successfully inserted their elements into the set. By permitting such a result, a programmer can increase concurrency. With abstract serializability, if Transaction 1 stalls between the inserts of x and y, Transaction 2 may still proceed to insert z.

It's important to note that abstract serializability still demands some form of conflict detection and resolution. We cannot simply execute the two transactions in the above example concurrently with *no* conflict management. First, we must ensure that the set insertions themselves happen atomically *individually*. For example, a concurrent execution that results in (x, z) is neither physically nor abstractly serializable. Second, we must ensure that *abstract conflicts* do not occur. Consider the slightly more complex example in Figure 2. In this case, assuming s is initially empty and our set-based notion of abstract serializability, the only two valid execution results are {w, x, y} and {w, x, z} (where the sets are unordered). If Transaction 1 observes that s does not contain z, concurrently executing Transaction 2 must not be allowed to insert z without properly serializing with Transaction 1. Transaction 2 must either wait for Transaction 1 to complete, or one of the transactions must abort.

We use open nested transactions to support abstract serializability. In the examples above, the set operations can be implemented as open nested transactions operating at a lower-level of abstraction. In this case, the individual set operations can themselves be physically serializable. However, once they complete, they can release memory resources to enhance concurrency. To provide conflict detection at the abstract level, we integrate an *abstract locking* mechanism into our design. In the preceding example, Transaction 1 should use an abstract lock to establish the fact that z is not in the set. Transaction 2 will later try to obtain an abstract lock to permit inserting z into the set, but this lock conflicts with the one held by Transaction 1, preventing Transaction 2 from inserting z. If both transactions get past their if statement, then at least one of them will have to abort, since each conflicts with the other.

To abort a transaction properly, we allow the programmer to provide an abstract *compensating action* for each of its open nested transactions. In the case at hand, it is not sufficient merely to roll back the state of memory. Undoing the memory operations that Transaction 1 performed to insert x into the list may inadvertently also remove w if w was appended after x. Instead, an abstract compensating action must be performed. In this case, to abort the first transaction, we must logically remove x from the list, e.g., by executing s.remove(x).

The examples above highlight the potential scalability shortcomings of TM systems that support only physical serializability. Without transactions, the programmer has the burden of expressing with locks all possible conflicts in a system, but also has the freedom of determining that certain actions do not conflict whether it be for physical or logical reasons. A TM implementation based upon physical serializability significantly reduces the burden on the programmer by automating conflict detection, but is forced to be conservative as the example above shows.

Our goal is to offer open nested transactions as a means for programmers to obtain the precision of locks while retaining the benefits of transactional memory including serializability, composability, and deadlock-freedom. Some of this power will come at the cost of increased complexity since a system cannot automatically reason at

the level of application semantics. Nevertheless, we believe that, for expert programmers, the scalability benefits of open nesting can outweigh the costs. Open nesting is for expert library programmers to build software components that average programmers can compose together in a scalable way using transactions. Average programmers can use these software components without being aware of open nesting, so they still benefit from the productivity advantage of transactions but also get good performance.

Most existing TM implementations provide no open nesting mechanism. Two very recent notable exceptions are TCC [2, 10] and LogTM [11], both hardware TM implementations that support open and closed nesting. In both cases, however, the model of open nesting is lower-level than what we propose here.

The novel contributions of our paper are as follows:

- We present the first implementation of open nested transactions in an STM, and show how to map open nesting functionality efficiently onto existing STM data structures. Prior implementations of open nesting relied on hardware support [11, 2, 10]. Ours is a pure software implementation integrated into a high-performance STM and managed run-time.

- We propose new language constructs for open nesting that support two-phase abstract locking with hierarchical lock modes. We demonstrate how these constructs enable a straightforward implementation of scalable, open-nested variants of standard Java data structures. We also show how to integrate abstract locks into an STM implementation efficiently.

- We present the first evaluation of open nested data structures for transactional memory. Previous implementations studied open nesting in a more limited setting. Although they supported compensating actions, they evaluated open nesting only in a setting that did not require them.

- We demonstrate that open nested transactions enhance the scalability of concurrent data structures for long-running transactions without introducing significant overhead to short-running transactions in an STM.

- We demonstrate that open nested transactions also reduce transactional overhead for long-running software transactions on a single thread by reducing the memory requirements of STM validation.

- We discuss the potential pitfalls of open nesting, and we propose a set of guidelines the programmer and TM implementations should follow in order to exploit open nesting safely.

2. Language Extension and Semantics

We now define syntax and semantics for open nested transactions.[1] Because open nesting is a general mechanism that can violate physical serializability, it must be used with care. Our focus here is to provide the programmer with the tools to enforce abstract serializability, not general escape from serializability altogether. We discuss the potential dangers of badly formed open nested actions, and we propose programming conventions for using open nesting safely.

Atomic actions. We begin by summarizing the syntax and behavior of standard closed nested transactions as they are now widely accepted [14, 15]. We use the term *atomic action* to denote a unit of sequential computation that must execute atomically. Java syntax for transactions typically permits a method or block to be marked as atomic, by writing the keyword atomic where the keyword synchronized is permitted. A block/method may not be both

---

[1] We have developed a language proposal that is more detailed and includes a few more features, but space precludes detailed discussion in this paper. We leave broader and more detailed consideration of the language design issues to future publication.

`atomic` and `synchronized`. Each execution of an `atomic` block or method occurs as a transaction. An atomic action has three possible outcomes:

- It can *succeed* (complete successfully), in which case its effects are *committed*.

- It can *abort*, in which case its effects are *undone*, and the action will be *retried* from the beginning.

- It can *fail* (complete abruptly) when it throws an unhandled exception, in which case its effects are undone specially and the action is *not* retried.[2]

The *effects* of an atomic action include assignments to (shared) instance and class fields, and (unshared) local variables and formal method parameters and exception handler parameters (i.e., all declared variables), as well as the effects of nested atomic actions that it executes. With closed nesting, when a nested atomic action commits, it merges its read and write sets to those of the outer atomic action. None of its effects will be visible to other atomic actions (except its ancestors) until the top-level atomic action commits.

Open atomic actions. Just like a normal closed nested transaction, an open atomic action executes as an atomic action. However, when it succeeds its effects may be made visible to the world immediately. This may break the physical serializability of its enclosing transaction, while improving concurrency. To preserve *abstract* serializability while relaxing *physical* serializability, a nested open atomic action is augmented with *handlers* that execute (as separate open atomic actions) depending on whether the *enclosing* transaction succeeds, aborts, or fails. The handlers are registered with the enclosing transaction when the open atomic action succeeds (if it fails then its effects are discarded in the same way as for closed transactions). The handlers we support for an open atomic action include:

- *on-abort* to define compensation actions to undo the global effects of committed children when their parent fails.

- *on-commit* to define cleanup actions for committed children to be performed when their parent succeeds.

- *on-validation* to permit high-level checking (e.g., to validate optimistic abstract locks, to check integrity, or to check invariants when running in unreliable environments) for committed children when their parent is about to commit.

- *on-top-commit* to perform finalization actions for all committed descendants of a top-level transaction (e.g., to flush buffered output) when the top-level transaction commits.

Before an open atomic action can succeed, it usually must also acquire an associated set of abstract locks, each of which specifies locking some object on behalf of the open atomic action in some predefined lock mode. If a lock on object `o` in mode `m` is compatibly acquired in the context of the current atomic action then we say that the current atomic action *holds* a lock on `o` in mode `m` in the current context. Lock acquisition may fail due to lock conflict, when comparing triples consisting of lock context, lock object `o`, and lock mode `m`, in which case the current action aborts and will be retried.

Syntax for an open atomic action might be indicated by augmenting the standard `atomic` block/method syntax with handlers and a locks clause. However, because the purpose of open atomic actions is to permit programmers to construct *abstractions* that relax physical serializability while preserving abstract serializability, we prefer to express open actions for Java in terms of Java's principal abstraction mechanism, the *class*. Defining open atomic actions in terms of

---

[2] Our current implementation treats an exception as normal completion and commits the transaction, as does the design of Harris and Fraser [5]. Transaction behavior on exceptions is largely an orthogonal issue to open nesting.

```
MethodDeclaration:
      MethodHeader LocksClause_opt MethodBody
LocksClause:
      locks ( LockExpressions )
LockExpressions:
      LockExpression
      LockExpressions , LockExpression
LockExpression:
      Expression : Expression
MethodBody:
      Block OpenAtomicHandlers_opt
OpenAtomicHandlers:
      OpenAtomicHandler
      OpenAtomicHandlers OpenAtomicHandler
OpenAtomicHandler:
      onabort      Block
      oncommit     Block
      onvalidation Block
      ontopcommit  Block
```

Figure 3. Syntax for open atomic classes and methods

classes and their public methods promotes their use in construction of appropriate abstraction levels for concurrency.

Open atomic classes. An open atomic class is declared by the (new) modifier `openatomic`. This indicates that the (separately designated) `openatomic` instance or class fields of the class may be accessed only during execution of `openatomic` instance or class methods of the class. Subclasses of `openatomic` classes are (implicitly) `openatomic`. The `openatomic` modifier makes no sense for interfaces since they carry no implementation.

Open atomic fields. Open atomic classes may not have public fields, since there is no way to prevent public fields from being accessed outside the class. All (non-public) class or instance fields of an open atomic class are implicitly `openatomic`, and accessible only during the execution of `openatomic` instance or class methods of the class and its subclasses (this is enforced statically, though we omit details here). We provide additional syntax (not described here) for declaring specific non-public fields *not* to be `openatomic` where the programmer needs an escape from this default.

Open atomic methods. All public methods of an open atomic class are themselves `openatomic`, and may have any of the handlers enumerated above (`onabort`, `oncommit`, `onvalidation`, and `ontopcommit`), and optionally a `locks` clause, indicating the abstract locks the method must acquire before it can return.

2.1 Syntax

Java syntax for open atomic handlers and `locks` clauses is given in Figure 3. Handlers are within the scope of their method's header (i.e., as siblings to the scope of the method's block (if any). Thus, handlers can refer to `this` and instance fields (in an instance method), the method's parameters, and class fields. Except for the method's parameters, when the handler is executed the value of these variables is whatever they are at that time. The values of method parameters used in the handler are initialized to the value they had when the method completed. To simplify writing these handlers, we allow two special expression forms:

- `@result` is the value that the method originally returned

- `@old(exp)` is whatever value *exp* had at the time the method was entered (i.e., immediately *before* the method's `Block` was executed).

Supporting these constructs, and supporting use of method parameter values, requires generating code that saves the necessary values and makes them available to the handler if and when it is run.

The `locks` clause is attached to the method's header so as to make conflict properties more clearly apparent in the method declaration. The locks themselves must be acquired for the `openatomic` method to return (successfully). A *LockExpression* consists of a pair of expressions: the object to be locked `o` and the lock mode `m`.

2.2 Example: An open atomic `Map`

Figure 4 illustrates how an open atomic implementation of the `Map` interface can be defined as a wrapper for `Map` objects. In this example, we see that `OpenMap` is declared as an `openatomic` class implementing the `Map` interface. It wraps an unsynchronized `Map` object to permit safe concurrent access to the map, with `get`, `put`, `remove`, and `size` operations defined as `openatomic` methods.

In this example (and generally), `onabort` handlers are needed only for methods that mutate the state of the map. The `put` operation on `Map` objects returns the previous value associated with the given key in the map, or `null` if there was none. Thus, the `onabort` handler for `put` must either revert the map to contain that previous association if there was one, or simply remove the new association. Likewise, `remove` returns the previous value if any, so its `onabort` handler must restore that previous association.

This example makes use of three lock modes:[3] shared (`S`), exclusive (`X`), and intention exclusive (`IX`). The compatibility matrix for these locks is given in the following table:

	S	IX	X
S	yes	no	no
IX	no	yes	no
X	no	no	no

where the rows indicate the lock that is held on some object by one transaction and the columns indicate the lock that is requested on the same object by another transaction, and the entries say whether the lock can be granted. Thus, shared locks are compatible since multiple readers can operate on the same data item at the same time. On the other hand, one cannot write a data item while it still has readers; similarly, one cannot read a data item while it has a writer. Intention locks reveal (at a coarser granularity) that some writer is modifying some portion of a larger data item. Thus, to `put`/`remove` an association for some key in the map requires an intention lock on the map as a whole (as represented by the `lock` object). Two requests to `put`/`remove` associations for different keys do not conflict. However, to `put`/`remove` an association for any key does conflict with requests that read the state of the map as a whole, such as the `size` operation. These constraints are recorded for `put` and `remove` by acquisition of an exclusive lock on the key for which the association is being changed, to prevent others from changing that particular association, along with acquisition of an intention exclusive lock on the map as a whole (for `lock`) to prevent others from shared mode access to the whole map (such as `size` requires).

We need the separate `lock` object, rather than `this`, because equality on locked objects is defined in terms of Java `.equals` (see below), whereas in Java the return value of the method `equals` or `hashCode` depends on the associations defined by the map. The same map may have different `equals` or `hashCode` return values as its contents are changing, which in turn may cause a lock operation on `this` to succeed when it shouldn't. We need a constant `lock` object to uniquely identify each map (and also to avoid costly comparison of map contents using `equals` or `hashCode`).

---

[3] These are a subset of standard lock modes defined by Gray [4].

```
import java.util.*;
import java.openatomic.SXModes.*;
public openatomic class OpenMap implements Map {
  private final Map map;
  private final Object lock = new Object();
  public OpenMap(Map map) { this.map = map; }
  public Object get(Object key)
    locks(key:S)
    { return map.get(key); }
  public Object put(Object key, Object value)
    locks(key:X, lock:IX)
    { return map.put(key, value); }
    onabort {
      if (@result == null) map.remove(key);
      else map.put(key, @result);
    }
  public Object remove (Object key)
    locks(key:X, lock:IX)
    { return map.remove(key); }
    onabort {
      if (@result != null) map.put(key, @result);
    }
  public int size()
    locks(lock:S)
    { return map.size(); }
}
```

Figure 4. A concurrent map implemented as an open atomic class. We omit other methods of the `Map` interface.

2.3 Detailed Semantics

We describe the semantics of transaction execution, both closed and open, in terms of *abstract logs*. While these logs are similar to the logs one might use in an implementation (as we do), they are for descriptive purposes only. A new transaction begins with an empty log. If a transaction reads a memory location, it adds that fact to the log, and if it writes a memory location, it notes that fact plus the previous value of the memory location. Read and write entries are adequate for implementing flat transactions and closed nesting. In that setting, committing a closed nested action appends its log to its parent's log, and committing a top-level action discards the log. Aborting a transaction processes the log in reverse order, and for each write entry, restores the previous value. We define conflict in terms of logs, too: two actions conflict if neither action is an ancestor of the other and both attempt to access a location in conflicting modes (i.e., at least one of them tries to write). Conflicting actions must not both commit, and they must also never both write the same location (or undoing the writes may not work properly).

Open atomic actions add two kinds of things to the logs: handlers and abstract locks, and mandate more complicated committing and aborting procedures.

If there is no conflict, an open atomic action does the following to commit. First, it invokes any `onvalidation` handlers in its log, as open atomic actions, in the order in which they were logged. An `onvalidation` handler may force abort (retry), in which case we proceed to abort the action. If all the `onvalidation` handlers complete successfully, the next step is to invoke any logged `oncommit` handlers, as open atomic actions, in the order in which they were logged. After invoking the `oncommit` handlers, we might need to process the `ontopcommit` handlers. If we are committing a nested open action, we append the `ontopcommit` handlers, as well as the other handlers and abstract locks of the committing open action, to the end of the parent's log, and discard our own log. If we are committing a top-level transaction—in this case, which could be just atomic, rather than open atomic—we invoke all `ontopcommit` handlers, as open atomic actions, in the order in which they were logged.

In the presence of open actions, aborting transactions is also more complex. We first process the log of the aborting transaction in reverse order, processing two kinds of entries: writes and `onabort` handlers. To process a write entry, we restore the written location to its previous value; to process an `onabort` handler, we invoke it as an open atomic action. Notice that if the same location is written multiple times, with intervening open actions that have `onabort` handlers, we will roll back the same location more than once, generally with different values. The purpose here is to guarantee, as much as possible, that an `onabort` handler sees a memory state that is abstractly equivalent to the state at the end of its corresponding committed forward action.

Concerning abstract locks and concurrency control, an abstract lock consists of: a *context object* (or class), namely the object (or class) whose method was being run as an open action; a *locked object*, which is the object mentioned in the lock expression; and a *lock mode*. Lock modes come from a given lock mode class (we allow users to write their own lock mode classes, specifying which modes conflict with other modes for that lock mode class). Two abstract locks conflict if: they specify the same context object (in the sense of Java ==), the same locked object (in the sense of Java .equals), and conflicting modes of the same lock mode class. A lock acquiring action *A* conflicts with a lock holder *B* if *B* is not an ancestor of *A* and the lock that *A* is requesting conflicts with some lock in *B*'s log. We do not specify exactly when an open action attempts to acquire abstract locks—only some time during the execution of the action.

2.4 Using Open Nested Transactions Safely

We have defined open nesting to enable a programmer to relax constraints of physical serializability and, instead, rely on abstract serializability. In this manner, a programmer can exploit high-level application semantics to enhance the scalability of a program. This power, however, comes at a cost. In particular, the programmer must be aware of several issues specific to open nesting.

Correctly specify abstract locks and handlers. The first issue should be clear: a TM system that permits open nesting cannot guarantee the serializability of transactions. It relies on the programmer not to make mistakes. Without open nesting, the system can guarantee that an `atomic` block is serializable with respect to other `atomic` blocks. A TM system can reason about loads and stores. It can automatically detect conflicts among memory accesses and perform compensating actions. With open nesting and abstract serializability, the programmer must specify conflicts (via abstract locks) and provide compensation actions (via handlers). Moreover, if these are specified incorrectly, the application will behave in an unintended manner. For example, in Figure 4, the `put` operation's `onabort` handler must differentiate between a new insertion to a map and an update that replaces an existing value. If the `onabort` handler was incorrectly written only to remove new values, it would not properly compensate the `put` operation. For example, if the transaction in Figure 5 is forced to abort at a late point, the incorrect version of the handler would inadvertently remove the original entry. When the transaction is re-executed, it would fail to insert the new value.

Avoid deadlocks in on-abort handlers. The second issue is more subtle: open nested transactions can lead to deadlock. Consider the example in Figure 6. Suppose both transactions are concurrently executing their respective `onabort` handlers. In each case, the `onabort`

```
atomic {
  if(map.get(key) == value1)
    map.put(key,value2); // same key, new value
}
```

Figure 5. Serializability relies upon a correct `onabort` handler

```
Transaction 1:                Transaction 2:

atomic {//closed atomic       atomic {//closed atomic
  x = ...                       y = ...
  m1();                         m2();
}                             }
m1() {//open atomic           m2() {//open atomic
  // no write to y              // no write to x
} onabort {                   } onabort {
  y = ... // deadlock           x = ... // deadlock
}                             }
```

Figure 6. A potential deadlock in `onabort` handlers

handler conflicts with the other transaction. It is not sufficient merely to re-execute the handler. One of the top-level transactions must be aborted for either to make progress. However, we are already in the middle of aborting each transaction; otherwise, we would not be executing the `onabort` handlers in the first place! Each transaction must make progress to abort correctly, and both are blocked. As a result, we are deadlocked.

It seems that the problem here is that x and y are accessed sometimes open atomic and sometimes not. But this problem is actually more insidious than it appears. Imagine that we altered Transaction 2 in Figure 6 to write xx and yy instead. Even though these values are disjoint from those in Transaction 1, deadlock may still occur. A TM implementation will typically register conflicts between accesses to different memory addresses that map to the same underlying resources (e.g., cache lines in hardware TM or objects in software TM). If xx and x conflict and yy and y conflict, our altered program may deadlock even though the programmer sees no conflict at the application level. So there are also issues of granularity of access in addition to inside/outside of open actions.

The programmer can take the following guidelines to avoid a deadlock as in Figure 6.

1. The programmer must guarantee that, if a certain memory location in an object is accessed via a certain open nesting context, then all memory locations in that object must be accessed only via that same context.

2. The programmer must guarantee that open nesting contexts are partially ordered. That is, if an open action on object o_1 invokes an open action on object o_2, then an open action on o_2 should never invoke an open action on o_1.

Guideline 1 ensures that two conflicting accesses can only occur within the same context (or within no open context). Under this condition, a deadlock as in Figure 6 can only occur if the open nesting contexts form a cycle. However, if the programmer observes guideline 2, no such cycle can be formed.

In addition, the TM implementation must guarantee that two memory locations, at least one of which is open atomic, can conflict only if they are within the same object. This guarantees there is no false sharing to cause the same deadlock.

Use escaping newly-created objects with caution. Another kind of situation that can arise using open actions concerns the state of objects created by a closed ancestor of an open action. If the open action updates shared data to refer to the new object, and the parent aborts, the state of the object will be rolled back. The TM implementation must guarantee that the new object that is rolled back remains type-safe. Software TM can build this into its allocation support; hardware TM may need to run allocation, including object header initialization but not the `init` method of a constructor, as an open action.

```
public Object put(final Object key,Object value) {
  final Object[] returnValue = new Object[1];
  TxnDesc _td = txnGetDesc();
  Action handler = new Action() {
    public void execute() {
      if (returnValue[0] != null)
        map.put(key, returnValue[0]);
      else
        map.remove(key);
    }};
  while (true) {
    THandle _th = txnStart(_td);
    try {
      txnLock(_td, key, X);
      txnLock(_td, lock, IX);
      oldValue[0] = map.put(key, value);
      return oldValue[0];
    } finally {
      int committed = txnCommitOpen(_td, _th);
      if (committed == 1) {
        txnLogOnAbortHandler(_td, handler);
      } else if (committed == 0) continue;
    }}}
```

Figure 7. Code generated for `OpenMap.put` using Polyglot

3. Supporting Open Nesting in STM

We implemented open nesting as an extension to a Java-based software TM [1]. At the core of our system lies McRT-STM [19], an in-place update STM that implements strict two-phase locking [4] for writes, and optimistic concurrency control using versioning for reads. One may find detailed descriptions of our base system in earlier papers [1, 19]; here we give only a brief overview of those features relevant to understanding the open nesting implementation.

McRT-STM keeps track of the data accessed by a transaction using a per-transaction *read set*, *write set*, and *undo log*. The read and write sets record version numbers of the data that the transaction reads and writes, respectively. The undo log contains old values of data written by the transaction. The system uses the read set to validate the execution of a transaction. At validation time, if all the data read by the transaction still have the same version numbers as when they were originally read (and recorded in the read set), no other transaction can have written the data, and the transaction can commit. A different version number indicates a conflict and requires transaction abort. When a transaction commits or aborts it release locks on all the data it has written. On abort it also restores memory locations that it has written to their original values recorded in the undo log.

Our base system supports closed nested transactions. A nested transaction shares its read set, write set, and undo log with its ancestors. For each nested transaction, the system maintains a structure called a *transaction memento* that contains pointers to the portions of the read set, write set, and undo log pertaining to the nested transaction. This effectively turns the read/write sets and undo logs into a stack of such sets and logs, which we call the *transaction activation stack*, where each memento defines a frame in the stack.

3.1 Language Constructs

In our previous work [1], we prototyped transactional constructs in the Java language using Polyglot [16], an extensible source-to-source compiler for Java. We use our Polyglot extension to translate a Java program with transactional constructs to pure Java, and we have extended this further to support open nesting. In Figure 7, we show the generated Java code for the `put` method from Figure 4. The `onabort` handler is translated into the Java approximation of a closure: an object of an anonymous class extending a predefined class `Action`. When the open nested transaction commits, this object is

```
mcrtSTMStartNonNested(TxnDescriptor* td)
mcrtSTMCommitNonNested(TxnDescriptor* td)
mcrtSTMStartNested(TxnDescriptor* td,
                   Memento* mmt)
mcrtSTMCommitNested(TxnDescriptor* td,
                    Memento* mmt)
mcrtSTMCommitOpenNested(TxnDescriptor* td,
                        Memento* mmt)
mcrtSTMLock(TxnDescriptor* td,int object,int mode)
mcrtSTMLogOnAbortHandler(TxnDescriptor* td,
                         int object)
mcrtSTMLogOnCommitHandler(TxnDescriptor* td,
                          int object)
mcrtSTMLogOnValidationHandler(TxnDescriptor* td,
                              int object)
mcrtSTMLogOnTopCommitHandler(TxnDescriptor* td,
                             int object)
```

Table 1. Low-level STM API routines for open nesting

registered with the STM using `txnLogOnAbortHandler`. Abstract locks are translated into calls to `txnLock`, and released automatically by McRT-STM. All of these methods with the prefix `txn` are handled specially by the system. We modified StarJIT, our dynamic compiler, to translate these methods to a low-level McRT-STM API. We defined this API previously [19, 1] and extend it here with new functions to support open nesting. Table 1 shows relevant part of the extended API. StarJIT is able to track the nesting-level of transactions at compile time and translates STM operations to the appropriate top-level (e.g., `mcrtSTMStartNonNested`) or nested (e.g., `mcrtSTMStartNested`) McRT-STM call.

3.2 Validation and Commit

When an open nested action tries to commit, we first validate its read set, just as for closed actions. We scan the read set from the end (most recent entry) back to the memento point for the committing action. We compare version numbers in the read set to the current version numbers for the corresponding objects. If all reads are up to date, validation succeeds; otherwise, it fails.

If validation succeeds, the open action can commit, which requires three steps. First, we release the locks acquired by the action (these are recorded in the tail of the write set). Second, we discard the read/write sets and undo log, by cutting back to the memento pointers. This second step effectively pops the top frame from the transaction activation stack. And then, we append the handlers and the abstract locks of the committing atomic action to the parent's log.

If validation fails, the current open atomic action must abort, partially. First, we undo its writes and invoke any `on-abort` handlers registered by committed child open nested transactions, by scanning the undo log from the end to the memento, restoring memory locations to their recorded previous values and calling handlers. Then we release the memory-level or abstract locks acquired by the current action and its child transactions, by scanning the tail of the write set backward to the memento's start point. Finally, we discard the read/write sets and undo logs. After all this, execution returns to where the aborted action started.

If we fail to acquire a lock, not only do we abort the current action, but we roll back to the beginning of the containing top-level transaction. This is necessary to avoid deadlocks and livelocks. For example, transactions `T1` and `T2` below can livelock if we roll back only partially. (Methods `m1` and `m2` are open atomic.)

```
T1: atomic { a.f=...; m1(); }   m1(){ b.f=...; }
T2: atomic { b.f=...; m2(); }   m2(){ a.f=...; }
```

In a full roll back to the top level, we scan and undo all entries in the undo log, release all locks, and discard the entire read/write set and log, returning execution to the beginning of the top-level transaction. In certain cases, it may be safe to roll back less far.

3.3 Pessimism and Optimism

The semantics as presented in Section 2.3 have more the flavor of pessimistic locking: acquiring and checking read and write locks as we go, detecting conflicts early. Previous work with STM systems (our own and others) has shown that optimistic handling of reads (read versioning) is more efficient than pessimistically checking at each access. Rather, one notes the version read by any given transaction, and immediately before commit, validates that this is still the current version. In the case of flat transactions and closed nesting, it is straightforward to argue that pessimistic and optimistic execution both maintain consistency of committed work. Does adding open actions break this property? We claim that if open actions are abstractly serializable, then we maintain *abstract* consistency either way. The only difference is that an optimistic scheme may detect conflicts later, and need to execute more compensating actions.

3.4 Parent/Child Read/Write Set Overlap Problems

The validation and commit protocols we gave above work fine when parent actions and their open child actions access different memory locations. However, they require extension to handle cases where the sets overlap.

Parent reads location, then child writes it: Here the child's write, because it increases a version number, appears to conflict with the parent's read. Our proposed solution is to maintain, with each parent, an additional list, the *committed write list*, that enumerates versions written by committed descendants of the the given action. If a committing action has read version k of some object whose current version is $n > k$, then we check to make sure that *all* versions k through n (not including n) are in the committed write list, i.e., all the changes are "ours". If not, validation fails, otherwise it succeeds. Note that this also handles the case of an arbitrary number of committed written versions during the parent action.[4]

Parent writes location, then child also writes it: Here there are three issues. One is that when the child commits, we might release the write lock on the modified object. We record the write lock only in the oldest writing ancestor's frame, so it will be released only when that action commits. A second concern is that we might fail to undo properly if the child aborts and needs to be retried. But we undo back to the memento point, which properly restores previous memory contents. Another concern would be that our optimizations that try to avoid logging more than one write per transaction might overzealously optimize away a necessary log entry. But both our static and our dynamic optimizations respect transaction boundaries, so it is not a problem.

3.5 Handlers

Handlers and abstract locks generalize our existing STM data structures. On-abort handlers generalize undo log entries, on-validation handlers generalize read set entries, and on-commit handlers generalize buffered writes. (We do not actually buffer writes in the current implementation; our point is the conceptual correspondence.) Finally, abstract locks clearly generalize write locks. To implement handlers and abstract locks, we simply used the existing STM data structures, and extended them to deal with the new kinds of entries. In the extended STM, simple read/write set entries and undo logs are interleaved with abstract locks and closures for handlers.

---

[4] Our current implementation does not maintain and check the committed write list, since the discussed case is rare and does not come up in most properly structured programs including our benchmarks.

Note that we interleave entries in the natural way to achieve the previously specified semantics. In particular, on-abort handler entries interleave with previous-value entries in the undo log so that processing them all in reverse order implements our abort semantics. While the interleaving order is less important for on-validation handlers, we interleave them with read versions in the read set, and validation processes all entries, invoking on-validation handlers as they come up in the scan. On-commit and on-top-commit handlers require an additional list in our current STM, but are straightforward to add. In a buffering STM that creates copies for shared data in a private buffer, on-commit and on-top-commit handlers may be mapped into buffer entries for speculative writes.

Each handler is recorded in the STM data structures as a closure, i.e., code plus an execution environment. We implement these closures using Java anonymous classes. Each handler definition is translated source-to-source into an instantiation of an object of an anonymous class implementing a well known interface. The object is then recorded in the STM data structures after the associated open nested action commits.

3.6 Lock Manager

As described in Section 2, we rely on abstract locks to provide isolation at the abstract level. In general, abstract locking could be provided by the user, via handlers. For example, in a paper on LogTM [11], the authors show how to lock a B-Tree entry to enforce isolation at the entry level using open nesting. However, this approach is rather complicated for the programmer. It requires that a lock slot be added to each entry. It requires user-level condition synchronization in case the lock is already acquired. It also requires some mechanism for "locking" the fact that a key/entry is *not* in the data structure (as in Figure 2). For these reasons, we instead tightly integrated abstract locking into the language itself.

At an implementation-level, we also had the choice of providing abstract locks at the Java level (i.e., by mapping language extensions to a user-provided Java lock manager) or natively integrating into the underlying STM. For performance reasons we chose the latter route. We implemented a lock manager similar to that described by Gray [4]. It supports several standard lock modes to support hierarchical locking, including exclusive, shared, and intention locks. We can readily extend this approach to allow user-defined sets of lock *modes*, each such set having its own lock mode conflict matrix.

When an open nested transaction begins, the system attempts to acquire its abstract locks as shown by the `txnLock` operations in Figure 7. When an abstract lock is requested in a particular mode, the system checks whether the lock is free or held by any other threads. It tests whether the requested mode is compatible with the existing mode (e.g., multiple threads may hold a shared lock). If the modes are compatible, the existing mode is upgraded if necessary. The calling thread is added to the lock's list of holders, and the abstract lock is added to the current transaction's log. If the mode is not available, the lock cannot be acquired. In this case, we do a full abort. Full abort on a locking failure ensures that abstract locks do not lead to deadlock. For example, the partially lowered code sequence in Figure 8, and more complicated sequences, won't deadlock, even though the same locks are acquired in the opposite order.

Abstract locks are released by STM without intervention of user code or compiler-generated code. When an open atomic action com-

Transaction 1:

```
open {
  txnLock(desc, o1, X);
  txnLock(desc, o2, X);
}
```

Transaction 2:

```
open {
  txnLock(desc, o2, X);
  txnLock(desc, o1, X);
}
```

Figure 8. Abstract locks will not deadlock

mits, the abstract locks that it has acquired are appended to its parent's log. From then on, STM handles the abstract locks as ordinary locks, releasing them only when a top-level or open atomic action commits.

4. Experimental Results

To evaluate our system, we performed experiments using the standard `HashMap` and `TreeMap` data structures from the Java class library. In particular, we compared the performance of synchronized and atomic versions of these data structures with our new open nested versions. The workload consists of long-running transactions of map operations. In our experiments, we had 16 transactions, each consisting of 4096 `put` operations. We executed these transactions in various ways: all 16 serially in one thread; break them into groups, each group executed serially on one thread, concurrently with the other groups; or each transaction in its own thread, concurrent with all other transactions. In any case, a transaction still consisted of 4096 operations, which must be run as a single (large) atomic step. The code of each transaction looks like this:

```
atomic {
    for (int i = 0; i < 4096; i++)
        map.put(key[i], value[i]);
}
```

All keys across all transactions are different: there is no *logical* conflict. Ideally, if we run each transaction in a separate thread, we should see no conflicts. However, there will be physical conflicts on the shared internal state of the data structure, e.g., the buckets in the hash map, or nodes in the red-black tree.

A transaction consisting of a massive sequence of library calls is only one particular form of long-running transaction. Other long-running transactions might have a small number of library calls, each very time consuming. Although open nesting should also improve scalability of such long transactions, we have not studied them.

For open nesting, we initialized `map` using `OpenMap` as described in Section 2. For comparison, we used three other variants of the workload. For closed nesting, we initialized `map` using the `AtomicMap` wrapper. Closed nesting was implemented as runtime flattening in these experiments. No partial abort was used for closed nesting. For coarse-grained locking, we used the standard Java library class `HashMap` or `TreeMap`, changing the `atomic` keyword to `synchronized` in the driver above. For the unsafe version, we simply removed the `atomic` keyword from the code, and used the standard Java library classes. We do not present data here for unsafe code in multi-threaded cases, because such executions are incorrect.

We performed all experiments on a 16-way IBM xSeries 445 machine. This machine has 4 boards, each having 4 Intel Xeon processors running at 2.2 GHz. Each processor has 8 KB L1 cache, 512 KB L2 cache, and 2MB L3 cache. Each board has a shared 64MB L4 cache.

Consider first our results for `TreeMap`, varying the atomicity strategy and the number of threads, presented in Figure 10. We can see that closed nesting does not scale very well for this benchmark: it gives no speed-up compared to the unsafe single threaded version. In closed nesting, the read/write sets of all child transactions are ap-

```
import java.util.*;
public class AtomicMap implements Map {
    private final Map map;
    public AtomicMap(Map map) { this.map = map; }
    public Object put(Object key, Object value) {
        atomic { return map.put(key,value); }
    }
    ...
}
```

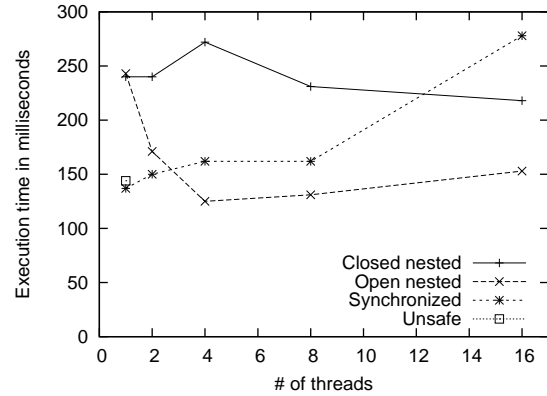**Figure 9.** A concurrent wrapper for `Map` using closed nesting

Figure 10. 65536 `put` operations on a `TreeMap`
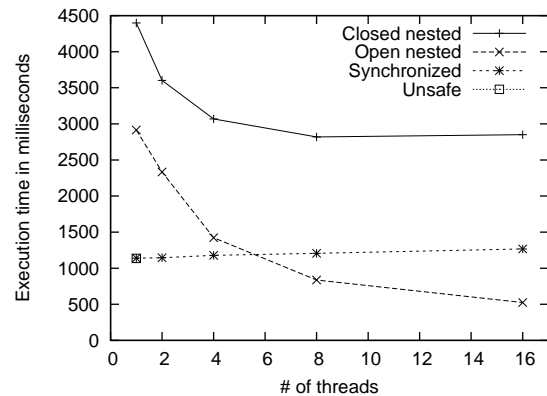

Figure 11. 65536 `put` operations on a 128 bucket `HashMap`

pended to the parent's. Thus, if any child writes the root of the red-black tree, it will conflict with other transactions, because all transactions read the root. And any such conflict will cause the transaction to abort and retry. Given the large number of `put` operations in each transaction, such conflicts are essentially unavoidable, so the transactions executed serially. Open nesting avoids this problem, so it scales fairly well. With 4 or more threads, it beats the synchronized version. With 4 or 8 threads, it is even faster than the unsafe single thread version. This demonstrates that, given enough concurrency and enough processors, open nesting's scalability enables it to overcome its overheads. Open nesting stops scaling after 4 threads, because the intentional lock on the single global `lock` object per data structure becomes a bottleneck and causes contention.

Now consider our results for `HashMap`. In this case, conflicts usually occur in accessing buckets or linked list entries. We populated the map before the long transaction of `puts`, so we avoided inserting new entries. In other words, we only did in-place updates. Therefore, we didn't have conflicts due to the buckets; but we still had conflicts due to linked list entries. If two keys are mapped to the same bucket, a transaction writing an entry nearer the front of the list will cause another transaction writing an entry nearer to the end of the list to abort, because the second transaction must read the earlier entry written by the first transaction in order to compare and find its entry to write. Figure 11 shows the results for a `HashMap` with 128 buckets. Closed nested transactions had huge read sets (from searching the lists) and poor performance (because of the conflicts in updating buckets' list pointers). It stopped scaling after 8 threads. Open nesting
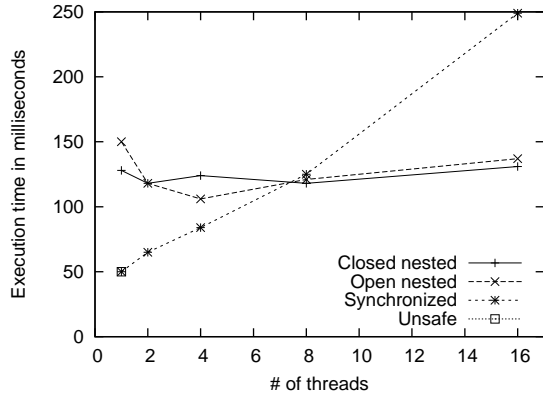
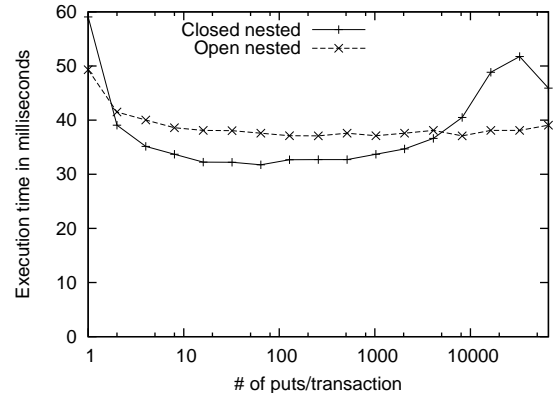**Figure 12.** 65536 `put` operations on a 65536 bucket `HashMap`

Figure 13. Locking overhead

Figure 14. 65536 `put` operations in a single thread
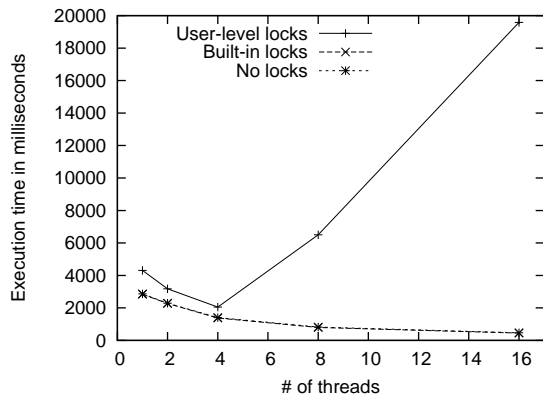
had smaller read sets and avoided the false conflicts due to linked list traversals, so it scaled very well and obtained speed-up compared to coarse-grained locking and unsafe code.

If we increase the number of buckets, we increase the concurrency of the `HashMap`, and the performance of the various approaches changes. Figure 12 shows results for a `HashMap` with 65536 buckets.

In Figure 13 we present the overhead for abstract locks—user-level or built-in—compared to a baseline that uses no abstract locks in open atomic actions. We repeated our previous experiments using `OpenMap` with 128 buckets, with abstract locks using different lock managers, and even without abstract locks. These results show that a user-level lock manager is unacceptably slow, although it allows rapid prototyping. A user-level lock manager is much easier to implement because we benefit from all the rich features we added into Java for transactional memory, such as atomic actions, `retry`, and open nesting. Note that the user-level lock manager caused more scalability problems in this experiment, due to conflicts accessing its lock table. In contrast, a built-in lock manager implemented natively in the STM is much faster, although more difficult to implement. Compared to the same program with all abstract locking removed, a built-in lock manager added only ~25ms overhead in our experiments, while the overall execution time (without abstract locks) varies from 2844ms to 453ms, as we increase the number of threads.

In Figure 14 we show the effect of transaction size At each point we executed 65536 `put` operations in a single thread. The x-axis shows the number of `put` operations per transaction for a given run. At the far left, we executed 65536 transactions of one `put` each, and

at the far right, we executed one single transaction of 65536 `puts`. In between, the transaction size doubled at each data point from left to right, while the total number of `puts` stayed constant. In other words, the number of transactions decreased by 50% at each data point from left to right. On the *open nested* line, each `put` is an open atomic action, while on the *closed nested* line, the `put` is part of the single top-level transaction.

We can see that, for small transaction sizes in general, open nesting exhibits a small overhead over flattened, closed nesting. This overhead is from 2% to 18% for transaction size from 2 to 4096. As transaction size increases, however, we see a penalty for closed nested transactions. At 32768 `puts` per transaction, the open nested variant is roughly 26% *faster* than the flattened one, and at 65536 `puts` per transaction, 15% *faster*. We believe that this results from memory effects when the read set size exceeds the lowest level of cache. With open nested transactions, however, each `put` operation is validated almost immediately, and its read set entries removed at that time. As a result, the maximum read set size is much lower, speeding validation. Note that we used a more light-weight driver in this experiment, so we observed better performance than our previous results for `HashMap`.

Figure 14 suggests that reducing read set size would potentially improve the performance of closed nesting. We did some preliminary experiments using runtime read set filtering, which reduces redundant read set entries. The results did show improvement to single thread performance for flattening. However, it did not improve the scalability of closed nesting. Open nesting still scaled better, and showed better performance given enough processors. Details about runtime filtering are out of the scope of this paper, and will be covered in our future publication.

5. Related Work

Transaction processing [4] has been widely studied in the database community for decades. Moss proposed [12] nested transactions to improve the performance and reliability of transaction execution for distributed systems. In that model, nested transactions are closed; the parent transaction inherits all locks of a committing child transaction. However, on conflict a child transaction can be aborted without aborting the parent, reducing the overall cost of an abort.[5] Other researchers in the database community extended this idea to support open nesting models that allow child transactions to release locks early. Garcia-Molina proposed [3] to use semantic knowledge to allow a transaction to release locks early and, instead, register a compensating action to

---

[5] Moss's scheme also permits child transactions themselves to be performed in parallel; this form of nested parallelism is beyond the scope of this paper. We use *linear nesting*, as described by Moss and Hosking [14, 15].

be performed if a transaction must abort. Weikum and others [21, 22] described a theory of *multi-level serializability* that describes serializability at different levels of abstraction and accounts for compensating actions. In this database context, the different levels of abstraction were well-defined and disjoint. For example, in a two-level scheme, the higher level might correspond to tuples and their insert, removal, and updates, while the lower level might correspond to pages and specific read and write operations to those pages. Importantly, a system only needs to reason about conflicts at the same level.

Herlihy and Moss [8] coined the term *transactional memory* to denote transactional access to shared memory. Since then, researchers [10, 17, 20, 7, 5, 9, 18, 23] have proposed and built a number of hardware and software transactional memory implementations. Most implementations either did not support nested transactions or simply *flattened* nested transactions into a single top-level transaction. More recently, Harris et al. [6] argued that closed nested transactions, supporting partial rollback, are important to implement composable transactions, and presented an *orElse* construct that relies upon closed nesting. In an earlier paper [1], we demonstrated a Java-based system that provided both nested atomic regions and *orElse*, and we introduced the notion of *mementos* to support efficient partial rollback.

In the past year, a number of researchers have proposed the use of open nesting in transactional memory. Moss described the use of open nesting to implement more highly concurrent data structures in a transactional setting [13]. In contrast to the database setting, the different levels of nesting are not well-defined; thus different levels may conflict. For example, a parent and child may both access the same memory location and conflict. Both TCC [2, 10] and LogTM [11] describe hardware transactional memory implementations of closed and open nesting. Both support commit and abort handlers for open nesting. Our work is different in several ways. We describe the first implementation of open nesting in a *software* transactional memory system. Second, our implementation is the first to support arbitrary levels of open nesting. Both hardware implementations above are inherently limited by the the number of bits set aside per cache line or the associativity of the cache. Third, we provide the first system that integrates two-phase abstract locking. Finally, we offer performance results from a running system, while hardware proposals necessarily rely on simulation.

6. Conclusions

We presented a full-featured and efficient implementation of open nested transactions in a software transactional memory system. We prototyped extensions to Java to define open nested regions with handlers to support high-level validation, early release of memory resources, and rollback, and with abstract locking to support high-level concurrency control. We described an efficient mapping of these features to a high-performance software transactional memory implementation. Finally, we illustrated the costs and benefits of open nesting on Java data structures in a running implementation. We found that open nesting is effective in increasing concurrency among transactions and reducing the overhead of individual transactions.

Acknowledgments

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay S. Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, June 2006.

[2] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM Press.

[3] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[4] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[5] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA 2003: Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[6] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP 2005: Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[7] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC 2003: Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA 1993: International Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[9] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Design tradeoffs in modern software transactional memory systems. In *LCR 2004: Languages, Compilers, and Run-time Support for Scalable Systems*, 2004.

[10] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 378–391, San Jose, CA, USA, October 2006.

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

[13] J. Eliot B. Moss. Open nested transactions: Semantics and support. In *WMPI 2005*, 2005. Poster presentation.

[14] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and preliminary architecture sketches. In *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[15] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming (Elsevier)*, 63(2):186–201, December 2006.

[16] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: an extensible compiler framework for Java. In *CC: International Conference on Compiler Construction*, Lecture Notes in Computer Science 2622, pages 138–152, April 2003.

[17] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA 2005: International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Michael F. Ringenburg and Dan Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP 2005: International Conference on Functional Programming*, pages 92–104, New York, NY, USA, 2005.

[19] Bratin Saha, Ali-Reza Adl-Tabatabai, Rick Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Principles and Practice of Parallel Programming*, pages 187–197, 2006.

[20] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC 1995: Principles of Distributed Computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[21] Gerhard Weikum. A theoretical foundation of multi-level concurrency control. In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 31–43, New York, NY, USA, 1986. ACM Press.

[22] Gerhard Weikum, Christof Hasse, Peter Broessler, and Peter Muth. Multi-level recovery. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 109–123, New York, NY, USA, 1990. ACM Press.

[23] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *ECOOP 2004: European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.