Closed and Open Nested Atomic Actions for Java

Language Design and Prototype Implementation

Keith Chapman Antony L. Hosking
Purdue University
{keith,hosking}@cs.purdue.edu

J. Eliot B. Moss Tim Richards
University of Massachusetts at Amherst
{moss,richards}@cs.umass.edu

Abstract

We describe the design and prototype implementation of a dialect of Java, XJ, that supports both closed and open nested transactions. As we have previously advocated, open nesting most naturally attaches to the *class* as the primary abstraction mechanism of Java. The resulting design allows natural expression of layered abstractions for concurrent data structures, while promoting improved concurrency for operations on those abstractions. Moreover, we describe our approach to constructing a prototype implementation of XJ that runs on standard Java virtual machines, by grafting support for transactions onto both application code and library code via load-time bytecode rewriting, for full execution coverage. We rely on extensions to the `javac` compiler, a JVMTI run-time agent to intercept and rewrite Java classes as they are loaded into the virtual machine, and a run-time library that tracks and manages all transaction meta-data. The resulting prototype will allow further exploration of implementation alternatives for open and closed nested transactions in Java. Our design also addresses the issue of internal deadlock caused by accessing the same data in both closed and open nesting fashion by carefully disallowing such access.

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

Transactional memory (TM) is a paradigm for programming concurrent applications that allows programmers to designate sections of code as *atomic transactions*. Transactions appear to execute atomically: no thread executing an atomic section sees the intermediate states of other transactions executing in other threads. Transactional memory is more abstract than using Java *synchronized* blocks and methods that lock a specific object, and avoids problems encountered with locks, such as deadlock, priority inversion, convoying, pre-emption, and reduced concurrency.

PPPJ'14, September 23–26, 2014, Cracow, Poland.
Copyright © 2014 ACM 978-1-4503-2926-2/14/09…$15.00.
http://dx.doi.org/10.1145/2647508.2647525

There have been a number of proposals to support TM abstractions in programming languages, including a standardization effort for C++ [21], and several proposals for Java [1, 2, 5, 9, 11, 12]. Most allow *nesting* so that an atomic block or method can be used by another atomic block or method, but ignore the internal structure and simply group them into one large transaction. However, some systems support *nested transactions*, typically in *closed* form [14]. We have also advocated for *open* nested transactions [15, 17] as a powerful extension to closed nesting, allowing improved concurrency, at the cost of some programmer effort to program the necessary concurrency control to achieve the desired concurrency semantics, and compensating actions to handle transaction abort. Our approach extends the language for transactional memory [4, 11, 12], rather than exposing a programmatic API [5, 7–9, 16, 20], allowing full integration of transaction semantics for both application code and external/legacy libraries.

Contributions. We present the design and implementation of our prototype extension of the Java programming language that supports both open and closed nested transactions, producing Java bytecode that will run on an unmodified Java virtual machine. We refine our earlier proposal for open nesting constructs [17] to combine open nested *classes* with a rich range of *abstract locks*, used to represent the abstract resources acquired by open nested transactions as they commit. For example, when an open nested transaction commits physical insertion of a particular element into a Java `HashMap` it must abstractly lock the presence of that element to prevent other transactions from observing that element until its outermost parent transaction commits. We provide a number of lock abstractions for use by programmers of transactional data structures, and illustrate their use with examples. We give a full description of the new language features for open and closed nesting, describe how those extensions propagate through the source code compiler to be represented in bytecode class files, how to dynamically inject operations into the bytecode to track memory read/write conflicts, and the run-time library enabling transactional execution.

Earlier approaches to adding open nesting to a programming language were vulnerable to a kind of internal deadlock that could prevent both forward progress of a transaction and successful undo to abort and remove the transaction. The design presented here solves that problem, overcoming a key reliability concern.

Since open nesting requires additional programmer effort, and incurs a bit more run-time overhead, it will likely find greatest use in selected places in any given program, places that would otherwise be concurrency bottlenecks. Our design makes it straightforward to extend an existing non-atomic or closed nested version of a class with an open nested subclass. One simply adds the abstract locking and operation undo information in "wrapper" methods that invoke the superclass version of the same methods. This makes it easy to refine a program to overcome concurrency bottlenecks as they are identified.

2. Principles and approach

Transactions are usually described in terms of read and write operations performed against disjoint memory locations. Hardware and software transactional memory work in terms of either hardware memory units such as bytes, words, or cache lines, or, when incorporated into a programming language, in terms of variables/fields or objects. Regardless of the memory units in play, the transaction mechanism tracks reads and writes of those units to detect conflicts (two transactions access the same unit and at least one of them writes it) and manage atomicity (either all of a transaction's writes occur, or none of them, and they appear to occur at a single instant in time).

Here we will describe a design for Java that performs conflict detection on the unit of objects, and that tracks writes at the level of object (and static) fields. We adopt pessimistic concurrency control for objects modified by a transaction (i.e., writing requires acquiring a lock) and handle atomicity of update by allowing updates in place and undoing a transaction's uncommitted writes if the transaction aborts. Thus we will use an undo log. In principle any of these decisions could be varied; some would have a degree of visible impact on the language design, though much would remain the same. While we agree that no particular transaction management policy offers the best performance under all workloads, this seems to be a reasonable "middle of the road" choice. Nevertheless, we have designed our prototype implementation to allow future experimentation with alternative approaches.

What we just described briefly characterizes flat, *non-nested*, transactions. Here is a correspondingly short description of closed nesting. A closed nested transaction is either *top-level*, or a *child* subtransaction nested within a *parent* transaction. Logically, transactions accumulate read and write sets, which determine *conflicts* as well as what writes become visible upon commit.

Updates become globally visible only when a top-level transaction commits. When a transaction reads a value, it sees the value in its own read or write set (if there is one), otherwise the value seen by its parent. A top-level transaction will see the latest (globally committed) value, subject to subsequent overriding of the parent's values when the child commits, as follows.

When a nested transaction commits, its read and write sets merge with its parent's, the child's writes overriding any previous value in the parent. When a top-level transaction commits, its writes become permanent. When a transaction aborts, its read and write sets (and associated updates) are discarded or *rolled back*.[1] A transaction can succeed only if it has no conflicts with other transactions. Nested transactions refine the earlier definition of conflict (two transactions both accessing the same unit, at least one of them writing it) to say that there is conflict only when neither transaction is an ancestor of the other. In case of conflict, either or both must abort to prevent violation of transaction semantics, in which a legal execution is equivalent to a serial execution of the committed transactions (only), in some order (i.e., *serializability* [18]).

Nested transactions allow decomposition of a large transaction into smaller subtransactions, each of which can attempt some portion of work, and possibly fail (and perhaps be retried) without aborting work already accomplished by the parent. However, the parent still accumulates the read and write sets of all of its committed children (so writes by a child become visible to other unrelated transactions only when the top-most ancestor commits). Thus, as large transactions accumulate ever larger read and write sets from their children they will become more prone to failure due to higher probability of conflict. These failures reduce system throughput (the effective degree of concurrency).

Open nesting allows *further* increases in concurrency [17], by releasing concrete resources (e.g., memory reads and writes) earlier and applying conflict detection (and roll back) at a higher abstraction level. 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 and the updated field would not be part of the parent transaction's read or write set. However, if the parent later aborts, it needs to run a compensating decrement to roll back the logical effect of the committed open nested transaction.

The only difference between open and closed nesting in terms of the read/write set execution model concerns what happens when a transaction commits. When an open nested transaction commits, it discards its read set, and commits its writes globally *at top level*.[2]

To support moving conflict detection from the concrete to the abstract level, when the committing open nested transaction releases its concrete memory resources (i.e., its memory 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 roll back.

Prior work [17] showed that in some cases open nesting can greatly increase concurrency. However it does place more of burden on programmers who use it, since they (a) need to get the compensating actions right, and (b) likewise need to provide suitable abstract concurrency control. It has also been observed that if open and closed nesting are ever applied to the *same object*, deadlocks can occur that block both the completion of an open nesting action and a compensating action needed to abort the ongoing transaction.

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 *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[3] 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* [10]: 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* an open nested transaction using much the same mechanism as for closed nesting.[4]

[1] If the system performs updates in place and keeps an undo log, on commit of a child transaction the child's undo log is appended to the parent's. Thus, abort of the parent will remove the effects of the child *and* any other preceding effects recorded earlier in the parent's undo log.

[2] It further discards its written data elements from the read and write sets of all other transactions. Given the conflict rules, these can only be its ancestors (it cannot commit unless those other unrelated conflicting transactions also abort). Well-structured programs respecting proper abstraction boundaries (not manipulating the same state at different transaction levels) will avoid this situation, but the rule makes the commit global, as intended.

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

[4] Transactional boosting [6], however, recognizes that *how* that linearizability is achieved does not matter, and thus naturally supports an approach where existing non-transactional code is extended with transactional wrappers. It still needs abstract concurrency control of some kind, etc.

3. Language design

We now present details of our extensions to the Java language that add transactions, with both open and closed nesting, to the language. While closed nesting need not be associated only with classes, we connect open nesting with classes. In order to avoid deadlock internal to the transaction system, the design prevents any given static or instance field from being accessed by both closed and open nested transactions. Associating open nesting with classes also facilitates this segregation.

3.1 Atomic actions

A block (or method) may be designated `atomic`, by writing the keyword `atomic` where the keyword `synchronized` is permitted. A block (or method) cannot be both `atomic` and `synchronized`.[5] Each execution of an `atomic` block (which includes method bodies) occurs as an *atomic (trans)action*.[6] An atomic action has three possible outcomes:

- It can *succeed*, 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), in which case its effects are undone specially (§3.1.5) and the action is *not* retried. Action failure results from throwing of exceptions.

The *effects* of an atomic action include assignments to (shared) instance and static 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 (see §3.2 for consideration of the case of *open* atomic actions).

In addition to designating `atomic` methods individually, one may write `atomic` as a class modifier. This causes all methods of the class to be implicitly `atomic`. Any class that extends an `atomic` class is implicitly also `atomic`, unless the extending class is explicitly marked `openatomic` (see §3.2).

3.1.1 Effect logging

It is helpful to consider the run-time system as (conceptually) associating with each thread a *log* of all the thread's assignments. Each record in the log indicates the variable that was assigned and the variable's previous value (§3.2.3 extends this model to include other kinds of log records). Undoing the effects of an atomic action requires processing each of the log records since the action started, from last to first, restoring each variable to its logged prior value.[7] Undoing also discards each log record after it is processed. Likewise, committing a top-level action discards that thread's log records. Committing a non-top-level action appends its log to its parent's log.

3.1.2 Concurrency control

If a thread reads a variable while executing an atomic action, the variable is said to be a member of the action's *read set*. Likewise, if a thread writes a variable while executing an atomic action, the variable is said to be a member of the action's *write set*. An action's *accessed variable set* is the union of its read set and its write set. If the write set of an action has a non-empty intersection with the accessed variable set of another thread's action, the actions are said to *conflict*. If two concurrent actions conflict, then at least one of them must abort.

[5] We may propose to remove `synchronized` entirely.

[6] We use the term *atomic action* for brevity, to refer to the execution of an atomic block/method as a transaction.

[7] Undoubtedly many optimizations are possible!

3.1.3 `Retry` statement

The `retry` statement allows explicit programming of abort. It is useful in implementing open atomic concurrency control (§3.2.6), etc. When a thread executes a `retry` statement, the atomic action aborts immediately, and will be retried from the beginning of the action's block. Executing a `retry` statement when not in an atomic action causes a run-time error exception to be thrown.

```
RetryStatement:
  retry;
```

Syntactically, a `retry` statement can appear anywhere a `return` statement can appear.

3.1.4 `Require` statement

The *require* statement supports *conditional* atomic actions:[8]

```
RequireStatement:
  require Expression ;
```

The *`Expression`* must be boolean-valued. The effect of evaluating

```
require exp ;
```

is similar to evaluating

```
if (!exp) retry ;
```

However, an implementation may be able to use knowledge of the required condition to avoid retrying if the condition's value cannot have changed.[9]

3.1.5 Exceptions

If an exception is thrown and not caught within an atomic action (i.e., the atomic action would complete abruptly), the atomic action fails, and is undone in a special way, as follows. Exception objects that are constructed and thrown, and new objects reachable from them, should not have effects related to them undone. If those effects were undone, the objects would have their fields reset to the value before any initializers were run (i.e., zero). Therefore, an implementation must not undo effects on fields of objects created since the action began. Moreover, at the time of an exception, this enables programmers easily to capture and communicate the state of previously existing objects using cloning or other copying of state into the corresponding exception object. This state will survive the failure of the enclosing action.

3.2 Open atomic classes

A class can be declared with the (new) modifier `openatomic`. This indicates that the open atomic instance or static *fields* of the class can be accessed only during execution of open atomic instance or static *methods* of the class.

> *Commentary:* As noted with our principles (§2), `openatomic` is a property of a class because all operations of the abstract data type implemented by the class need to cooperate in providing suitable abstract concurrency control and recovery.

The `openatomic` modifier is independent of the other usual class modifiers (`abstract`, `final`, etc.), and applies equally

[8] We considered calling this *wait* or *await*, but its semantics are different enough from Java's current wait/notify model that we prefer to emphasize that it is different.

[9] We considered as an alternative the `watch` statement of Atomos [4], but felt that because it is so low level, it might overly constrain implementation strategies. Also, if a programmer mentions too small a watched variable set, then the program can surprisingly wait forever.

to enumerations and nested classes. Of course, a class cannot be both `atomic` and `openatomic`. Any class that extends an `openatomic` class is implicitly also `openatomic`. An `openatomic` class can extend an `atomic` class, but an `atomic` class cannot extend an `openatomic` class. We detail the reasons for this in §3.2.4.

Interfaces cannot be declared `openatomic` (which is a semantic and implementation property, not affecting signature or usage).

3.2.1 Open atomic fields

All `private` or `protected` instance fields of `openatomic` classes are open atomic. Only `private` static fields of `openatomic` classes are open atomic. All accesses to open atomic fields are statically guaranteed to occur during the execution of an open atomic action. All other fields are not open atomic, and a warning will be emitted at their declaration in an `openatomic` class.[10] Any field with the `final` modifier is treated as open atomic irrespective of its access modifier (this allows a `final` field to be accessed by open atomic methods and also from elsewhere, according to its access modifier).

3.2.2 Open atomic methods

A method is open atomic if it has at least one of the following clauses attached: `onabort`, `oncommit`, `onvalidate`, `ontopcommit`, or `locking`. (The first four are introduced in §3.2.5; `locking` clauses are described in §3.2.6.) Only an `openatomic` class can have open atomic methods. Moreover, all public or package access methods of an `openatomic` class are implicitly open atomic; they cannot be `atomic`.

> *Commentary:* These rules are intended to prevent calls from outside the class that access open atomic *instance* fields other than during execution of an open atomic method on that *instance*, and likewise to prevent access to open atomic *static* fields other than during execution of an open atomic *static or instance* method. We assume that open atomic *instance* methods that directly or indirectly access open atomic *static* fields provide suitable *class*-level concurrency control and recovery.

Private or protected methods of `openatomic` classes can still be non-atomic. They can also be `atomic`, allowing a method that atomically composes invocations of two or more open atomic methods, for example.

3.2.3 Open atomic method execution

An open atomic method always executes as an atomic action. However, if it completes successfully (commits), its writes are made permanent (globally visible), and its log is discarded. Moreover, if the open action is also nested then it has the following effects on its parent's log:

- Its handler clauses (`onabort`, `oncommit`, `onvalidate`, and `ontopcommit`, whichever exist) *take effect* (are logged). Clauses in effect may later be executed, under certain conditions.
- It acquires abstract locks, as described in its `locking` clauses (if any), which are logged.

Discarding its log means that any clauses in effect from open atomic actions on other instances or classes, committed during this open atomic action, become no longer in effect. Discarding its log also means releasing locks held from such actions.

Because the open atomic public/package methods of an `openatomic` class are its only external entry points, each of which begins an open atomic action on entry, all methods of an `openatomic` class are guaranteed to execute in the dynamic context of an open atomic action, or nested within one. There are occasions when one open atomic method may *internally* call another open atomic method in the same class (or superclass), in which case their effects are *aggregated*, merging the open atomic callee into the caller's action. This avoids the need to duplicate internal subtransaction handlers in their parent's handlers.

For example, if a linked list class has open atomic `add` and `remove` methods, one might write an open atomic `move` method to move an item from its current position to the end of the list. If `move` is written as `remove` followed by `add`, then the `onabort` actions for both `move` and `add` accrue to `move`, instead of being discarded when `move` commits. Otherwise, one would be forced to duplicate them in the `onabort` clause for `move`.

> *Commentary:* One might implement aggregation as follows. For each open atomic method `m` create a corresponding non-open "internal method" `mInternal` having the same signature and body, but not open atomic. Rewrite internal calls of `m` to call `mInternal`.

When an open atomic method completes successfully, its open atomic clauses and locks, some of which may come from aggregated calls, are *logged* at that time to its parent. Thus the log described in §3.1.1 also contains records for `onabort`, `oncommit`, `onvalidate`, `ontopcommit`, and `locking` clauses.

Undoing an atomic action (because of abort or failure), processes its portion of the log in reverse order (as in §3.1.1). Processing ignores `oncommit`, `onvalidate`, and `ontopcommit` records. It also ignores records corresponding to `locking` clauses (these are released as described in §3.2.6). When undoing encounters an `onabort` record, it executes the corresponding `onabort` block as an open atomic action. Notice that undoing of writes and execution of `onabort` clauses are interleaved (but not concurrent): all occur in reverse log order. The processed log records are discarded, as described in §3.1.1. Finally, control resumes at the beginning of the aborted action, if it is to be retried, or the exception causing failure is propagated.

Committing an atomic action processes its portion of the log in forward order from the beginning to the end. Processing first runs the `onvalidate` records to ensure the transaction is in a state that can be committed. It then processes the `oncommit` records. If the committing action is a top level transaction it then processes the `ontopcommit` records. Processing these handler records causes their corresponding clauses to be executed as open atomic actions. Log records for writes, and `onabort` and `locking` records, are ignored when committing. Committing then discards the processed log records and releases all of the committing action's abstract locks (see §3.2.6). Control then continues normally.

3.2.4 Inheritance, overriding, and nesting

An `openatomic` class can extend a class having public/package `atomic` methods, but inheriting those methods without overriding them in the `openatomic` class is dangerous because it allows accessing fields of the open atomic instances in both open and closed execution modes. Mixing access modes in this way can lead to deadlocks [17].

To avoid this, we can either require that all inherited `atomic` methods be explicitly overridden with open atomic methods in the `openatomic` class, or implicitly "copy down" the inherited method as an open atomic method. The latter may save some typing

[10] Because `public` and `package` access fields can be accessed directly from outside of the class, we cannot restrict them to be accessed only during execution of open atomic methods. Similarly, `protected` static fields can be accessed directly from subclasses, so we cannot restrict their access to occur during execution of open atomic methods.

by the programmer, but the former has the advantage of forcing her to think through the abstract locking protocol for all the open atomic methods of the `openatomic` subclass. Our inclination is toward forcing the programmer to provide explicit overrides. Invoking the `atomic` superclass method with a `super` call from the body of the overriding open atomic method (or elsewhere in the subclass) is always safe, because instance field accesses will always occur in the context of an open atomic action.[11]

Conversely, an `atomic` class cannot extend an `openatomic` class. Otherwise, calls using `super` would enable the subclass to access fields in both open and closed modes.[12]

Similarly, a nested class (either static or non-static), which can directly manipulate the open atomic fields of its outer class or instance, is implicitly `openatomic` if its outer class is `openatomic`. This ensures that external entry points via the nested class also preserve the open atomic nature of the enclosing class's open atomic fields.

One additional piece of mechanism is necessary to ensure proper handling of open atomic fields. It is possible for an open atomic method on instance *o* to call methods on some other objects, resulting in a call chain that comes back to calling a method on *o*. Unlike aggregation to construct larger open atomic actions from smaller ones `operating on the same object`, where the call chain does not leave the scope of the instance, in this case the call chain is *re-entrant* after leaving the instance. In such cases, the re-entrant open action cannot safely release its physical updates, since the outer open action on that object is still active. Thus, we also formulate an additional run-time restriction, as follows. For any given object accessed in an open atomic way, indirect (non-aggregating) re-entrant calls to open atomic actions run instead as closed atomic. The requirement is analogous to the tracking of re-entrant nesting depth for Java `synchronized` blocks/methods, where the lock is released only when exiting the outermost lock level.

3.2.5 Open atomic method suffix clauses

We now give the syntax for the handler clauses that may be attached to the end of an open atomic method, namely `onabort`, `oncommit`, `onvalidate`, and `ontopcommit` clauses. A given method may have at most one of each kind of clause attached. Moreover, because the handlers may wish to use values computed at the beginning of the action, an optional list of local variable declarations can be evaluated before the method body proper. These pre-declarations (`PreDecls`) evaluate at the same level as the method body, in the scope of the formal parameters, and are delimited syntactically by square brackets `[]`. The variables they declare are in scope for both the method body and the handler clauses.

```
MethodDeclarator:
  Identifier ([FormalParameterList]) [PreDecls]
PreDecls:
  [{LocalVariableDeclarationStatement}]
MethodBody:
  Block {OpenAtomicClause}
  {OpenAtomicClause} ;
OpenAtomicClause:
  onabort     Block
  oncommit    Block
  onvalidate  Block
  ontopcommit Block
```

[11] It may not be *correct*, however, unless the overriding method adds suitable `locking` and `onabort` clauses, etc.

[12] Alternatively one could have it mean something like "copy down all the methods, removing all their open atomic clauses".

```
public interface LockTable
  <S extends LockSpace, M extends LockMode> {
    public S getSpace();
    public void
      add(Lock<S> space, M mode, TxnDescriptor
           desc) throws TxnException;
}

public interface LockSpace<T> {}

public interface Lock<S extends LockSpace> {
    public S getSpace();
}

public interface LockMode<T> {
    public boolean conflictsWith(T m);
}
```

Listing 1: Lock tables, spaces, and modes

Supporting these constructs, and supporting use of method parameter values and pre-declarations, requires generating code that saves the necessary values and makes them available to the handler clauses if and when they run.

3.2.6 Open atomic method `locking` clause

We provide a framework for users to construct their own abstract locking protocols, along with several pre-defined abstract lock libraries. The basics of this framework rely on the declarations shown in Listing 1.

An instance of an open atomic class will have one or more *lock table* instances (implementing `LockTable`) for representing abstract locks held on the open atomic instance. A lock table comprises a *lock space* (`LockSpace`) instance and a *lock mode* type (`LockMode`). An open atomic method invocation can try to obtain one or more abstract locks. These are specified via `locking` clauses associated with the method, and `return` or `throw` statements in its body. An abstract lock needs to:

1. indicate the lock table instance in which to request the abstract lock;
2. indicate the specific lock *shape* requested (and any parameters needed for that shape) within the table's lock space; and
3. indicate the specific lock mode instance to use.

As an example, consider an open atomic class `OrderedSet<T>` implementing `java.util.SortedSet<T>`. A suitable lock space would be the one dimensional set of all possible `T` instances, ordered by the total order being used. Within this space we can imagine a number of shapes:

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

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

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

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

Example lock modes might include reader/writer locks, as illustrated in Listing 2: reads conflict only with writes, and writes conflict with both reads and writes.

Another example implements modes "pin" (reading a value "pins" it, so that any change to it conflicts), and "change" (changing

```
enum RW implements LockMode<RW> {
  Read {
    public boolean conflictsWith(RW other) {
      return other == Write;
    }
  },
  Write {
    public boolean conflictsWith(RW other) {
      return true;
    }
  }
}
```

Listing 2: `Read/Write` lock modes

```
enum PinChange implements LockMode<PinChange> {
  Pin, Change;
  public boolean conflictsWith(PinChange other) {
    return this != other;
  }
}
```

Listing 3: `PinChange` lock modes

```
openatomic class OrderedSet<T> ... {
  private LockTable<LockOneD<T>,RW> eltLocks =
       new LockTable<...>();
  private LockTable<Points<T>,PinChange>
       statsLocks = new LockTable<...>();
  ...
}
```

Listing 4: `OrderedSet` lock tables

a value by incrementing or decrementing it commutes with other changes, but not with reads). These modes are captured in Listing 3.

An `OrderedSet<T>` might then have two lock tables, one for the set of elements and one for statistics (current size, total number of insertions/deletions, etc.), as in Listing 4.

In addition to the suffix clauses, an open atomic method may acquire abstract locks before it can complete successfully. The `locking` clause is attached to the method's header, revising the syntax of *`MethodDeclaration`*:

```
MethodDeclaration:
  MethodHeader [LockingClause] MethodBody
LockingClause:
  locking [+]( LockExpressions )
LockExpressions:
  LockExpression {, LockExpression}
LockExpression:
  LockTableExp : LockShapeExp : LockModeExp
LockTableExp:
  Expression
LockShapeExp:
  Expression
LockModeExp:
  Expression
```

A locking clause is syntactic sugar for acquiring a lock from a lock table.

The *`LockTableExp`* must have a type that implements the `LockTable` interface; it indicates the lock table in which to request the abstract lock denoted by the locking clause. A `LockTable` encapsulates a `LockSpace` instance and a `LockMode` type. These are defined in a standard library as shown in Listing 1. An open atomic class will typically have one or more `LockTable` instances for representing abstract locks held on itself or its instances. The *`LockShapeExp`* must return a `Lock`, by invoking the indicated method on the lock table `LockSpace`, itself obtained using `getSpace()`. It indicates the specific shape requested (and any parameters needed for that shape) within the table's lock space. The *`LockModeExp`* must be of a type that implements the `LockMode` interface; it indicates the mode in which to acquire the lock. A `locking` clause has the same scoping behavior as the suffix clauses.

An overriding method inherits the overridden method's `locking` clause. If an overriding method supplies its *own* `locking` clause, then the overridden clause is not inherited. If a method needs to *extend* an inherited `locking` clause, it can use the optional + sign with its `locking` clause.

At the time an open atomic method execution accumulates locks, one evaluates each *`LockExpression`* in turn, in textual order. To evaluate a *`LockExpression`*, one first obtains the `LockSpace` from the `LockTable`. One then calls the method described by the *`LockShapeExp`*; this results in a `Lock` type. The next step is to attempt acquiring the abstract lock. This is done by calling the `add` method on the `LockTable` instance specified by the *`LockTableExp`*. If the `add` call completes successfully, then we say that the current transaction *holds* a lock on the specified object in the specified mode. The call may fail due to lock conflict. In this case the current transaction aborts and will be retried.

When a transaction completes (successfully or unsuccessfully) and *releases* its locks, it no longer holds them.[13]

3.2.7 Acquiring locks at `return` or `throw`

Sometimes, throwing an exception indicates something about an object's state. For example, calling `remove()` on an empty `Queue` throws `NoSuchElementException`. Arguably, this should lock the fact that the queue is empty. However, our interpretation of exceptions as causing abort prevents `remove()` from acquiring such an abstract lock on the queue's state. Hence, we allow one to attach a `locking` clause to a `throw` statement:

```
ThrowStatement:
  throw Expression [LockingClause] ;
```

The indicated locks are acquired as the exception is thrown, and are logged as part of the containing action. If execution is not within an atomic action (open or not), the `locking` clause has no effect.

Similarly one can have a `locking` clause attached to a `return` statement and the locks are acquired as the result is returned and logged as part of the containing action:

```
ReturnStatement:
  return [Expression] [LockingClause] ;
```

A *`LockingClause`* attached to the *`MethodDeclaration`* is *inherited* by all `return` and `throw` statements by default. A `return` or `throw` statement may choose to override the inherited *`LockingClause`* by providing its own. If it wants to extend the inherited *`LockingClause`* it must use the optional + sign with its *`LockingClause`*.

3.2.8 Open atomic concurrency control

To define open atomic action concurrency control we introduce a conceptual device we call the *augmented log*. In addition to recording writes and open atomic clauses, the augmented log records

[13] Since release might be implemented in batch in a variety of ways, we do not specify the interface here. Since each lock is associated with a given transaction, and is held until the transaction completes, one always releases all of a transaction's locks at the same time.

reads of shared variables. An action's current read set is those variables that have a read record in the action's log, and its current write set is those variables that have an assignment record in the action's log. In the presence of open atomic actions, read and write sets can shrink as well as grow, as nested open atomic action commit and discard their related portion of the log. Beyond that, conflict is as in §3.1.2, with the addition of explicit locking specified in `locking` clauses and associated conflicts. (Notice that in this log-based view of concurrency control, the locks that an action holds are exactly those recorded in its log.)

3.2.9 Open atomic actions and **`new`**

When an atomic action aborts, what happens to objects it allocated? In the absence of open atomic actions, it is clear that no other action can have seen, or will see, the newly allocated objects, so there is no issue. However, in the presence of open atomic actions, an open atomic action can publish to a globally accessible variable a reference to an object allocated in a containing action. If the containing action aborts, and the published reference remains, to what does the reference refer? (The situation is similar to abrupt completion of constructors as discussed in the Java Language Specification (Sections 12.4 and 12.5 in the Third Edition).) We require that the compiler and run-time system guarantee that the reference refers to a type-safe instance (of the class indicated in the `new` expression). However, the instance may be partially or completely unconstructed, i.e., fields (including `final` fields) may have their default initial values. In other words, the situation may be as if the constructor has not yet run.

It is helpful if we consider instance creation to consist of *allocation* followed by *initialization* (constructor execution), as occurs in the Java Virtual Machine. We require that allocation be effectively an open atomic action. Constructor execution then proceeds with a type-safe instance of the class being allocated, each of its fields having the default value for their type. Thus, if a constructor aborts, it unwinds the instance to this default state. We observe that, as per the Java Language Specification, it is not a good idea to publish a reference before the referent is fully constructed.[14]

3.2.10 Concerning **`volatile`** and **`synchronized`**

Given the power of atomic actions, and open atomic actions in particular, there seems little additional value to `volatile` fields when used for synchronization. When used for applications such as access to memory-mapped I/O device registers, in the presence of atomic actions `volatile` fields may best be used within `oncommit` clauses. The same might be said concerning invocations of library routines and operating system calls.

Concerning Java `synchronized` blocks and methods, we believe that they, along with wait and notify support, can be implemented using open atomic actions in stylized ways. This would replicate their semantics faithfully. The same field should not be accessed in both atomic and synchronized code, since atomic code's undo, retry, and `oncommit` are somewhat unpredictable as to whether and when they occur.

In the long run, code using `synchronized` could be converted to `atomic` or `openatomic`. We note that `openatomic` can be used to build any ordering and signaling mechanism desired.

4. Example: An open atomic **`Map`**

We illustrate using the new open nesting features by fleshing out the example that served in our earlier work [17]. Listing 5 shows how an open atomic implementation of the `Map` interface can be defined as a concurrency-safe wrapper for unsynchronized `Map` implementations: `OpenMap` is declared as an `opanatomic` class implementing the `Map` interface, and permits safe concurrent access to the wrapped map, with `get`, `put`, `remove`, and `size` operations defined as `openatomic` methods.

Generally, `onabort` handlers are needed only for methods that mutate the abstract state of the map. The `put` operation 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.

The example uses three lock modes: `SHARED`, `EXCLUSIVE`, and `INTENTION_EXCLUSIVE`, with compatibility defined by their `conflictsWith` methods. Shared locks are compatible since multiple readers can operate on the same data item (i.e., `key`) at the same time. On the other hand, one cannot write a data item while it still has readers, nor 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—in this case the map itself: `this`. Thus, to `put`/`remove` an association for some `key` in the map requires an intention lock on the map as a whole (`this`). Two requests to `put`/`remove` an association for different keys do not conflict. However, to `put`/`remove` an association for any given key does conflict with requests that read the state of the map as a whole, such as the `size` operation. The necessary constraints are recorded for `put`/`remove` by acquiring an exclusive lock on the `key`, to prevent others from changing that association, along with an intention exclusive lock on `this` to prevent others needing shared mode access to the whole map (such as `size` requires).

5. STM implementation

Our implementation approach is similar to that of the McRT software transactional memory (STM) system [19]. McRT associates with each object (or word, the granularity being determined on a per-type basis) a *transaction record*. This record contains either a *version number* (for an object/word that does not have uncommitted writes) or a (pointer to) the *transaction descriptor* of the writing transaction. A transaction (atomic action) accumulates two lists of transaction records, one for items it reads (and the version number seen) and one for items it writes (including the old version number and the old value). It updates fields in place. When a transaction desires to commit, it must first *validate* its read set: each item must either contain a version number that is equal to what the read set recorded, or must point to the descriptor of the committing transaction (i.e., be later written by this transaction).

In the presence of nesting, open atomic actions commit by validating reads and installing new version numbers for written items. Commits of non-open atomic actions simply append their read and write set lists to those of the containing action, first updating written item transaction descriptors to refer to the parent (or we can introduce an additional level of indirection). They need not validate read sets, since the read sets need to be validated upon commit of an open atomic or top-level ancestor anyway. (Validating on nested action commit might detect conflicts earlier, but is extra work for successful transactions.)

We need an additional mechanism to group write entries so that appropriate batches of them are undone before invoking `onabort` clauses when undoing. This can be done by starting a new (closed) nested action after the commit of an open atomic action.

Our STM library's API is designed to support a range of possible STM implementations. Transactions read and write fields via accessor functions. We can change the code we generate for the

[14] It may also be useful to view constructors as being open atomic, with no `onabort` or `locking` clause, though adjustment may need to be made for their effects on other objects and on any static fields.

```
public openatomic class OpenMap implements Map {
  private final Map map;

  private
    LockTable<Points<Object>, LockModes> keyLocks
      = new LockTable<Points<Object>,
                    LockModes>(new Points<Object>());
  private
    LockTable<Points<Map>, LockModes> mapLock
      = new LockTable<Points<Map>,
                    LockModes>(new Points<Map>());

  public OpenMap(Map map) { this.map = map; }

  public Object get(Object key)
  locking (keyLocks:point(key):LockModes.SHARED)
  { return map.get(key); }

  public Object put(Object key, Object val)
  [ Object result; ]
  locking (
    keyLocks : point(key) : LockModes.EXCLUSIVE,
    mapLock : point(this) :
      LockModes.INTENTION_EXCLUSIVE)
  { return result = map.put(key, val); }
  onabort
  {
    if (result == null) {
      map.remove(key);
    } else {
      map.put(key, result);
    }
  }

  public Object remove(Object key)
  [ Object k = key; Object result; ]
  locking (
    keyLocks : point(key) : LockModes.EXCLUSIVE,
    mapLock : point(this):
      LockModes.INTENTION_EXCLUSIVE)
  { return result = map.remove(key); }
  onabort
  { if (result != null) map.put(k, result); }

  public int size()
  locking (
    mapLock : point(this): LockModes.SHARED)
  { return map.size(); }

  // ... other methods of the Map interface

  enum LockModes implements LockMode<LockModes> {
    SHARED {
      public boolean conflictsWith(LockModes m) {
        return m != SHARED;
      }
    },
    INTENTION_EXCLUSIVE {
      public boolean conflictsWith(LockModes m) {
        return m != INTENTION_EXCLUSIVE;
      }
    },
    EXCLUSIVE {
      public boolean conflictsWith(LockModes m) {
        return true;
      }
    }
  }
}
```

Listing 5: `OpenMap` class

accessors in order to deploy different strategies. Further, any given transaction must "open" an object before accessing it. An object may be opened for reading only, or for writing (and reading), and may be upgraded from reading to writing. Accessing an open for reading (writing) requires having the object open for reading (writing). Thus the "open" functions and the accessors are "hooks" that can be used to create almost any policy. The current prototype perform concurrency control on whole scalar objects and on chunks of arrays. Further, its atomicity strategy is to update in place, saving previous values in a write log, and to undo when necessary.

6. XJ: A portable reference implementation

Our portable XJ reference implementation has three primary components: an extended Java source code to class file compiler based on OpenJDK's `javac`, a run-time bytecode rewriting tool implemented as a JVMTI agent, and the XJ run-time library. The XJ compiler accepts our extended syntax for closed and open atomic actions, performs compile-time checks, and compiles the language extensions to standard Java class files. Most of the work it does is to provide the structure needed to represent atomic actions in Java bytecode, supporting transaction abort and retry, and to represent the suffix clauses of open nested actions. It provides only a framework for atomic action execution, with the remaining support for transactions injected into the bytecode by the JVMTI agent to make the transformations complete. The bytecode is then ready to run with the XJ library that keeps track of the transactions as they execute and supports the run-time functionality for concurrency control, and transaction abort and roll-back.

6.1 XJ compiler

Our implementation of the XJ compiler is based on version 1.7.0-ea-b19 of OpenJDK's `javac`. This has been extended to accept the new XJ syntax and generate compliant Java bytecode that will run on any standard Java virtual machine (though transaction support comes only when combined with the XJ run-time rewriter and XJ run-time library). We modified the parser to accept the new syntax, the annotation processor to statically check the new constructs, the abstract syntax tree (AST) to represent handlers and lock expressions, and the lowering phase to transform the high-level XJ constructs into a standard Java AST. We had no need to modify the bytecode generation parts of `javac`.

We focus our explanation of the compile-time transformations on those needed for open nested methods, which subsume those for closed atomic methods and blocks, illustrated for the `remove` method of the `OpenMap` example shown in Listing 5. The XJ compiler produces Java bytecode for this method equivalent to the Java source shown in Listing 6, as follows.

- The `PreDecls` in a `MethodDeclarator` transform into local variable declarations in the method body, allowing the capture of state at the beginning of the open nested action, as seen at line 3.
- Lock expressions can be inherited by overriding methods. To facilitate this we transform lock expressions into protected methods of a class and invoke the method at the point the lock needs to be acquired (line 45).
- The body of the method moves into a collapsed version of the method as seen at line 42.
- Suffix clauses are encapsulated as anonymous instances of inner classes that capture their unbound variables from the enclosing scope as final variable declarations as described in detail below.

6.1.1 Handlers on open atomic methods

If an open atomic method runs to completion then its handlers need to be logged. In the case that it fails it must retry from the beginning. To allow retry we wrap the method body in a `try`/`finally` block (line 6 of Listing 6) nested within an infinite loop (line 5). The outer `try`/`finally` block is used to detect the successful completion of the method. In the case that it does complete successfully we then create a new instance of a `TxnHandler` class overriding the corresponding method defined in the handler, and log the handler (line 25). We then commit this open atomic transaction. If a `TxnException` occurs while trying to log the handler, we abort the transaction and retry it from the beginning. Inside of the main `try`/`finally` block is another nested `try`/`catch` block. This is used to run the corresponding collapsed method body (line 12). Prior to running the method body we create a new nested transaction (line 8). If the collapsed method throws a `TxnException` we abort the transaction and retry the transactional method. In the case where an `Error` is thrown we abort the transaction and throw the `Error`. In either of these cases we avoid logging the handlers; they are logged only when the method completes successfully.

It is possible for a constructor of an open atomic class to have handlers associated with it. The transformation described above cannot be applied to constructors directly because the first statement in a constructor should be a call to a superclass constructor or another constructor in the current class. To get around this issue we use a two phase transformation for constructors of open atomic classes. The first phase is done by the XJ compiler while the bytecode instrumenter completes the second phase. The XJ compiler leaves the `super` or `this` call as it is in the constructor (even if it has complex expressions as arguments to the other constructor) and moves the rest of the statements to a `get_$init` method. A call to this `get_$init` method is added to the constructor. The transformation done in the second phase is explained in Section 6.2.3.

6.2 Bytecode instrumenter

To add transaction support to classes we adopt an approach similar to that of our previous work in transparent distribution for Java [13], allowing mediation of all accesses to static and instance fields, as well as elements of arrays. The transactional machinery needed by objects (the lock word, etc.) reside in instances of `TxnObject`. Ideally, all objects that are going to be read from or written to inside a transaction extend this class. Also reads and writes inside transactional methods and transactional blocks need to be logged. We accomplish this by instrumenting classes at run time. The instrumentation that needs to be performed on a class depends on the classification of the class. We divide classes into two categories, *direct* classes and *wrapped* classes. Direct classes are ones that can be transformed to inherit from `TxnObject` and on which our rewrites can be performed directly. Figure 1a shows the manner in which direct classes are transformed.

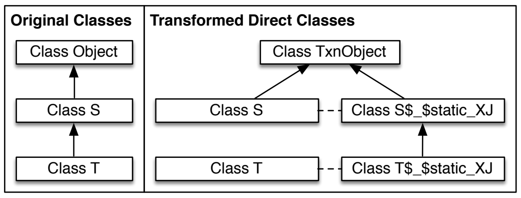
There are a few classes in the JVM that cannot be rewritten directly in this ideal manner. The JVM has intimate knowledge of these classes; e.g., the offsets of fields in these classes are hard coded into the JVM (`java.lang.ref.SoftReference` in Oracle's Hotspot JVM is an example), thus they cannot extend `TxnObject`. In order to get the transactional machinery into these classes we *wrap* them. We also wrap all classes that have native methods.[15] The manner in which wrapped classes are transformed is shown in Figure 1b.

We preprocess all classes used by the application prior to running it. Preprocessing helps us classify classes beforehand. The process used is similar to that of McGachey et al. [13].

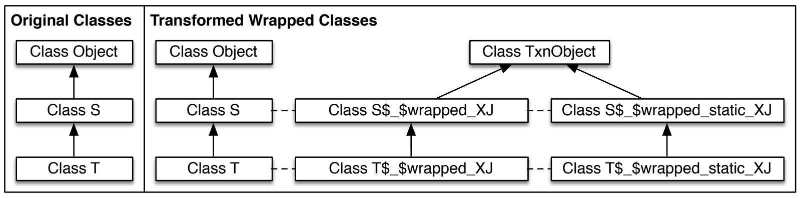
[15] This is safe, but it may not always be necessary, depending on how JNI libraries are coded.

```
 1 public Object remove(Object key) {
 2   TxnDescriptor _$current_desc = null;
 3   Object k = key;
 4   boolean _$succeed = true;
 5   while (true) {
 6     try {
 7       _$succeed = true;
 8       TxnDescriptor.beginOpen(
 9         _$current_desc);
10       try {
11         return
12           this.remove_$collapsed(key,
                  _$current_desc);
13       } catch (TxnException ex) {
14         TxnDescriptor.abortOpen(_$current_desc);
15         _$succeed = false;
16         continue;
17       } catch (Error ex) {
18         TxnDescriptor.abortOpen(_$current_desc);
19         _$succeed = false;
20         throw ex;
21       }
22     } finally {
23       if (_$succeed) try {
24         final Object _$k = k;
25         _$current_desc.getOpenLog()
26             .logHandler(new TxnHandler() {
27           public void _$abort() {
28             if (result != null) {
29               map.put(_$k, result);
30             }
31           }
32         }, TxnHandler.ON_ABORT_HANDLER);
33         TxnDescriptor.commitOpen(_$current_desc);
34       } catch (TxnException ex) {
35         TxnDescriptor.abortOpen(_$current_desc);
36         continue;
37       }
38     }
39   }
40 }
41
42 private Object remove_$collapsed(
43   Object key, TxnDescriptor _$current_desc)
44 {
45   this.remove_$locking(
46     key,_$current_desc);
47   result = map.remove(key);
48   return result;
49 }
50
51 protected void remove_$locking
52   (Object key, TxnDescriptor _$current_desc)
53 {
54     LockSpace space;
55     Lock shape;
56     space = keyLocks.getSpace();
57     shape = ((Points) space).point(key);
58     keyLocks.add(
59       shape, LockModes.EXCLUSIVE, _$current_desc);
60     space = mapLock.getSpace();
61     shape = ((Points) space).point(this);
62     mapLock.add(
63       shape, LockModes.INTENTION_EXCLUSIVE,
64       _$current_desc);
65 }
```

Listing 6: Transformed `remove` method of `OpenMap`

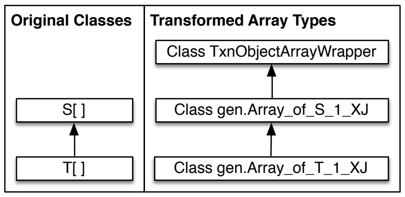


(a) Direct



(b) Wrapped

Figure 1: Class transformations



Original Classes
Transformed Array Types
Class TxnObjectArrayWrapper
S[ ]
T[ ]
Class gen.Array_of_S_1_XJ
Class gen.Array_of_T_1_XJ

Figure 2: Array type transformations

6.2.1 Statics

Object locking in XJ is done via a lock field in `TxnObject`. This mechanism does not work for static fields, since they are not part of an object. In order to use the same transactional machinery on static fields we move the static fields and static methods of a class to a generated class, where they become instance fields and instance methods. We also generate get/set methods for these moved fields. We guarantee that there is only one instance of this generated class, which we call the *static singleton* of the original class. The static singleton is initialized via the static initializer of the generated class. When a static singleton is initialized it also initializes its superclass, which would be the static singleton of the original class's superclass. Each static singleton class has a static `get_$singleton` method to get the single instance. The instrumenter rewrites `getstatic` and `putstatic` bytecodes to first obtain the corresponding singleton for the field and then invoke the appropriate get/set method on it. `invokestatic` is also rewritten such that the invocation is on the static singleton instance.

6.2.2 Arrays

We generate special "array classes" for array types. This helps us get the transactional machinery into arrays. Arrays do not use the same locking mechanism used by scalar objects. Having a single lock word for the whole array would not perform well. Instead, we allow customizing of the lock scheme used on arrays, having a lock for each element or a lock for each portion of the array. The `TxnArray` interface defines the API for obtaining locks on an array. The XJ run-time library provides wrappers for each primitive array type and for the object array type. Each generated array class extends one of these wrapper classes, enabling it to gain access to the transaction machinery. The structure of the generated array classes is similar to that of McGachey et al. [13] for arrays. Figure 2 shows the transformation for array types.

6.2.3 Object creation

As mentioned before, constructors go through a two-phase transformation. The second transformation is performed by the instrumenter. The purpose of this transformation is to move all the code from inside the constructor to the `get_$init` method. We do this by adding a dummy constructor to each class. The dummy constructor is used purely for object creation. This enables us to create an empty object for a given class. We then transform the constructors such that any call to a superclass constructor is replaced with a call to the dummy constructor in the superclass. Also, within the corresponding `get_$init` method for that constructor we insert a call to the corresponding `get_$init` method of the superclass constructor. This transformation enables us to create an empty object first, and then run all the code of the constructor within the boundary of a transaction.

6.2.4 Java agent

Dynamic code rewriting is performed via a Java agent using the Java Virtual Machine Tool Interface (JVMTI). In order to rewrite all Java classes (including those loaded by the bootstrap class loader) the agent creates a separate operating system process for running the instrumenter. We use the ASM library [3] for instrumenting Java classes. The instrumenter process is created in the `Agent_OnLoad` function. The agent uses the `ClassFileLoadHook` callback to intercept classes loaded by the JVM. Intercepted classes are then presented to the instrumenter. The agent communicates with the instrumenter via pipes. The result of this instrumentation process could be a single class or multiple classes. If the result is a single class, the agent returns the bytes received from the instrumenter as the bytes of the instrumented class. If the result consists multiple classes, the action taken by the agent depends on the class loader of the class being loaded. If the original class is loaded by the bootstrap class loader, any additional class files are written to a special directory which happens to be on the bootstrap class loader's class path (set via the `-Xbootclasspath` VM option). If the original class is not loaded by the bootstrap class loader, any additional class files are injected into the VM via the `DefineClass` JNI function. In both cases the bytes of the original loaded class are replaced by the rewritten bytes. Calling the `DefineClass` function on the additional classes inside the agent causes those class definitions to be intercepted again (because of the `ClassFileLoadHook`), but there is no need to call the instrumenter for them because the agent already has their instrumented versions. To support this functionality the agent keeps a local cache for any additional class files obtained from the instrumenter (if not loaded by the bootstrap class loader) and the agent passes an empty byte array to the `DefineClass` function. When the agent intercepts the loading of any class, it first checks if the class already has an instrumented version in the local cache, and if so, it uses that version instead of invoking the instrumenter and then removes the class from the cache. Otherwise it sends the class to the instrumenter for instrumentation as usual.

6.2.5 Instrumenter process

The instrumenter runs in a infinite loop polling for messages by the agent. The first byte of each message from the agent is a code indicating the action requested from the instrumenter. This control byte indicates the class loader of the object (the bootstrap class loader or not), or that the VM has been initialized or is being shut down. Once a request is received for instrumenting a class, the instrumenter performs rewrites based on its classification. The preprocessed information is used to determine the classification of a class. Although we divide classes into two categories, the general rewrites we perform on individual elements of these classes are similar. We now describe those rewrites.

- Generate a static singleton for the given class
- Generate accessor classes for each field in a class. The accessor classes are used for logging reads/writes as explained in Section 6.2.6
- The first rewrite we do is to redirect to newly generated types. This includes redirecting to wrapped versions of objects and rewriting `getstatic`, `putstatic`, and `invokestatic` to refer to static singletons. We also redirect to the newly generated array and accessor classes
- Transform constructors as described in Section 6.2.3
- Create transactional versions of all methods. This is done by duplicating methods and adding a `TxnDescriptor` as the last parameter to the method. We also add logging to reads and writes (`getfield`, `putfield`, array loads and stores) in this method.

6.2.6 Accessor objects

We generate accessor classes for each field of a class; each extends `org.ruggedj.xj.xjrt.runtime.Accessor`, which has a single abstract method `restoreField` used by the run-time library to perform undo operations. It takes a `TxnWriteLog` as an argument and returns `void`. The generated accessor class also has a `set` method for setting the value of the field and a `get` method corresponding to its data type for getting the value of the field. The `set` method pushes the object being updated into the write log along with the accessor instance and the value been written. It also sets the value of the field in the object. The corresponding `get` method pushes the current object into the read log along with the value being read, and returns that value.[16] The generated accessor class instances are created in the static initializer of a class and held in new static final fields. During the instrumentation phase, `getfield`, `putfield`, `getstatic`, and `putstatic` bytecodes are rewritten to use the accessor object for setting and getting a field. The `restoreField` method pops the object from the write log and then pops a value of the corresponding data type from it (one of the primitive types or `Object`). It then sets the field of the class to the popped value (cast to the field's declared type), restoring its value. The run-time library provides accessor classes for array types, one for each primitive array type, and one for object arrays, so these do not need to be generated for each type.

6.3 Run-time library

The XJ run-time library provides the underlying support needed to manage transactions at run time. It consists of logs needed for logging reads, writes, and handlers (for open atomic transactions), which capture handlers and abstract locks, and it supports conflict detection and rollback. It also consists of key classes such as `TxnObject`, which holds the lock field for objects, `TxnArrayWrapper` which holds the locks for array classes, and `TxnDescriptor`, which represents a top-level transaction and its subtransactions. The lock field in `TxnObject` is represented as an `int` with its low 3 bits indicating a mode. If the mode is 000, the value stored in the lock field is a version number; mode 001 indicates that the value stored is the id of a transaction that has locked the object. The other modes are reserved for future use. All classes used within a transaction are expected to extend `TxnObject` (directly or indirectly). The methods for acquiring and releasing locks lie in the `TxnObject` class. This enables us to lock all objects used (written) within a transaction. This also implies that we obtain locks on a per-object basis.[17] `TxnArrayWrapper` has an `int` array whose elements are used as locks. As mentioned before, depending on the configuration a lock can be allocated per element or per group of elements.

The `TxnDescriptor` class represents transactions in our system. It contains an `int` field used as the unique id of the transaction, a read log (for logging reads (if needed by the transaction management protocol)), a write log (for logging writes), and an "open" log (for logging handlers of open atomic actions). It also contains a stack for keeping track of nested transactions. Note that we do not create a new `TxnDescriptor` for each nested transaction. Rather, we use the same `TxnDescriptor` for all nested transactions with the `TxnDescriptor` instance keeping track of nesting and the state of the transaction. `TxnDescriptor` provides methods for obtaining the logs and performing transactional operations (begin, commit, abort) for both closed and open atomic transactions. Each method running inside a transaction has access to the current transaction descriptor via an argument to the method that is injected by the bytecode instrumenter.

7. Future work

Our work is not finished. In particular, we will shortly have the implementation tested and sufficiently functional to begin exploring its performance characteristics, as well as opportunities to improve performance, as others have done for closed nested transactions [2]. Naturally, our prototype approach will likely suffer from lack of assistance from modifications to the underlying Java virtual machine. We look forward to exploring what support the JVM can provide via extended bytecodes or internal logging of variable reads/writes. Ultimately, the JVM can also take advantage of recently-available hardware transactional memory support to further reduce execution overheads. We have recently also completed extensions to ASM in support of generalized forward and backward data flow analyses to support rewrite-time optimization of the placement of logging and locking operations for transactions and removal of redundant operations.

We look forward to feedback from others regarding the flexibility and utility of open nested transactions as a means to gaining improved concurrency for applications built using XJ.

Acknowledgments

This work is supported in part by NSF grants CNS-1161237 and CNS-1162246.

References

[1] Y. Afek, U. Drepper, P. Felber, C. Fetzer, V. Gramoli, M. Hohmuth, E. Riviere, P. Stenstrom, O. Unsal, W. Maldonadao Moreira, D. Harmanci, P. Marlier, S. Diestelhorst, M. Pohlack, A. Cristal, I. Hur, A. Dragojevic, R. Gerraoui, M. Kapalka, S. Tomic, G. Korland, N. Shavit, M. Nowack, and T. Riegel. The Velox transactional memory stack. *IEEE Micro*, 30(5):76–87, Sep./Oct. 2010. doi: `10.1109/MM.2010.80`.

[2] Y. Afek, G. Korland, and A. Zilberstein. Lowering STM overhead with static analysis. In *International Workshop on Languages and*

[16] Our current system records only a version number of the whole instance, but the API allows for a wide range of transaction management strategies.

[17] Our use of accessors would support per-field locking; it is the run-time library that does not offer that functionality at present.

Compilers for Parallel Computing, volume 6548 of *Lecture Notes in Computer Science*, pages 31–45, Houston, Texas, Oct. 2010. Springer. doi: `10.1007/978-3-642-19595-2_3`.

[3] E. Bruneton. *ASM 4.0: A Java bytecode engineering library*, Sept. 2011. URL `http://download.forge.objectweb.org/asm/asm4-guide.pdf`. Version 2.0.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1–13, Ottawa, Canada, June 2006. doi: `10.1145/1133981.1133983`.

[5] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Anaheim, California, 2003 2003. doi: `10.1145/949305.949340`.

[6] 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`.

[7] M. Herlihy and E. Koskinen. Composable transactional objects: A position paper. In *European Symposium on Programming*, volume 8410 of *Lecture Notes in Computer Science*, pages 1–7, Grenoble, France, Apr. 2014. Springer. doi: `10.1007/978-3-642-54833-8_1`.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 84–91, Boston, Massachusetts, 2003. doi: `10.1145/872035.872047`.

[9] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 253–262, Portland, Oregon, Oct. 2006. doi: `10.1145/1167473.1167495`.

[10] 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`.

[11] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *ACM SIGPLAN Workshop on Memory System Performance and Correctness*, pages 82–91, San Jose, California, Oct. 2006. doi: `10.1145/1178597.1178611`.

[12] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Workshop on Programmability Issues for Heterogeneous Multicores*, Pisa, Italy, Jan. 2010.

[13] P. McGachey, A. L. Hosking, and J. E. B. Moss. Class transformations for transparent distribution of Java applications. *Journal of Object Technology*, 10:9:1–35, Aug. 2011. doi: `10.5381/jot.2011.10.1.a9`.

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

[15] 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`.

[16] Multiverse. URL `http://multiverse.codehaus.org`.

[17] 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`.

[18] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979. doi: `10.1145/322154.322158`.

[19] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, Mar. 2006. doi: `10.1145/1122971.1123001`.

[20] ScalaSTM. URL `http://nbronson.github.io/scala-stm`.

[21] Transactional Memory Specification Drafting Group. Original draft specification of transactional language constructs for C++. Technical report, Programming Language C++, EWG, SG5 Transactional Memory, Feb. 2012. Version 1.1.