Automatic Scalable Atomicity via Semantic Locking

Guy Golan-Gueta
Yahoo Labs, Israel
ggolan@yahoo-inc.com

G. Ramalingam
Microsoft Research, India
grama@microsoft.com

Mooly Sagiv
Tel Aviv University, Israel
msagiv@tau.ac.il

Eran Yahav
Technion, Israel
yahave@cs.technion.ac.il

Abstract

In this paper, we consider concurrent programs in which the shared state consists of instances of linearizable ADTs (abstract data types). We present an automated approach to concurrency control that addresses a common need: the need to *atomically* execute a code fragment, which may contain multiple ADT operations on multiple ADT instances.

We present a synthesis algorithm that automatically enforces atomicity of given code fragments (in a client program) by inserting pessimistic synchronization that guarantees atomicity and deadlock-freedom (without using any rollback mechanism). Our algorithm takes a *commutativity specification* as an extra input. This specification indicates for every pair of ADT operations the conditions under which the operations commute. Our algorithm enables greater parallelism by permitting commuting operations to execute concurrently.

We have implemented the synthesis algorithm in a Java compiler, and applied it to several Java programs. Our results show that our approach produces efficient and scalable synchronization.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Automatic synchronization, Transactions, Semantics

1. Introduction

A key challenge in writing concurrent programs is concurrency control: ensuring that concurrent accesses and modifications to shared mutable state do not interfere with each other in undesirable ways. Solutions used in practice are predominantly handcrafted and based on locks. However, this is a tedious and error-prone process which may result in safety violations and deadlocks (e.g., see [22]).

Atomic sections [11] are a language construct that allows a programmer to declaratively specify that a given code fragment must (appear to) execute atomically, leaving it to a compiler and runtime to implement the necessary concurrency control. However, existing approaches to implementing atomic sections have not been widely adopted due to various concerns, including high runtime overhead, and limited applicability (e.g., see [8]). We address these concerns by presenting a novel approach for realizing a restricted form of atomic sections in which the shared mutable state consists of several linearizable [15] *abstract data types* (ADTs).

```
1  atomic {
2    set=map.get(id);
3    if(set==null) {
4      set=new Set(); map.put(id, set);
5    }
6    set.add(x); set.add(y);
7    if(flag) {
8      queue.enqueue(set);
9      map.remove(id);
10   }
11 }
```

Figure 1. An atomic section that manipulates several linearizable ADTs — a Map, a Set, and a Queue. This example is inspired by the code of *Intruder* [7], further discussed in Section 6.

```
1  atomic { map.lock({get(id),put(id,*),remove(id)});
2    set=map.get(id);
3    if(set==null) {
4      set=new Set(); map.put(id, set);
5    }
6    set.lock({add(*)}); set.add(x); set.add(y);
7    if(flag) { queue.lock({enqueue(set)});
8      queue.enqueue(set); queue.unlockAll();
9      map.remove(id);
10   }
11   map.unlockAll(); set.unlockAll();
12 }
```

Figure 2. The atomic section of Fig. 1 with semantic locking operations automatically inserted by our compiler.

Example The example in Fig. 1, inspired by the code of *Intruder* [7], illustrates the problem we address in this paper. The shared (global) state of this code fragment consists of three ADTs: (i) a Map ADT (pointed by the variable `map`); (ii) a Set ADT (pointed by the variable `set`); (iii) and a Queue ADT (pointed by the variable `queue`). (All program variables, such as `flag`, are thread-local.) Each of these ADTs is linearizable [15], and thus each individual ADT operation appears to execute atomically (even

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3205-7/15/02...$15.00
http://dx.doi.org/10.1145/2688500.2688511

in the presence of concurrent operations). However, in this case, we wish the entire code fragment to execute atomically: the individual ADTs cannot provide this guarantee.

The Problem We consider a Java multi-threaded program (also referred to as a *client*), which makes use of several linearizable ADTs. We assume that the only mutable state shared by multiple threads is *instances* of ADTs. We permit atomic sections as a language construct: a block of code may be marked as an atomic section (as shown in Fig. 1). An execution of an atomic section is called a *transaction*. Our goal is to ensure that transactions appear to execute atomically and make progress (avoiding deadlocks), while exploiting the semantic properties of the ADT operations to achieve greater parallelism. We also wish to avoid the use of any rollbacks (i.e., speculation is not permitted).

Our Approach Our approach is based on locking which utilizes semantic properties of the shared ADTs. In this approach, the transactions acquire locks on ADT operations (rather than acquiring locks on memory locations or data): a transaction is permitted to invoke an operation, only when it holds a lock on that operation. The meaning of the locks is based on the semantics of the operations, that is, two transactions are allowed to simultaneously hold locks on operations op_1 and op_2, respectively, only if op_1 and op_2 are commutative operations.

Automatic Atomicity We present a compiler for atomic sections. Given a client program with atomic sections, the compiler guarantees atomicity and deadlock-freedom of these atomic sections by inserting code that acquires locks on operations of the shared ADTs. In order to implement the locks, for each ADT, our compiler takes a *commutativity specification* as an extra input. This specification indicates for every pair of ADT operations the conditions under which the operations commute. For example, Fig. 3(b) shows a commutativity specification for a Set ADT. Commutativity specifications are discussed in Section 5.

Fig. 2 shows the locks inferred for the atomic section of Fig. 1: in the shown code, operations of the shared ADTs are locked by using two special methods: `lock` and `unlockAll`. The meaning and the implementation of these methods are described later.

Key Features and Advantages Our approach offers the following key features and advantages:

- *Greater parallelism via semantic conflict detection*. This approach permits transactions to concurrently execute commuting operations, even though the underlying operation implementation may *access and modify the same locations*. E.g., two concurrent executions of the code fragment in Fig. 1 can execute the `add` operation on the *same* Set concurrently, because `add` operations on a Set commute.
- *No rollback*. This approach does not use any rollback mechanism [11], and enables the use of irrevocable operations (as demonstrated in Section 6.2).
- *Modularity and compositionality*. This approach is decoupled from the *implementation* of the shared ADTs. In particular, it allows the different ADT implementations to use their own concurrency control solutions internally. In a sense, our approach can be seen as a simple way to compose several unrelated linearizable ADTs.

Fine-Grained Synchronization Our compiler aims to produce fine-grained synchronization (for the sake of better potential parallelism) by trying to handle each shared Java object as an independent ADT with a dedicated synchronization mechanism. We show that this is not always possible for programs with dynamic pointer updates — for such programs, our compiler handles several Java objects as a single global ADT (Section 3.4).

2. Semantic Locking

In this section we introduce some terminology, describe our methodology for realizing atomic sections using semantic locking, and formalize the problem addressed in the subsequent sections.

2.1 Basics

Clients A *client* is a concurrent program that satisfies the following restrictions. All state shared by multiple threads is encapsulated as a collection of *ADT instances*. (The notion of an ADT is formalized later.) The shared mutable state is accessed only via ADT operations. The language provides support for *atomic sections*: an atomic section is simply a block of sequential code prefixed by the keyword `atomic`. Shared state can be accessed or modified only within an atomic section. We will use the term *transaction* to refer to the execution of an atomic section within an execution.

In the simpler setting, a client is a whole program (excluding the ADT libraries). More generally, a client can be a module (library) or simply a set of atomic sections. However, we assume that all atomic sections accessing the shared state are available.

ADTs An *abstract data type* (ADT) encapsulates state accessed and modified via a set of methods. Statically, it consists of an *interface* (also referred to as its API) and a *class* that implements the interface. The implementation is assumed to be *linearizable* [15] with respect to the sequential specification of the ADT. We also assume that its object constructor is a pure-method [23].

We will use the term *ADT instance* to refer to a runtime Java object that is an instance of the ADT class. We will abbreviate "ADT instance" to just ADT if no confusion is likely. Two different ADT instances can have no shared state. Every ADT instance is assumed to have an unique identifier (such as its memory address).

Global ADTs In some cases, our compiler cannot treat each shared object as an independent ADT: in those cases, we treat several runtime objects as a single global ADT (such an ADT has a single instance). The discussion about global ADTs appears in Section 3.4.

Operations We use the term *operation* to denote a tuple consisting of an ADT method name m and runtime values $v_1, \cdots, v_n$ for m's arguments (not including the ADT instance on which the operation is performed), written as $m(v_1, \ldots, v_n)$. An operation represents an invocation of a method on an ADT instance (at runtime). For example, we write `add(7)` to denote the operation that represents an invocation of the method `add` with the argument 7.

Methodology Our goal is to realize an implementation of the atomic sections in clients, using locking (pessimistic concurrency control). Our approach is realized by the compiler in two steps:

1. In the first step (Sections 3 and 4), we assume that each ADT provides synchronization API that enables locking and unlocking its operations (for example, in Fig. 2 at line 6, the `add` operations of the Set are locked by using this API). Using this assumption, our compiler enforces atomicity by inserting calls to this synchronization API.
2. In the second step (Section 5), the compiler implements the calls to the synchronization API by utilizing semantic properties of the ADT operations. This is realized by using commutativity specifications that are given to the compiler.

2.2 ADTs with Semantic Locking

An *ADT with semantic locking* is an ADT that provides a *synchronization API*, in addition to its standard API, that allows a transaction to lock (and unlock) operations (on ADT instances). We start by assuming that all shared ADTs already provide a synchronization API, in Section 5 we show how this API is implemented by our compiler.

```
void add(int i);
void remove(int i);
boolean contains(int i);
int size();
void clear();
```

(a)

	add(v')	**remove**(v')	**contains**(v')	**size**()	**clear**()
add(v)	true	$v \neq v'$	$v \neq v'$	false	false
remove(v)		true	$v \neq v'$	false	false
contains(v)			true	true	false
size()				true	false
clear()					true

(b)

Figure 3. API of a Set ADT and its commutativity specification. The Set's methods are described in (a). Its commutativity specification is shown in (b): for every pair of operations, the specification indicates a condition under which the operations commute (e.g., in the shown specification the operations `add`(v) and `remove`(v') are commutative when $v \neq v'$).

2.2.1 Synchronization API

The synchronization API of an ADT contains two methods `lock` and `unlockAll`: we refer to invocations of these methods as *locking operations*. We refer to invocations of the other methods as *standard operations* (since they belong to the ADT's standard API).

The method `lock` An invocation of the method `lock` is meant to be used (by a client transaction) to acquire (permission to invoke) certain standard operations. We think of such invocation as corresponding to a lock on a *set of standard operations* (on the corresponding ADT instance).

Statically, a call to a `lock` method is expressed using *symbolic operations* and *symbolic sets* [9]. Let Var be a set of the program's variables, and $*$ be a symbol such that $* \notin$ Var. A *symbolic operation* (over Var) is denoted by $p(a_1, \cdots, a_n)$, where p is a method name, and each $a_i \in \text{Var} \cup \{*\}$. A *symbolic set* is a set of symbolic operations. The method `lock` receives a symbolic set as a parameter.

EXAMPLE 2.1. *In Fig. 2, line 1 contains a call to the Map's `lock` method where the parameter is the symbolic set* $\{get(id), put(id, *), remove(id)\}$. *Line 6 contains a call to the Set's `lock` method where the parameter is the symbolic set* $\{add(*)\}$.

The meaning of a call to `lock` is defined using the runtime values of the programs variables. Let *Value* be the set of possible values (of the variables in Var). Given a function σ : Var $\mapsto$ *Value* and a symbolic set *SY*, we define the set of operations $[\![SY]\!](\sigma)$ to be

$$\bigcup_{p(a_1,\ldots,a_n)\in SY} \{p(v_1, \ldots, v_n) \mid \forall i.(a_i \neq *) \Rightarrow (v_i = \sigma(a_i))\}.$$

Notice that, the $*$ symbol is used to refer to all possible values.

Let "`lock(`*SY*`)`" be a call to `lock` where *SY* is a symbolic set. A runtime invocation of "`lock(`*SY*`)`" locks all operations in $[\![SY]\!](\sigma)$ where σ maps each variable to its runtime value.

EXAMPLE 2.2. *Consider the invocations of `lock` in Fig. 2. When* $id = 7$, *the call `lock(`*$\{get(id), put(id, *), remove(id)\}$*`)` , at line 1, locks the operations in the set* $\{get(7), put(7, v), remove(7) \mid v \in$ *Value*$\}$ *(i.e., all invocations of `get`, `remove` and `put` for which the first parameter is 7). The call `lock(`*$\{add(*)\}$*`)`, at line 6, locks all the operations in the set* $\{add(v) \mid v \in$ *Value*$\}$ *(i.e., all invocations of `add`). As will be explained below, in both cases, each of these `lock` invocations acquires permission to invoke the corresponding operations.*

The method `unlockAll` The method `unlockAll` is simpler, its invocation unlocks all the ADT operations that are held by the current transaction. For example, in Fig. 2 at line 11, the code unlocks all the Map and Set operations that have been locked in the preceding lines.

2.2.2 Semantic Guarantees of the Synchronization API

We now describe the semantic guarantees of the synchronization API. We first formalize the notion of commutativity of operations. Two operations are said to be *commutative* if applying them to the same ADT instance in either order leads to the same final ADT state (and returns the same response).

EXAMPLE 2.3. *Consider a Set ADT with the API described in Fig. 3(a) (the Set has a standard meaning). For this ADT, the operations `add(7)` and `remove(7)` are not commutative. In contrast, the operations `add(7)` and `remove(10)` are commutative.*

The synchronization API is required to satisfy the following guarantee: no two transactions are allowed to concurrently hold locks on non-commuting operations (on the same ADT instance). Specifically, if a transaction t holds locks on the operations in the set O_t for an ADT instance A and, at the same time, a different transaction t' holds locks on the operations in the set $O_{t'}$ for the same ADT instance A, then every operation in O_t must commute with every operation in $O_{t'}$.

This means that the implementation (of the ADT's synchronization API) must block, whenever necessary, to ensure the above requirement. That is, if a transaction t holds locks on the operations in O_t, and a transaction t' tries to lock the operations in $O_{t'}$ where some operation in O_t does not commute with some operation in O'_t, then t' waits (blocked) until it is legal (for t') to hold locks on all operations in $O_{t'}$.

EXAMPLE 2.4. *Consider a Set ADT with the API described in Fig. 3(a) (the Set has a standard meaning). For this ADT, a transaction should not be allowed hold a lock on the set* $\{size(), clear()\}$ *while another transaction holds a lock on* $\{add(v) \mid v \in$ *Value*$\}$, *because (for example) `size()` does not commute with the `add` operations. However, it is legal to permit two different transactions to simultaneously hold locks on the set* $\{add(v) \mid v \in$ *Value*$\}$, *because `add` operations commute with each other.*

2.3 Synchronization Protocol

We now describe the synchronization protocol used by our compiler.

The S2PL Protocol Our synchronization is based on a *semantics-aware two-phase locking protocol* (S2PL) [6]. We say that an execution π follows S2PL, if the following conditions are satisfied by each transaction t in π:

- t invokes a standard operation p of an ADT instance A, only if t currently holds a lock on operation p of A.
- t locks operations only if t has never unlocked operations.

An execution that satisfies S2PL is a *serializable execution* [6]. Such execution is equivalent to an execution in which no two transactions are interleaved. Therefore, a serializable execution in which all transactions are completed can be seen as an execution in which all transactions are executed atomically.

```
void f(Set x, Set y) {
 x.lock({size()});
 int i = x.size();
 y.lock({add(i)}); x.unlockAll();
 y.add(i);
 y.unlockAll();
}
```

Figure 4. Code that follows the S2PL protocol.

EXAMPLE 2.5. *Consider a transaction t that executes "$f(s_1, s_2)$" where f is the function shown in Fig. 4, and s_1, s_2 are two different Sets. This transaction follows the S2PL rules.*

The S2PL protocol enables substantial parallelism. Consider two transactions t and t' that execute "$f(s_1, s_2)$". In this case, all operations locked by t commute with all operations locked by t' (even though they work on the same ADT instances). Hence, it is possible for the two transactions to run in parallel, while guaranteeing serializability.

The OS2PL Protocol The S2PL protocol does not guarantee deadlock-freedom — in order to avoid deadlocks we use the *Ordered S2PL Protocol* (OS2PL). We say that an execution follows the OS2PL protocol if the execution follows S2PL and satisfies the following additional condition:

- There exists an irreflexive and transitive relation $\sqsubset$ on ADT instances such that if a transaction t locks operations of ADT instance A after it locks operations of ADT instance A', then $A' \sqsubset A$.

The rule requires that ADT operations be locked according to a consistent order on the ADT instances. Notice that A and A' may represent the same ADT instance in the above rule. Hence, the rule implies that a transaction should not invoke multiple locking operations on the same ADT instance. Following this rule ensures that an execution cannot reach a deadlock.

3. Automatic Atomicity Enforcement

In this section, we present the basic version of our algorithm. The presented algorithm inserts semantic locking operations into the atomic section to ensure that every transaction follows Ordered S2PL.

In this section, when the algorithm infers a call "`lock(SY)`" of an ADT A, the symbolic set *SY* contains all A's operations (i.e., for any given σ, the set $[\![SY]\!](\sigma)$ contains all A's operations). Such symbolic set is created by using a set with all A's methods, where all the method arguments are $*$ — e.g., for the Set ADT, mentioned in Fig. 3(a), this symbolic set is "$\{add(*), remove(*), contains(*), size(), clear()\}$". For brevity, in this section, we denote such a call by writing "`lock(+)`". Fig. 17 shows the code produced by the algorithm for the atomic section presented in Fig. 1. In essence, this algorithm uses a locking granularity at the ADT instance level: two transactions cannot concurrently invoke operations on the same ADT.

In Section 4, we show a way to permit more fine-grained concurrency by refining the symbolic sets that are passed to the `lock` methods. In Section 5 we show how these symbolic sets are translated to synchronization code.

In this section, we say that an ADT instance A is *locked* by transaction t if the operations of A are locked by t.

3.1 Enforcing S2PL

Ensuring that all transactions follow S2PL is relatively straightforward. For every statement `x.f(...)` in the atomic section that invokes an ADT method, we insert code, just before the statement, to lock the ADT instance that `x` points to, unless it has already been

```
1  LV(x) {
2    if(x!=null && !LOCAL_SET.contains(x)) {
3     x.lock(+);
4     LOCAL_SET.add(x);
5  }}
```

Figure 5. Code macro with the *locking code*.

```
void f(Set x, Set y) { atomic {
 LOCAL_SET.init(); // prologue
 LV(x); int i = x.size();
 LV(y); y.add(i);
 foreach(t : LOCAL_SET) t.unlockAll(); // epilogue
}}
```

Figure 6. Atomic section that follows the S2PL protocol.

locked by the current transaction. At the end of the atomic section, we insert code to unlock all ADT instances that have been locked by the transaction. We achieve this as follows.

Locked ADTs Each transaction uses a thread-local set, denoted `LOCAL_SET`, to keep track of all ADT instances that it has currently locked. This set is used to avoid locking the same ADT multiple times and to make sure that all ADTs are eventually unlocked.

Prologue and Epilogue At the beginning of each atomic section, we add code that initializes `LOCAL_SET` to be empty. At the end of each atomic section, we add code that iterates over all ADTs in the `LOCAL_SET`, and invokes their `unlockAll` operations.

Locking Code We utilize the macro `LV(x)` shown in Fig. 5 to lock the ADT instance pointed to by a variable `x`. The macro locks the ADT pointed by `x` and adds it to `LOCAL_SET`. It has no impact when `x` is `null` or points to an ADT that has already been locked by the current transaction.

Fig. 6 shows an example for an atomic section with inserted locking code that ensures the S2PL protocol.

3.2 Lock Ordering Constraints

The basic idea sketched above does not ensure that all transactions lock ADTs in a consistent order. Hence, it is possible for the transactions to deadlock. We now describe an extension of the algorithm that statically identifies a suitable ordering on ADT instances and then inserts locking code to ensure that ADT instances are locked in this order.

We first describe the *restrictions-graph*, a data-structure that captures constraints on the order in which ADT instances can be locked. We utilize this graph to determine the order in which the locking operations are invoked.

A Static Finite Abstraction At runtime, an execution of the client program can create an unbounded number of ADT instances. Our algorithm is parameterized by a static finite abstraction of the set of ADT instances that the client program can create at runtime. Let PVar denotes the set of pointer variables that appear in the atomic code sections. The abstraction consists of an equivalence relation on PVar. For any $x \in$ PVar, let $[x]$ denote the equivalence class that x belongs to. The semantic guarantees provided by the abstraction are as follows. Any ADT instance created by any execution corresponds to exactly one of the equivalence classes. Furthermore, at any point during an execution, any variable $x \in$ PVar is guaranteed to be either null or point to an ADT represented by the equivalence class $[x]$.

Note that the abstraction required above can be computed using any pointer analysis (e.g., see [18]) or simply using the static types of pointer variables. In our compiler, we utilize the points-to

```
1  void g(Map m, int key1, int key2, Queue q) {
2  atomic {
3    Set s1 = m.get(key1);
4    Set s2 = m.get(key2);
5    if(s1!=null && s2!=null) {
6      s1.add(1);
7      s2.add(2);
8      q.enqueue(s1);
9    }
10 }}
```

Figure 7. Atomic section that manipulates a Map, a Queue, and two Sets.

analysis of [5] to compute this information. Note that even though various pointer analyses give more precise information than that captured by the above abstraction, our implementation requires only this information.

EXAMPLE 3.1. *The atomic section in Fig. 7 has 4 pointer variables (`m`, `q`, `s1` and `s2`). The equivalence relation consisting of the three equivalence classes $\{m\}$, $\{q\}$ and $\{s1, s2\}$ is a correct abstraction for this atomic section. This abstraction can be produced using static type information, without the need for a whole program analysis.*

The Restrictions-Graph Each node of the *restrictions-graph* represents an equivalence class in PVar (and, hence, is a static representation of a set of ADT instances that may be created at runtime). An edge $u \rightarrow v$ in the restrictions-graph conservatively indicates the possibility of an execution path along which an ADT instance belonging to u *must be locked before* an ADT instance belonging to v (within the same transaction). We identify these constraint edges as follows.

We write `l: x.f(...)` to denote a call to a method `f` via the variable `x` $\in$ PVar at the program location l. Consider an atomic section with two calls `l: x.f(...)` and `l': x'.f'(...)` such that location `l'` is reachable from location `l` (in the CFG of the atomic section). Obviously, we need to lock the ADT pointed by `x` before location `l` and to lock the ADT pointed by `x'` before location `l'`. Clearly, we can lock the ADT pointed to by `x` before we lock the ADT pointed to by `x'`. However, when can we lock these two ADTs the other way around? If the value of `x'` is never changed along the path between `l` and `l'`, then the ADT pointed by `x'` can be locked before `l`. However, if `x'` is assigned a value along the path between `l` and `l'`, then, in general, we may not be able to lock the ADT pointed to by `x'` (at location `l'`) before the ADT pointed to by `x`, as we may not know the identity of the ADT to be locked. In such a case, we conservatively add an edge [x] $\rightarrow$ [x'] to the restrictions-graph.

EXAMPLE 3.2. *In Fig. 7, the ADT pointed by `m` should be locked before the ADT pointed by `s1`, because the call `s1.add(1)` can only be executed after the call `m.get(key1)`, and the value of `s1` is changed between these calls.*

EXAMPLE 3.3. *Fig. 8 shows a restrictions-graph for the atomic section in Fig. 7. According to this graph the ADTs pointed by `m` should be locked before ADTs pointed by `s1` or `s2`. This is the only restriction in the graph. For example, the graph does not restrict the order between the ADTs pointed by `m` and the ADTs pointed by `q`. Moreover, it does not restrict the order between ADTs pointed by `s1` and the ADTs pointed by `s2` (even though they are represented by the same node).*

The calls in `l` and `l'` can be the same call (i.e., `l` = `l'`). This is demonstrated in the atomic section of Fig. 9: the call `set.size()` is reachable from itself (because of the loop), and

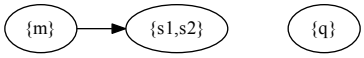

Figure 8. Restrictions-graph for the atomic section in Fig. 7.

```
1  atomic {
2  sum=0;
3  for(int i=0;i<n;i++) {
4    set = map.get(i);
5    if(set!=null) sum += set.size();
6  }}
```

Figure 9. Atomic section for which the restrictions-graph has a cycle.

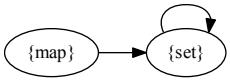

Figure 10. Restrictions-graph for the atomic section in Fig. 9.

Figure 11. Restrictions-graph for two atomic sections: the section in Fig. 1 and the section in Fig. 7.

`set` can be changed between two invocations of this call. A possible restrictions-graph is shown in Fig. 10.

The restrictions-graph is computed for all the atomic sections in the program. Fig. 11 shows a restrictions-graph for the atomic sections from Fig. 1 and Fig. 7.

3.3 Enforcing OS2PL on Acyclic Graphs

We now describe an algorithm to insert locking code into a the atomic sections to ensure that all transactions follow the OS2PL protocol. This technique is applicable as long as the restrictions-graph is acyclic. In Section 3.4, we show a technique to handle restrictions-graphs with cycles.

We sort the nodes in the restrictions-graph using a topological sort. This determines a total-order $\leq_{ts}$ on the equivalence classes. We define the relations $<$ and $\leq$ on the variables in PVar as follows. We say that $x < y$ iff $[x] <_{ts} [y]$ and that $x \leq y$ iff $[x] \leq_{ts} [y]$. Note that $\leq$ is only a preorder on PVar and not a total order. Variables belonging to different equivalence classes are always ordered by $<$, whereas variables belonging to the same equivalence class are never ordered by $<$.

The relation $<$ is used to statically determine the order in which ADT instances belonging to different equivalent classes are to be locked. However, we cannot do the same for ADT instances belonging to the same equivalence class. Instead, we dynamically determine the order in which ADT instances belonging to the same equivalence class are locked, as described below.

Locking Code Insertion Consider any statement `l: x.f(...)` in an atomic section that invokes an ADT method. We define the set $LS(\texttt{l})$ to be the set of variables y such that

1. $y \leq x$, and
2. There exists a (feasible) path, within the same atomic section, from `l` to some statement of the form `l': y.g(...)`, i.e., a statement that invokes an ADT method using `y`.

```
1  LV2(x,y) {
2   if(unique(x)<unique(y)) {
3     LV(x); LV(y) ;
4   } else {
5     LV(y); LV(x) ;
6  }}
```

Figure 12. Locking two equivalent variables in a unique order .

```
1  void g(Map m, int key1, int key2, Queue q) {
2  atomic {
3    LOCAL_SET.init(); // prologue
4    LV(m); Set s1 = m.get(key1);
5    LV(m); Set s2 = m.get(key2);
6    if(s1!=null && s2!=null) {
7      LV2(s1,s2); s1.add(1);
8      LV(s2); s2.add(2);
9      LV(q); q.enqueue(s1);
10   }
11   foreach(t : LOCAL_SET) t.unlockAll(); // epilogue
12 }}
```

Figure 13. The atomic section from Fig. 7 with the non-optimized locking code . The locking was created by using the order: m < s1,s2 < q.

```
1  atomic {
2  LOCAL_SET.init(); // prologue
3  LV(map); set=map.get(id);
4  if(set==null) {
5    set=new Set(); LV(map); map.put(id, set);
6  }
7  LV(map);LV(set);set.add(x); LV(map);LV(set);set.add(y);
8  if(flag) {
9    LV(map); LV(queue); queue.enqueue(set);
10   LV(map); map.remove(id);
11 }
12 foreach(t : LOCAL_SET) t.unlockAll(); // epilogue
13 }
```

Figure 14. . The atomic section from Fig. 1 with the non-optimized locking code . The locking was created by using the order: map < set < queue.

The set $LS(1)$ identifies the variables that we wish to lock before statement 1. We insert locking code for all variables in this set as follows:

- If $y < y'$, then the locking code for y is inserted before the locking code for y'.
- If y and y' are in the same equivalence class, the locking order is determined dynamically (since, we do not calculate a static order for such variables). This is done by using unique ADT identifiers (e.g., their memory addresses). Let `unique(y)` denote the unique identifier of the ADT pointed by y. These identifiers are used by the inserted code to (dynamically) determine the order in which the variables are handled. Fig. 12 demonstrates the case of two variables; in the general case ADTs are sorted to obtain the proper order. (For simplicity, we have omitted the handling of `null` pointers, which are straight-forward to handle.)

Fig. 13 shows the atomic section of Fig. 7 with the inserted code, and Fig. 14 shows the atomic section of Fig. 1 with the inserted code. For both atomic sections, we used the graph in Fig. 11.

```
1  class GlobalWrapper1 {
2    int size(Set s) {return s.size();}
3    ...
4  }
5  GlobalWrapper1 p1 = new GlobalWrapper1();
6  atomic { LOCAL_SET.init(); // prologue
7   sum=0;
8   for(int i=0;i<n;i++) {
9     LV(map); set = map.get(i);
10    if(set!=null) { LV(map); LV(p1); sum += p1.size(set); }
11  } foreach(t : LOCAL_SET) t.unlockAll(); // epilogue
12 }
```

Figure 15. The atomic section from Fig. 9 with the non-optimized locking code. Here, the objects pointed by `set` are represented by an instance of class `GlobalWrapper1` (pointed by a global pointer `p1`). The class `GlobalWrapper1` is a simple class that wraps Set objects.

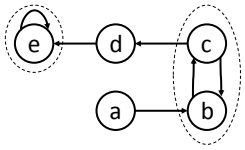

Figure 16. A graph with two cyclic components. One cyclic component is composed of nodes b and c, and another is composed of node e.

3.4 Handling Cycles

In Section 3.3, each Java object is treated as an individual ADT, and the acyclic restrictions graph is used to ensure that all ADTs are locked in a consistent order. In order to enforce the OS2PL protocol when the computed restrictions graph has cycles, we treat several Java objects as a single global ADT. For example, the OS2PL protocol can be enforced for the code in Fig. 9 (its graph appears in Fig. 10), by treating all objects pointed by the pointer `set` as a single global ADT. Fig. 15 shows the code section from Fig. 9 with locking code that follows OS2PL: in this code, the pointer `p1` points to a global ADT that wraps the relevant instances of Set.

Given a restrictions graph with cycles, we first find all its strongly connected components that contain cycles — such component is called a *cyclic component*. Fig. 16 shows an example for a graph and its cyclic components.

For each cyclic component C, we treat all the objects pointed by the variables in C as a single ADT that is pointed by the pointer p_C. This is realized by creating (for each cyclic component C) a global pointer p_C that points to an ADT that wraps the Java objects pointed by C's pointers (as demonstrated in Fig. 15). We now repeat the algorithm presented in Section 3.3 — in this case, the restrictions graph will be acyclic because each p_C replaces all pointers in C, and p_C always points to the same ADT (hence, no pointer has to be locked before p_C).

3.5 Optimizations

The algorithm described above is simplistic in some respects. In Appendix A, we present a sequence of code transformations whose goal is to reduce the overhead of the synthesized code and to increase the parallelism permitted by the concurrency control. In particular, we present transformations that remove inserted code that can be shown to be redundant, and transformations that move calls to `unlockAll` so as to release locks on ADTs as early as possible (as determined by a static analysis). Fig. 17 shows the optimized version of Fig. 14.

```
1  atomic { map.lock(+);
2  set=map.get(id);
3  if(set==null) {
4    set=new Set(); map.put(id, set);
5  }
6  set.lock(+); set.add(x); set.add(y);
7  if(flag) { queue.lock(+);
8    queue.enqueue(set); queue.unlockAll();
9    map.remove(id);
10 }
11 map.unlockAll(); set.unlockAll();
12 }
```

Figure 17. The optimized version of Fig. 14. Note that a large portion of the locking code is removed, and the set LOCAL_SET is not explicitly used. Also, the ADT pointed by queue is unlocked before the end of the section.

```
1  atomic { //{get(id),put(id,*),remove(id)}
2    set=map.get(id); //{put(id,*),remove(id)}
3    if(set==null) { //{put(id,*),remove(id)}
4      set=new Set(); map.put(id, set); //{remove(id)}
5    } //{remove(id)}
6    set.add(x); //{remove(id)}
7    set.add(y); //{remove(id)}
8    if(flag) { //{remove(id)}
9      queue.enqueue(set); //{remove(id)}
10     map.remove(id); //{}
11   } //{}
12 }
```

Figure 18. The code section from Fig. 1 annotated with the inferred symbolic sets for the variable map.

4. Inferring Refined Symbolic Sets

Our compiler refines the generic symbolic sets used in Section 3 by considering the actual operations used in the atomic sections. The compiler analyzes the atomic sections, to infer for every pointer variable x and code location l, a symbolic set $SY_{x,l}$ that conservatively describes the set of future ADT operations that may be invoked on the ADT instance that x points to. We use a simple backward analysis [20] to compute this information. As in Section 3, we do not distinguish between pointer variables belonging to the same equivalence class (i.e., the information computed is the same for all variables in the same equivalence class).

This analysis is equivalent to the static algorithm used in [9]. But, in contrast to [9] in which all shared memory is treated as a single ADT, we separately repeat these steps for each equivalence class.

Fig. 18 illustrates the inferred symbolic sets for the code section from Fig. 1 for the equivalence class consisting of variable map.

Recall that the algorithm presented in Section 3 identifies a set of pairs (x, l) such that we insert a locking operation on variable x at program location l. We use the same algorithm now, except that we use the symbolic set $SY_{x,l}$ instead of using the generic symbolic set from Section 3.

Fig. 2 shows the code from Fig. 17 after refining its symbolic set (each generic symbolic set "+" is replaced with a symbolic set that represents operations which are actually used in this section).

5. Realizing Semantic Locking

5.1 Locking Modes

The compiler implements the semantic locking of an ADT A by generating a finite number of *locking modes* for A where each mode represents a set of A's runtime operations (this can be seen as a generalization of the *read-mode* and the *write-mode* which are used for standard read-write-locks). The generated locking modes are based on the symbolic sets that have been inferred in Section 4: for each ADT A the compiler creates locking modes according to the symbolic sets that may be used to lock A's operations. In the following paragraphs we describe our approach to compute and use locking modes for ADT A.

We distinguish between two types of symbolic sets: *constant symbolic sets* and *variable symbolic sets*. A *constant symbolic set* is a symbolic set with no variables (each one of its arguments is either a specific value or a $*$ symbol) — e.g., $\{add(5)\}$ and $\{add(*)\}$ are constant symbolic sets. A *variable symbolic set* is a symbolic set that has variables (at least one) — e.g., $\{add(i), remove(j)\}$ is a variable symbolic set. Notice that, a constant symbolic set represents a constant set of runtime operations, whereas the set of runtime operations represented by a variable symbolic set depends on the actual runtime values of its variables.

For each constant symbolic set SY_c, we create a locking mode l that represents the same runtime operations; and replace each call lock(SY_c) with lock(l).

In order to handle variable symbolic sets, we use a hash function ϕ : *Value* $\rightarrow \{\alpha_1, \ldots, \alpha_n\}$ that maps runtime values to n *abstract values* (ϕ is a parameter of our algorithm). An *abstract value* α_i is used to represent a disjoint set of runtime values which may be used by the program (in particular, α_i represents the values in $\{v \in Value \mid \phi(v) = \alpha_i\}$). For any variable symbolic set SY_v with variables $a_1, \ldots, a_k$, we create a locking mode for each assignment of the abstract values to SY_v's variables — this will guarantee that each runtime instance of SY_v is represented by one of the locking modes. For example, if $n = 2$ then for the variable symbolic set $\{add(i), remove(j)\}$ we create 4 locking modes which are represented by: $\{add(\alpha_1), remove(\alpha_1)\}$, $\{add(\alpha_1), remove(\alpha_2)\}$, $\{add(\alpha_2), remove(\alpha_1)\}$ and $\{add(\alpha_2), remove(\alpha_2)\}$.

We then replace each call lock(SY_v) where SY_v is a variable symbolic set with code that dynamically finds the relevant locking mode l (by using ϕ) and passes l to the lock method. For example x.lock($\{add(i), remove(j)\}$) is replaced by the code:

```
1  t1 = φ(i)
2  t2 = φ(j)
3  l = the locking mode that represents {add(t1), remove(t2)}
4  x.lock(l)
```

In this code, lines 1, 2 find abstract values that represent the runtime values of i and j; lines 3, 4 pass the relevant locking mode to lock.

5.2 Realizing a Locking Mechanism

Commutativity Specification In order to utilize semantic properties of the shared ADTs, the user of our compiler has to provide a *commutativity specification* for each Java class that represents a shared ADT. For any two operations o, o' (which are implemented by the class) the specification contains a condition $I_{o,o'}$ such that if $I_{o,o'}$ is satisfied then o and o' commute. An example for a commutativity specification appears in Fig. 3: in this example, operations add(v) and contains(v') are commutative when their values are different (because of the condition $v \neq v'$); operations add(v) and add(v') are always commutative (their condition is true); and operations add(v) and size() are never commutative (their condition is false).

Commutativity Function. For any ADT A, we use the inferred locking modes and the given commutativity specifications to compute a *commutativity function* F_c that maps every two inferred locking modes l, l' to either *true* or *false*. This function satisfies $F_c(l, l') = true$, if all operations represented by l are commutative with all operations represented by l'.

```
1  object internalLock = new Object();
2  void lock(mode l) { start:
3    for each mode l'
4      if (C_l'>0 and not F_c(l,l')) goto start;
5    synchronized(internalLock) { // standard Java lock
6      for each mode l'
7        if (C_l'>0 and not F_c(l,l')) goto start;
8      C_l.Increment() ;
9    }
10 }
11 void unlock(mode l) { C_l.Decrement(); }
```

Figure 20. Pseudo code of locking mechanism for an ADT A. F_c is the commutativity function, and each locking mode l is represented by an atomic counter C_l . The atomic counter C_l holds the number of transactions that hold a lock on the ADT in mode l.

Fig. 19 shows an example for a commutativity function which is computed using the commutativity specification in Fig. 3(b). In this example we use a hash function ϕ : *Value* $\rightarrow \{\alpha_1, \alpha_2\}$ that satisfies $\phi(5) = \alpha_1$.

Locking Mechanism. We implement the locking mechanism of each ADT A (its `lock` and `unlock` methods) by utilizing the inferred locking modes and their computed commutativity function. Fig. 20 shows the pseudo code of the implementation used by our compiler. In the shown code, F_c is the commutativity function, and each locking mode l is represented by an atomic counter C_l . The counter C_l contains the number of transactions that hold a lock on the ADT in mode l (it is initialized to zero). At lines 3-4 the code tries to avoid using the internal lock at line 5 (when the lock cannot be acquired by the transaction). The main logic of `lock` is implemented at lines 5-9.

Lock Partitioning. The single internal lock in the above mechanism can become a synchronization bottleneck. We try to avoid such bottlenecks by trying to partition the above mechanism.

If M_A is the set of A's modes, we try to find k disjoint sets of modes $M_1, \ldots, M_k$ such that for every $i \neq i'$, all operations represented by the modes in M_i commute with all operations represented by the modes in $M_{i'}$. (i.e., $\forall l \in M_i, l' \in M_{i'} : i \neq i' \rightarrow F_c(l, l') = \mathit{true}$). Hence, if $l \in M_i$ and $l' \in M_{i'}$ then it is correct to simultaneously hold locks on mode l and mode l'. If we find $k \geq 2$ sets, we create a separate locking mechanism for each one of the sets.

For example, given a Set with 3 locking modes l_1, l_2, l_3 that, respectively, represent $\{get(7)\}$, $\{get(*)\}$, and $\{add(1)\}$. We create a locking mechanism Δ that handles mode l_1, and another mechanism Δ' that handles modes l_2 and l_3. (Becuase the operations represented by l_1 commute by the operations represented by l_2 and l_3.) When l_1 is locked/unlocked we use Δ, and when l_2 or l_3 is locked/unlocked we use Δ'.

5.3 Optimizations

We use the following optimizations:

1. Two locking modes l_1, l_2 are *indistinguishable* if $\forall l : F_c(l_1, l) = F_c(l_2, l)$. Each set of indistinguishable locking modes is implemented as a single locking mode.
2. The same ADT type may be used differently in various parts of the program. Hence, in order to create more specialized locking, we repeat the above approach (Sections 5.1 and 5.2) for each equivalence class of the pointers. In other words, we implement locking for every ADT and a node of the restrictions-graph.
3. In order to bound the number of inferred nodes, the compiler uses a numeric parameter N which represents the maximum number of locking modes (of each ADT instance). If we infer more than N modes, then we merge them until we have N modes.

The above optimizations enable using a large number of abstract values (e.g., in Section 6 we use a hash function that maps runtime values to 64 abstract values).

6. Performance Evaluation

In this section we present an experimental evaluation of our approach. The presented evaluation shows that our approach produces efficient and scalable synchronization. Furthermore, it shows that utilizing the semantic properties of ADTs is beneficial, and that the produced synchronization is comparable to hand-crafted synchronization.

Our evaluation is based on 5 different benchmarks. In 3 benchmarks we evaluate the performance of composite modules (*ComputeIfAbsent*, *Graph*, and *Cache*). In 2 benchmarks we evaluate the performance of Java applications (*Intruder* and *GossipRouter*). In most benchmarks (*Graph*, *Cache*, *Intruder* and *GossipRouter*) the shared state is composed from several (at least two) ADT instances.

We compare the performance of our approach to: (i) a single global lock (denoted by **Global**); (ii) an implementation of the standard two-phase locking protocol where each ADT instance is protected by a standard lock (denoted by **2PL**); and (iii) a hand-crafted synchronization (denoted by **Manual**).

The **2PL** synchronization was implemented by using the output of Section 3 — instead of locking operations of ADT instance A, we acquire a Java lock that protects A. For *Graph*, *Cache* and *GossipRouter*, **Manual** was implemented by manually optimizing the synchronization produced by [9]; for *ComputeIfAbsent* and *Intruder*, **Manual** was implemented by using variants of the lock striping technique (see below).

We use a machine with four Intel(R) Xeon(R) E5-4650 CPUs, eight cores each (i.e., 32 physical cores overall). The machine's hyperthreading is disabled.

6.1 Composite Modules

We evaluate the performance of 3 composite modules which are implemented using several general-purpose ADTs. Each (public) procedure of the modules is marked as an atomic section.

We use the evaluation methodology of [13] and consider several different workloads (as in other works, e.g., [12]). Every pass of the test program consists of each thread performing ten million randomly chosen procedure invocations on a shared module. To ensure consistent and accurate results, each experiment consists of five passes; the first pass warms up the VM and the four other passes are timed. Each experiment was run four times and the arithmetic average of the throughput is reported as the final result.

ComputeIfAbsent The *ComputeIfAbsent* benchmark [9] is a simulation of a pattern that is widely used in Java applications. Many bugs in Java programs are caused by non-atomic realizations of this simple pattern [22]. It can be described with the following pseudo-code:

```
if(!map.containsKey(key)) {
  value = ... // pure computation
  map.put(key, value);
}
```

The idea is to compute a value and store it in a Map, if and only if, the given key is not already present in the Map. In our experiments the computation was emulated by allocating 128 bytes.

Here, **Manual** was implemented by using a *lock striping technique* with 64 locks (similar to [12]) where each key is protected by one of the locks. This benchmark was also compared to a hand-

	$\{add(*)\}$	$\{add(5)\}$	$\{add(\alpha_1)$, $remove(\alpha_1)\}$	$\{add(\alpha_1)$, $remove(\alpha_2)\}$	$\{add(\alpha_2)$, $remove(\alpha_1)\}$	$\{add(\alpha_2)$, $remove(\alpha_2)\}$
$\{add(*)\}$	*true*	*true*	*false*	*false*	*false*	*false*
$\{add(5)\}$		*true*	*false*	*true*	*false*	*true*
$\{add(\alpha_1), remove(\alpha_1)\}$			*false*	*false*	*false*	*true*
$\{add(\alpha_1), remove(\alpha_2)\}$				*true*	*false*	*false*
$\{add(\alpha_2), remove(\alpha_1)\}$					*true*	*false*
$\{add(\alpha_2), remove(\alpha_2)\}$						*false*

Figure 19. A commutativity function for the Set ADT described in Fig. 3. For every pair of locking modes l, l' (described as constant symbolic sets) the function returns *true*, IFF, l and l' commute. The shown function has been computed for the symbolic sets $\{add(*)\}$, $\{add(5)\}$ and $\{add(i), remove(j)\}$. We assume that ϕ maps runtime values to the abstract values α_1 and α_2; and $\phi(5) = \alpha_1$.

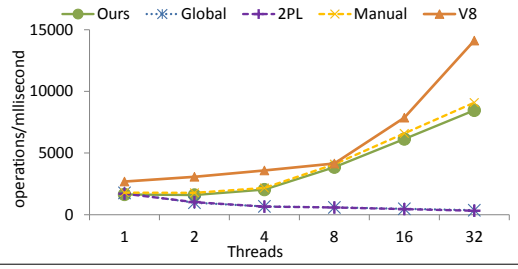

Figure 21. *ComputeIfAbsent* throughput as a function of the number of threads.

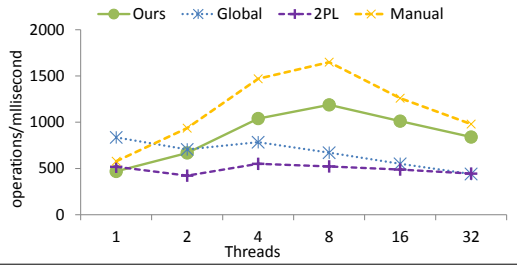

Figure 23. *Cache* throughput as a function of the number of threads. 90% *Get*, 10% *Put* and *size=5000K*.

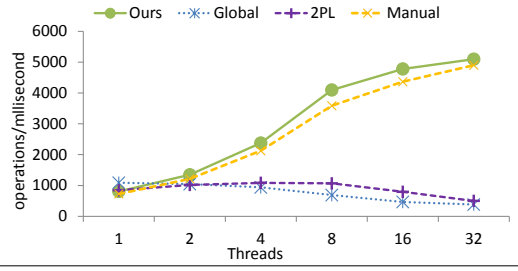

Figure 22. *Graph* throughput as a function of the number of threads. 35% *find successors*, 35% *find predecessors*, 20% *insert edge* and 10% *remove edge*.

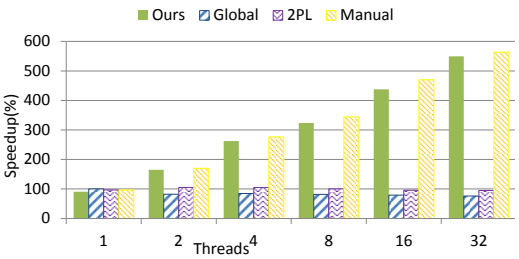

Figure 24. *Intruder*. Speedup over a single-threaded execution.

crafted implementation of the *ComputeIfAbsent* pattern from [1] (denoted by **V8**). The results are shown in Fig. 21.

Graph This benchmark uses a Java implementation of the Graph from [12]. The Graph consists of four procedures: *find successors*, *find predecessors*, *insert edge*, and *remove edge*. The graph is implemented by using two *Multimap* [4] instances. Results for one workload, from [12], are shown in Fig. 22 (these results are similar to the other workloads in [12]).

Cache This benchmark uses the Tomcat's cache [2]. This cache is implemented by using a Map and a *WeakMap* [4]. The cache consists of two procedures `Put` and `Get` which manipulate the maps. In this cache, the `Get` is not a read-only procedure (in some cases, it copies an element from one map to another). The cache gets a parameter (*size*) which is used by its algorithm. Results for one workload, from [9], are shown in Fig. 23 (these results are similar to the other workload in [9]).

6.2 Java Applications

Intruder The Intruder benchmark [7] is a multi-threaded application that emulates an algorithm for signature-based network intrusion detection. For our study we use its Java implementation from [16] in which atomic sections are already annotated. We use the workload which is represented by the configuration *"-a 10 -l 256 -n 16384 -s 1"* (see [7]).

For this benchmark, **Manual** is an ad-hoc synchronization in which we combine lock striping and efficient linearizable implementations of a Map and a Queue. The results are shown in Fig. 24 as the speedup over a single-threaded execution.

GossipRouter The *GossipRouter* is a Java multi-threaded routing server from [3]. Its main state is a *routing table* which consists of an unbounded number of Map ADTs. We use the version from [9] (in this version the atomic sections are already annotated). We used a performance tester from [3] (called *MPerf*) to simulate 16 clients where each client sends 5000 messages. In this experiment the number of threads cannot be controlled from the outside (because the threads are autonomously managed by the router). Therefore, instead of changing the number of threads, we changed the number of active cores. The results are shown in Fig. 25 as the speedup over a single-core execution.

An interesting aspect of this benchmark is that the atomic sections contain I/O operations (logging operations, and operations that send network messages). We treat the I/O operations as thread-local operations, which is safe for this benchmark as the I/O operations are never used to communicate between threads. This kind of extension is possible because our approach does not use rollbacks.

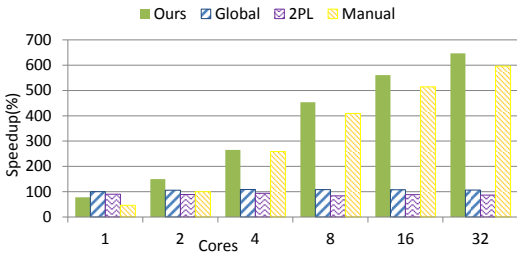

Figure 25. *GossipRouter*. Speedup over a single-core execution.

7. Related Work

Concurrent Libraries of Abstract Data Types The solution used in practice, to simplify concurrency control, is libraries that provide sophisticated concurrent ADTs such as sets and maps (e.g., see [14]). These ADTs ensure atomicity of their basic operations, while hiding the complexity of synchronization inside their libraries. Unfortunately, this approach does not address a common need: the need to execute a code fragment atomically, where the code fragment performs multiple ADT operations. The client programmers must devise their own concurrency control solution in this scenario. As shown in [22], this leads to many bugs in practice.

Atomic Sections Atomic sections (or software transactions) have been proposed as a language construct that simplifies a programmer's job and leaves it to a compiler and runtime to realize the necessary concurrency control. A number of implementation approaches have been proposed for realizing atomic sections [11]. The classical approaches to implementing atomic sections have the following distinguishing characteristics.

1. They rely on *optimistic concurrency control*, which detects conflicts after the fact, and requires the use of *rollbacks* to resolve conflicts.
2. They use a *data-based* approach to conflict detection: a dependence is inferred between two transactions if they both access the same location, and at least one of the accesses is a *write*.
3. They require all code in an atomic section (including library methods called from the atomic section) to use the same transactional runtime. In particular, this means that all code in an atomic section must use the same concurrency control solution.

Each of these aspects has associated disadvantages, some of which have been addressed by previous work as explained below.

Semantic Conflict Detection A data-based approach to detecting conflicts/dependence between transactions is imprecise and can lead to spurious conflicts/dependences. A semantics-based approach (e.g., one that identifies two high-level operations as commuting even though they access and modify the same data) to identifying dependences/conflicts between transactions can enable greater parallelism. This idea is quite old and was proposed early on for database transaction implementations (e.g., [6, 21]).

Similar ideas have also motivated the development of various implementation techniques and variants of software transactions (e.g., see [17]). Most of these approaches require the use of a rollback mechanism. In contrast, our approach exploits semantic dependence detection without using a rollback mechanism. This may be an advantage in some cases: for example, when a rollback mechanism has a high runtime and memory overhead [8], and when I/O operations are involved.

Automatic Locking Several algorithms have been proposed to realize atomic sections using pessimistic concurrency control, i.e., locking. These algorithms synthesize the necessary locking code using static analysis (e.g., [10, 19]). Our algorithm is inspired by these algorithms, with the key difference that the locks in our approach are associated with ADT operations while in previous work locks are associated with memory locations or data. Hence, in contrast to existing locking approaches, our algorithm can exploit semantic dependence detection and enable more parallelism.

The approach in [9] can be seen as a semi-automatic lock inference algorithm that is able to exploit semantic properties of the shared state. In contrast to this paper, in [9] the shared state has to be represented as a single ADT (a shared library), and this single ADT is required to support special synchronization operations (which are implemented manually). Moreover, in contrast to this paper which utilizes a semantic-aware variant of the two-phase locking protocol, the synchronization in [9] is implemented as a semantic-aware variant of the tree locking protocol.

References

[1] gee.cs.oswego.edu/dl/jsr166/dist/jsr166edocs/jsr166e/ConcurrentHashMapV8.html.

[2] www.devdaily.com/java/jwarehouse/apache-tomcat-6.0.16/java/org/apache/el/util/ConcurrentCache.java.shtml.

[3] http://www.jgroups.org.

[4] guava-libraries. code.google.com/p/guava-libraries/.

[5] Wala. http://wala.sourceforge.net.

[6] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] CAO MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford transactional applications for multi-processing. In *IISWC* (2008).

[8] CASCAVAL, C., BLUNDELL, C., MICHAEL, M., CAIN, H. W., WU, P., CHIRAS, S., AND CHATTERJEE, S. Software transactional memory: Why is it only a research toy? *Queue 6*, 5 (Sept. 2008), 46–58.

[9] GOLAN-GUETA, G., RAMALINGAM, G., SAGIV, M., AND YAHAV, E. Concurrent libraries with foresight. In *PLDI* (2013).

[10] GUDKA, K., HARRIS, T., AND EISENBACH, S. Lock inference in the presence of large libraries. In *ECOOP*. 2012.

[11] HARRIS, T., LARUS, J., AND RAJWAR, R. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture 5*, 1 (2010).

[12] HAWKINS, P., AIKEN, A., FISHER, K., RINARD, M., AND SAGIV, M. Concurrent data representation synthesis. In *PLDI* (2012).

[13] HERLIHY, M., LEV, Y., LUCHANGCO, V., AND SHAVIT, N. A provably correct scalable concurrent skip list. In *OPODIS* (2006).

[14] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kauffman, Feb. 2008.

[15] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *TOPLAS 12* (1990).

[16] KORLAND, G., SHAVIT, N., AND FELBER, P. Noninvasive concurrency with java stm. In *MULTIPROG* (2010).

[17] KOSKINEN, E., PARKINSON, M., AND HERLIHY, M. Coarse-grained transactions. In *POPL* (2010), pp. 19–30.

[18] LHOTÁK, O., AND HENDREN, L. Scaling java points-to analysis using spark. In *Proceedings of the 12th international conference on Compiler construction* (2003), CC'03.

[19] MCCLOSKEY, B., ZHOU, F., GAY, D., AND BREWER, E. Autolocker: synchronization inference for atomic sections. In *POPL* (2006).

[20] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[21] SCHWARZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 223–250.

[22] SHACHAM, O., BRONSON, N., AIKEN, A., SAGIV, M., VECHEV, M., AND YAHAV, E. Testing atomicity of composed concurrent operations. In *OOPSLA* (2011).

```
1  atomic {
2  LOCAL_SET.init(); // prologue
3  LV(map); set=map.get(id);
4  if(set==null) {
5    set=new Set(); map.put(id, set);
6  }
7  LV(set); set.add(x); set.add(y);
8  if(flag) {
9    LV(queue); queue.enqueue(set);
10   map.remove(id);
11 }
12 foreach(t : LOCAL_SET) t.unlockAll(); // epilogue
13 }
```

Figure 26. The code from Fig. 14 after removing redundant instances of `LV(x)`.

[23] SĂLCIANU, A., AND RINARD, M. Purity and side effect analysis for Java programs. In *VMCAI* (2005), pp. 199–215.

A. Optimizations

In this appendix we describe optimizations for the algorithm in Section 3. We present a set of simple semantics-preserving transformations, with the goal of reducing the overhead of the synthesized code and increasing the parallelism permitted by the concurrency control.

In the sequel, we use the term "path" to denote feasible execution paths *within a single atomic section*.

Removing Redundant LV(x) In some cases, the code `LV(x)` inserted at a location l is redundant. Our compiler removes redundant instances of `LV(x)` by repeatedly using the following rules:

- If the object pointed to by x at l is locked along all (feasible) paths from the beginning of the atomic section to l, then the code `LV(x)` has no effect at l and can be removed. For example, in Fig. 14, `LV(map)` can be removed from line 9 because the Map has already been locked.
- If the object pointed to by x at l is never used along any feasible path from l to the end of the atomic section, then the code `LV(x)` is not required at l and can be removed.

Fig. 26 shows the code from Fig. 14 after removing redundant instances of `LV(x)`.

Removing Redundant LOCAL_SET Usage Our algorithm uses `LOCAL_SET` to avoid locking the same object multiple times and to ensure that all objects are unlocked before the end of the atomic section. Often, we can achieve these goals without using `LOCAL_SET`. `LOCAL_SET` is not needed for a pointer variable x if the following conditions hold:

(1) No path contains an occurrence of `LV(x)` and another occurrence of `LV(y)` where `x` and `y` may point to the same object.
(2) The value of `x` is never modified along any path from an occurrence of `LV(x)` to the end of the atomic section.
(3) The value of `x` is null at the end of any path to the end of the atomic section that contains no occurrence of `LV(x)`.

Because of (1) we know that re-locking is not possible; and because of (2) and (3) we know that we can release all the objects used via x by calling to "`if(x!=null) x.unlockAll()`" at the end of the atomic section.

If the conditions are satisfied for x, we replace all instances of `LV(x)` with "`if(x!=null) x.lock(+)`" ; and at the end of the section we insert the code "`if(x!=null) x.unlockAll()`".

If, after all applications of the above transformation, the set `LOCAL_SET` is not used for any variable in an atomic section, we remove the set, and the corresponding prologue and epilogue. Fig. 27 shows the code from Fig. 26 after applying this optimization.

```
1  atomic { if(map!=null) map.lock(+);
2  set=map.get(id);
3  if(set==null) {
4    set=new Set(); map.put(id, set);
5  }
6  if(set!=null) set.lock(+); set.add(x); set.add(y);
7  if(flag) { if(queue!=null) queue.lock(+);
8    queue.enqueue(set);
9    map.remove(id);
10 }
11 if(map!=null) map.unlockAll();
12 if(set!=null) set.unlockAll();
13 if(queue!=null) queue.unlockAll();
14 }
```

Figure 27. The code from Fig. 26 after removing the code that uses `LOCAL_SET`.

```
1  atomic { if(map!=null) map.lock(+);
2  set=map.get(id);
3  if(set==null) {
4    set=new Set(); map.put(id, set);
5  }
6  if(set!=null) set.lock(+); set.add(x); set.add(y);
7  if(flag) { if(queue!=null) queue.lock(+);
8    queue.enqueue(set);
9    if(queue!=null) queue.unlockAll(); //this line was moved
10   map.remove(id);
11 }
12 if(map!=null) map.unlockAll();
13 if(set!=null) set.unlockAll();
14 }
```

Figure 28. The code from Fig. 27 after moving an `unlockAll` operation.

Early Lock Release Our basic algorithm unlocks all objects at the end of the atomic section. In some cases, it is possible to unlock some objects at an earlier point (before the end of the atomic section) without violating the locking protocol. We now describe the conditions under which we perform such an early lock release.

It is safe to move "`if(x!=null) x.unlockAll()`" occurring at the end of the atomic section to a program point l if the following conditions are satisfied:

(1) The object pointed to by x is not used between l and the end of the atomic section.
(2) No object is locked between l and the end of the atomic section.
(3) Every path to the end of the atomic section passes through l or ends with a *null* value for `x`.

The compiler tries to find the earliest program point l (as measured by the length of the shortest path from the beginning of the atomic section to l) that satisfies these conditions. If such a point l is found, we move the code "`if(x!=null) x.unlockAll()`" to location l. Because of (1) and (2), we know that the protocol rules are not violated. Because of (3), we know that the relevant object will eventually be unlocked. Fig. 28 shows the code from Fig. 27 after applying this optimization. Note that, the unlocking code of "`queue`" has been moved to line 9 of Fig. 28.

Removing Redundant If-Statements In some cases, the inserted if-condition "`if(x!=null)`" is not needed. For any location l and variable x for which we can prove that x is never `null` at l, we remove the condition "`if(x!=null)`" from l. Fig. 17 shows the code from Fig. 28 after applying this optimization.