Automatic Semantic Locking

Guy Golan-Gueta
Tel Aviv University
ggolan@tau.ac.il

G. Ramalingam
Microsoft Research
grama@microsoft.com

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

Eran Yahav
Technion
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 develop a novel 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.

In our approach, each ADT implements ADT-specific semantic locking operations that serve to exploit the semantics of ADT operations. We develop a synthesis algorithm that automatically inserts calls to these locking operations in a set of given code fragments (in a client program) to ensure that these code fragments execute atomically without deadlocks, and without rollbacks.

We have implemented the synthesis algorithm and several general-purpose ADTs with semantic locking. We have applied the synthesis algorithm to several Java programs that use these ADTs. Our results show that our approach enables efficient and scalable synchronization.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Compiler, Synchronization, Composition

1. The Problem

Atomic sections are a language construct that allow 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. In this work we develop a methodology and automation support for realizing a restricted form of atomic sections.

The example in Fig. 1, inspired by the code of the *Intruder* benchmark (from [1]), illustrates the problem we address in this paper. The shared 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, and thus each individual ADT operation appears to execute atomically. However, in this case, we wish the entire code fragment to execute atomically: the individual ADTs cannot provide this guarantee.

```
atomic { set=map.get(id);
         if(set==null) { set=new Set(); map.put(id, set); }
         set.add(x);
         if(flag) { queue.enqueue(set); map.remove(id); }      }
```

Figure 1. Code that manipulates several linearizable ADTs.

We consider a Java multi-threaded program (also referred to as a *client*), which makes use of several linearizable ADT libraries. We assume that the only mutable state shared by multiple threads are *instances* of ADTs. We permit atomic sections as a language construct: a block of code may be marked as an atomic section. 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.

2. Overview

Our approach decomposes the responsibility for the task into two parts: one to be realized by the ADT implementations and one to be realized by a compiler (on behalf of the client code). We require each ADT implementation to provide a set of *semantic locking* operations. We show how a compiler can, given these ADT locking operations, automatically compile atomic sections in a given (client) program so as to provide the desired guarantees.

ADTs with Semantic Locking A *semantic locking operation* of an ADT is used by a client of the ADT to acquire permission to invoke a specific set of *base operations* on the ADT: in this case, we say that the client has a lock on the corresponding set of underlying ADT operations. It is the client transaction's responsibility to ensure that it has a lock on a

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2656-8/14/02.
http://dx.doi.org/10.1145/2555243.2555281

```
// Standard API                  // Synchronization API
void add(int i);                 void lockAll();
void remove(int i);              void lockAdd();
boolean contains(int i);         void lockValue(int i);
int size();                      void unlockAll();
```

Figure 2. API of a Set with semantic locking.

base ADT operation before it invokes that operation. The ADT has the responsibility to ensure that two transactions are allowed to simultaneously hold locks on operations op_1 and op_2, respectively, only if op_1 and op_2 commute. Fig. 2 shows an example for an API of a Set ADT with semantic locking. In this example, an invocation of "`lockAdd()`" acquires locks on the `add` operations of the Set — hence, after transaction t invokes "`lockAdd()`", t is allowed to invoke the method "`void add(int i)`". For any integer v, an invocation of "`lockValue(`v`)`" acquires locks on all Set operations that refer to value v (e.g., "`lockValue(7)`" acquires locks on the operations: `add(7)`, `remove(7)` and `contains(7)`). The method "`lockAll()`" acquires locks on all operations of the Set; and "`unlockAll()`" releases all locks owned by the current transaction.

We have used a simple annotation language (adopted from [4]) to specify the semantics of a locking operation (namely, the set of base operations it corresponds to); these annotations enable our compiler to take an ADT description as a parameter (our compiler is not aware of any specific ADT).

Automatic Atomicity We have developed a compiler for atomic sections. Given a client program and a specification of the semantic locking operations of the ADTs used by the client, the compiler inserts invocations of semantic locking operations into the atomic sections in the client program to guarantee atomicity and deadlock-freedom of these atomic sections. The synchronization generated by our compiler follows a semantics-aware two-phase locking protocol [3]. Fig. 3 shows the result of applying our compiler to the atomic section of Fig. 1. The basic idea is to consider every base ADT operation invocation (say "`x.op()`") in the transaction and insert a conditional call ("`if (cond) x.lockY()`") to a semantic locking operation before the base operation. The condition is used to dynamically check if the object has already been locked. A static analysis identifies the set S of all base operations that may be performed on the relevant object (`x` in the above example) in the future (by the transaction), and a semantic locking operation (`lockY()` in the above example) is chosen so that it obtains a lock on a superset of S. (An optimizing phase is used to identify when either the conditional check or the lock-acquisition is redundant and eliminate them.)

This basic scheme is modified by determining an order in which objects are locked (to avoid deadlocks) and by moving the semantic locking operations earlier in the transaction to respect the determined ordering. All locks are released at the end of the transaction to ensure two-phase-locking.

```
atomic {
  map.lockKey(id); set=map.get(id);
  if(set==null) { set=new Set(); map.put(id, set); }
  set.lockAdd(); set.add(x);
  if(flag) { queue.lockAll();
    queue.enqueue(set); queue.unlockAll(); map.remove(id); }
  map.unlockAll(); set.unlockAll();  }
```

Figure 3. The atomic section of Fig. 1 with semantic locking operations automatically inserted by our compiler.

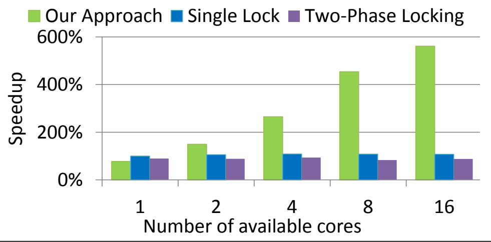

Figure 4. GossipRouter. Speedup over a single core.

A separate optimization phase it used to release locks on objects earlier when it is possible to do so safely.

Pointers and Limitations Our compiler handles programs in which pointers to ADTs are dynamically manipulated. For some of these programs, our compiler is unable to ensure deadlock-freedom by using only the semantic locking operations of the ADTs. These programs are handled using an additional specialized coarse-grain synchronization. However, our experimental evaluation shows that our compiler creates effective synchronization that benefits from semantic locking even in a program (the *GossipRouter* [2]) in which coarse-grained synchronization is used.

3. Performance Evaluation

We have applied our approach to 5 benchmarks. In 3 benchmarks we evaluate the performance of modules that are implemented using several general-purpose ADTs: a *Graph* (from [4]) a *ReferencesCounter* (reference counting module implemented using a Map and Counters) , and a specialized *Cache* (from [4]). In 2 benchmarks we evaluate the performance of Java applications: *Intruder* (from [1]) and *GossipRouter* (from [2]). The results show, for all benchmarks, that the our approach provides efficient and scalable performance. Fig. 4 shows the results for the *GossipRouter* application (in the figure, our approach is compared to a single lock and to a realization of the two-phase locking protocol).

References

[1] sites.google.com/site/deucestm/.

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

[3] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] Golan-Gueta, G., Ramalingam, G., Sagiv, M., and Yahav, E. Concurrent libraries with foresight. In *PLDI* (2013).