Finding Linearization Violations in Lock-Free Concurrent Data Structures

by

Sebastien Alberto Dabdoub

S.B., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2013
[June 2013]

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2013

Certified by . . . .
Frans Kaasheoek
Professor
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by . . . .
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Finding Linearization Violations in Lock-Free Concurrent Data Structures

by

Sebastien Alberto Dabdoub

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Finding bugs in lock-free concurrent programs is hard. This is due in part to the difficulty of reasoning about the correctness of concurrent algorithms and the timing-sensitive nature of concurrent programs. One of the most widely used tools for reasoning about the correctness of concurrent algorithms is the linearization property. This thesis presents a tool for automatic dynamic checking of concurrent programs under the Total-Store-Order (TSO) memory model and a methodology for finding linearization violations automatically with the tool.

Thesis Supervisor: Frans Kaasheoek
Title: Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

I wish to thank my advisors, Frans Kaashoek and Nickolai Zeldovich, for their guidance and mentorship. Thank you to Austin Clements for your advice and many discussions. Finally, thank you to Stephen Tu for our collaborations.

Contents

Chapter 1

Introduction

Concurrent programs are a staple of high-performance applications from the system level to the user level. Writing correct concurrent programs is a tricky business. Assumptions that hold for one set of abstractions can fail to hold for others when attempting to increase the concurrency of an algorithm. For example, when writing concurrent algorithms using locks, it is often assumed that the program executes reads and writes in the order they are written. This is referred to as **Sequential Consistency** and it holds when using locks. Sequential Consistency is a type of memory model, which dictates how reads and writes may be re-ordered in a multi-threaded environment. Sequential Consistency is the strictest memory model, stating that reads and writes must be performed in exactly the order they are presented in by a program. There are weaker memory models, such as the Total-Store Order memory model, which allow some well defined re-ordering of reads and writes between threads. When attempting to get rid of locks, the assumption of Sequential Consistency may no longer hold and a previously correct algorithm can become incorrect due to read/write re-orderings.

When reasoning about concurrent algorithms, it becomes necessary to consider the possible thread interleavings, or schedules, and ensure that each one produces correct behavior. This quickly becomes infeasible with larger programs and thus tools and abstractions have been developed to address this issue. One such reasoning tool, called **Linearizability**, will be of particular interest for this thesis [5]. Linearizability is a correctness condition that can be applied at the method call, or application interface, level. It requires that method calls appear to occur instantaneously at some point between method invocation and return. Linearizability allows us to abstract reasoning about correctness to the API level of a concurrent algorithm, thus simplifying the process of reasoning about concurrent algorithms.

Once an algorithm has been expressed as an actual program, it becomes necessary to test to ensure that the reasoning and assumptions (as well as the translation to real code) are sound. Concurrent programs are notoriously difficult to test and debug. Even if all tests pass on a run of the program, there may be another schedule where some tests fail. Using traditional testing methods, such as unit testing, makes it difficult to cover enough of the schedule space to ensure that the tests will never fail. This is because we have no control over the scheduling and thus we are forced to re-run the program many times in the hopes of covering more of the schedule space. A different methodology, called Dynamic Model Checking, addresses this problem by being able to induce arbitrary schedules in a program. This allows us to test concurrent programs comprehensively.

1.1 Contributions

This thesis presents a tool, Codex, for the automatic dynamic checking of concurrent programs under the Total-Store-Order (TSO) memory model and a methodology for finding Linearizability violations automatically with the tool. Codex performs Dynamic Model Checking of C++11 code under the Total-Store-Order (TSO) memory model. This thesis applies Codex in the verification of lock-free algorithms under TSO. Codex found known TSO bugs using assertion (safety-condition) violations in Dekker's Algorithm, Peterson's algorithm, and the PostgreSQL WaitLatch synchronization primitive. The methodology presented for finding Linearizability violations is a procedure for using Codex to verify that a program does not exhibit behavior that would violate Linearizability. Using this methodology for finding Linearizability violations, Codex was used to find Linearizability violations in a simple lock-free concurrent hash-table and to verify Valois' lock-free concurrent linked list. This methodology can be used for other algorithms as well and takes steps towards easier verification of concurrent lock-free programs.

1.2 Outline

In Chapter 2, I go into more detail about Linearizability, memory models, and model checking as well as related work. In Chapter 3, I discuss the design of the model checking tool and the methodology for checking Linearizability. In Chapter 4, I evaluate the model checker and the methodology for checking Linearizability on example programs and in the final chapter I give my conclusions.

Chapter 2

Background

2.1 Reasoning About Correctness

In the following section, I will describe some useful concepts for discussing the correctness of concurrent algorithms. I will describe the kinds of algorithms that we are interested in (lock-free), the machine model in which we are considering these algorithms, and a useful correctness condition (Linearizability). These will be our tools for reasoning about correctness of concurrent algorithms.

2.1.1 Lock-free algorithms

The machine model I use in this thesis for concurrent algorithms involves threads that communicate through shared memory. The usual correctness condition used in this model is that of data-race freedom. This is often achieved through the use of locks which provide mutual exclusion in order to ensure there are no concurrent accesses to the same memory location. The use of coarse-grained locking to avoid data races is fairly well understood and thus simplifies the reasoning of concurrent programs. The problem with coarse-grained locking is that it often does not scale well with large numbers of locks [4]. Fine-grained locking strategies can improve on this, but are hard to design and may still require unnecessary sharing across CPU caches. One attempt to avoid the pitfalls of locking is to forego their use altogether in what are known as lock-free algorithms. These algorithms often require atomic hardware instructions such as Compare-and-Swap (CAS) and may contain benign data races. More formally, a method is lock-free if infinitely often some method call finishes in a finite number of steps [5].

2.1.2 TSO Memory Model

When we forego the use of locking and other synchronization primitives, the memory model of our machine becomes important. The memory model defines the acceptable set of return values for a read of a memory address. Sequential Consistency is the most basic memory model and is the one implicitly assumed by most programmers. Sequential Consistency states that a read returns the value last written to that address. Locks and other synchronization primitives implicitly enforce Sequential Consistency. We can then consider every possible interleaving of a concurrent algorithm and reason about its correctness as we would with a sequential algorithm. On the other hand, if we don't use locks and the machine's memory model is weaker than Sequential Consistency, it is possible to produce schedules not valid under any sequential interleaving. Thus a program which is correct under Sequential Consistency may be incorrect under a weaker memory model.

In this thesis, I consider the Total-Store-Order (TSO) memory model, used by the ubiquitous x86 architecture. More specifically, I use the machine model described by Owens et al. in X86-TSO which formally defines the behavior of a multi-core machine with the TSO memory model [10]. Figure 2-1 illustrates the machine model. The basic idea is that every core has a write buffer. When a thread (logically interchangeable with core) writes to an address, the write gets stored on the buffer for some arbitrary amount of time before getting flushed to the shared memory. When a thread performs a read, it first checks its own write buffer for the most recent write and then resorts to the shared memory if there isn't one. Therefore two threads may see different values for a read of the same address. Figure 2-2 depicts a simple lock-free algorithm that produces behavior not possible under Sequential Consistency, but possible under TSO.

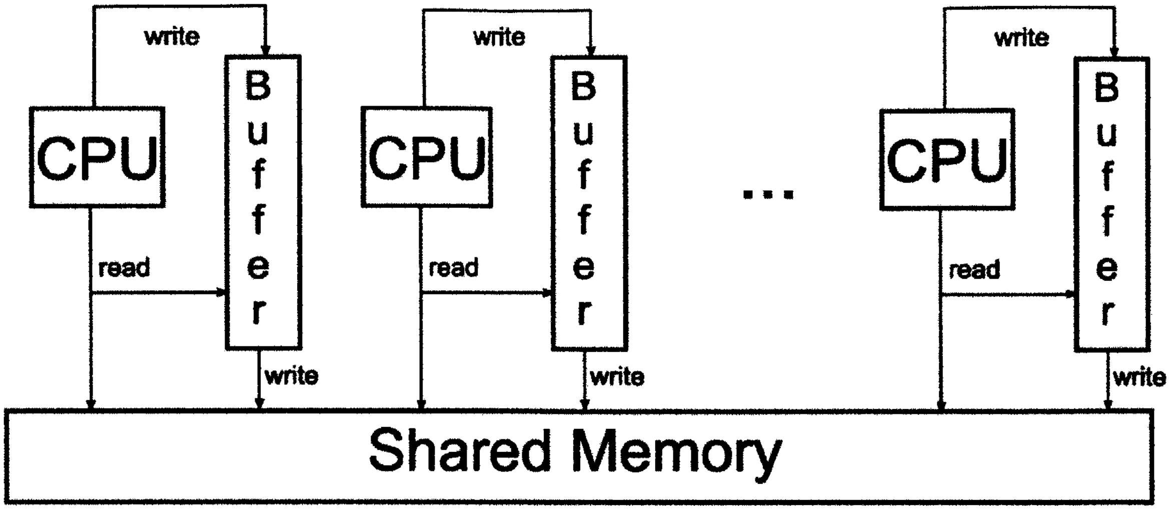

Figure 2-1: The x86-TSO Machine Model

init: x = y = 0;	
Thread 1:	Thread 2:

x=1;
print y;
	y=1;
	print x;

Figure 2-2: A simple lock-free algorithm. Allowed output under TSO: {0, 0}.

If we place a memory fence after each write in the program in Figure 2-2, which flushes the write buffers, we end up with sequentially consistent behavior. Atomic instructions like CAS also flush the write buffer.

2.1.3 Linearizability

Linearizability is the correctness condition most used in the literature when discussing lock-free algorithms [2]. Linearizability states that "each method call should appear to take effect instantaneously at some moment between its invocation and response" [5]. This property ensures that method calls behave atomically. Linearizability also has the property of composition. Composition is a handy property that further simplifies reasoning about concurrent algorithms by allowing smaller sub-problems to be tackled (in this case, individual methods instead of the program as a whole). In the context of lock-free algorithms, Linearizability is usually achieved in a few ways. If the method, or set of operations, does not make any changes (i.e. read-only) it is already linearizable. When we have a method that makes changes (i.e. writing a shared variable), it is often necessary to use atomic instructions like CAS or a memory fence to make the changes "take effect instantaneously". Indeed, researchers have shown that in some cases atomic instructions or memory fences are required for Linearizability [1].

2.2 Verifying Programs

Once we have a written program, it becomes necessary to test it to ensure it behaves as intended. For sequential programs, testing usually involves running the program once on a set of assertions and making sure those assertions are not violated. If the program passes, we can be assured that the program works as intended as long as our test cases are sufficiently comprehensive. In the concurrent setting, this kind of verification is insufficient. Even if the program passes all assertions once, it may fail them on a different run because a different interleaving has been induced by the machine. We need to ensure that the assertions would pass under any possible schedule. Dynamic Model Checking seeks to solve this issue.

2.2.1 Dynamic Model Checking

Dynamic Model Checking uses runtime scheduling to explore the state space of concurrent programs in order to verify them. The concurrent program is fed to the checker, which then executes it under a comprehensive set of schedules that guarantee correct execution under any possible schedule. Dynamic Model Checking is limited to programs that are deterministic, closed (i.e. self contained with assertions and input included), and in some cases partially re-written to use the instrumentation of the checker.

Dynamic Model Checkers are often able to make assertions about liveness and traces as well as user-level assertions (such as `foo!=NULL`). They can state if some code has deadlock or a data race (which in my experience is not very useful since most tend to be benign in practice). Dynamic Model Checking may sound like a panacea for the problems associated with concurrent testing and debugging, but there remains a glaring issue. The state space of schedules explodes exponentially with the size of the program. Currently, this makes Dynamic Model Checking impractical for all but the smallest of programs.

2.2.2 Dynamic Partial Order Reduction (DPOR)

Despite this issue, it turns out to be unnecessary to run all schedules. In practice, if a bug exists, it is found very quickly when the state space is explored in order of increasing thread switches. This agrees with the intuition that most concurrency bugs can occur between only two threads. However, in order to verify the program completely, the rest of the state space still needs to be covered.

One approach to overcome this challenge is approximation. That is, sample the state space and verify the program approximately [2]. The problem with this approach is that it is difficult to tell how good the approximation is. Another strategy is to reduce the set of schedules that need to be executed by only looking at the important thread transitions. Dynamic Partial Order Reduction (DPOR) is an algorithm by Flanagan et al. for reducing the schedules that need to be explored, since many schedules turn out to be redundant [3]. For example, there is no reason to verify a schedule that re-orders two reads from a verified schedule. DPOR points out that the re-orderings that really matter are between writes to the same shared variable, more generally referred to by Flanagan as dependent transitions. A read and a write to the same variable is also a dependent transition. Most other transitions, such as writes to different variables, are independent and thus do not need to be re-ordered and checked.

2.3 Related Work

CHESS is the project that originally inspired this thesis [9]. It is a Dynamic Model Checker which can check user assertions and liveness conditions. It does not include weaker memory models. Line-up is a follow up project which performs Linearizability checks on top of CHESS [2]. It does not consider weaker memory models because it relies on CHESS. The authors mention that Line-up is memory model agnostic, so presumably it would work if using a checker which included other memory models. Like my project, Line-up seems to require less manual work than other tools to check Linearizability. This is achieved by comparing "observations" (presumably program traces) from the sequential and concurrent run of the program. There has also been work on model checking under the TSO memory model using code rewriting to model TSO [8], but not discussing correctness conditions like Linearizability directly.

Eddie Kohler's work and notes on Read-Copy Update (RCU) has also been invaluable to this thesis [6]. Kohler points out the limitations and costs of Linearizability and describes how RCU avoids them. Of course, RCU is not without its own costs such as the additional memory often required for shadow copies, but RCU represents a viable alternative approach to the problem of concurrent algorithms. Kohler's notes do much to clarify the notion of Linearizability and his hash table examples are a direct inspiration to the simple hash table used in the evaluation section of this thesis.

Chapter 3

Design

The following sections describe the design of Codex, the Dynamic Model Checker, and the methodology for finding Linearizability violations. Codex is a software system implemented in C++ while the process for finding Linearizability violations is a methodology that uses Codex.

3.1 Codex

The Codex model checker consists of a schedule generator and a schedule executor. The executor and generator have a Client-Server architecture where the executor is the client and the generator is the server. The diagram in Figure 3-1 depicts Codex's architecture.

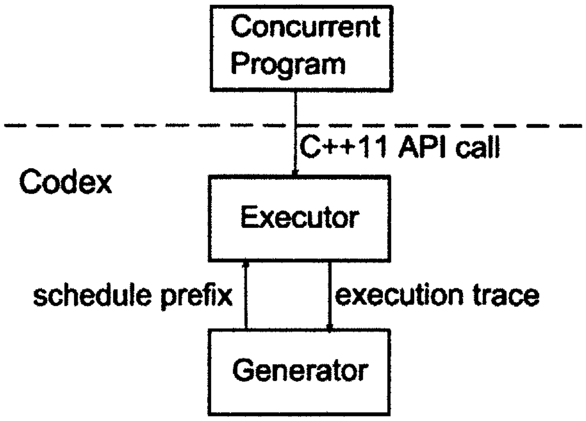

Figure 3-1: Codex's Client-Server Architecture.

I built the C++ executor for Codex that can run standard (albeit strict) C++11 code under a Sequentially Consistent or TSO memory model. The Codex executor implements a runtime scheduler by hooking the C++11 threading and atomic library calls (C++11 is used because it is the first C++ standard to have proper multi-threading and atomic support). This allows Codex to capture any program behavior related to concurrency, as long as the program is properly (strictly) written in C++11. The executor's runtime scheduler works by receiving some schedule prefix from the generator, replaying the prefix, running the program to completion, and sending back the completed execution trace to the generator. This is repeated until the generator runs out of schedules to run or a violation occurs. Codex can run in Sequential Consistency mode or TSO mode in order to model the respective memory model. Codex also supports a basic form of transactional memory which is fairly straightforward because we run programs in a single-threaded fashion. Codex uses a generator that implements DPOR to generate schedules.

3.1.1 Modeling Sequential Consistency

The executor runs the schedule in a single-threaded manner. That is, it only allows one thread to run at a time. A user level thread is modeled by a C++11 thread under the control of the runtime scheduler. This allows us careful control of the thread interleavings. Any shared variables must use the atomic C++11 library, which Codex hooks into to capture reads and writes. Since Codex controls threads and shared variables using locks and condition variables, Sequential Consistency is enforced.

3.1.2 Modeling TSO

In order to model TSO write-buffers, every thread in Codex has a separate buffer thread that it sends its writes to. The buffer thread is merely a thread that, while running, will perform the writes that have been added to its FIFO structure. The generator will interleave these buffer threads as it would any other kind of thread which produces behavior equivalent to TSO. Threads can also read from their buffer threads which is a required TSO behavior. In order to verify that this implementation models TSO correctly, I implemented each of Owen's x86-TSO litmus tests [10]. The litmus tests are a series of ten tests that cover all of the behavior observable under TSO and not Sequential Consistency. Codex produces all of the correct behaviors when used to run the litmus tests.

3.2 Testing for Linearizability Violations

At first, my goal was to study the kinds of user assertions that one would want to check in order to verify lock-free programs under TSO. While finding user assertion violations is useful, they can be rather non-intuitive and difficult to come up with when looking for TSO bugs. Looking for user assertion violations merely moves the difficulty from reasoning about lock-free programs to coming up with these assertions. I find it to be more useful to provide a more general and simpler method for finding TSO bugs. This is where Linearizability violations come in.

Linearizability is checked at the method, or API, level. The program is first run with each method call being atomic. Codex achieves this with its transactional memory capability. Codex logs the output of each method call for each schedule. This means that the program has to contain some calls with return values. The program is then run under Sequential Consistency. Codex makes sure that the return values match the logs from the atomic run to ensure the program is correct under Sequential Consistency. Finally, the program is run again under TSO. The return values should never produce an output not in the log of the atomic run and all of the return values in the log should be output by some schedule under TSO and Sequential Consistency. Running the program with atomic method calls gives all of the possible return values under a linearizable implementation of the algorithm. If a whole method call is atomic, it is linearizable by definition. If TSO or Sequential Consistency runs fail to match the log, they have produced behavior not possible under Linearizability and thus have a violation. If the violation occurs on the Sequential Consistency run and the algorithm is linearizable, likely some implementation error occurred. If the violation occurs on the TSO run, but not the Sequential Consistency run, the algorithm likely failed to take into account TSO or made an incorrect assumption about the TSO model.

Chapter 4

Evaluation

In order to evaluate Codex, there are two questions to ask: "Can Codex find violations?" and "How does Codex perform when verifying programs?". I first check that Codex can find user assertion violations in general. Then I use Codex to find Linearizability violations, a specific kind of user assertion violations, on two concurrent lock free data structures. Finally, I evaluate the performance of Codex verifying the programs.

4.1 Finding User Assertion Violations Using Codex

To show that Codex can find user assertion violations, I verified the failure of some known TSO bugs and the success of their fixes.

4.1.1 Dekker's Algorithm and Peterson's Algorithm

I started with two of the simplest concurrent lock-free algorithms. Dekker's algorithm and Peterson's algorithm are two related algorithms for providing mutual exclusion without locks (indeed, they can be used as the implementation for a basic lock). The basic algorithm for Dekker's on two threads is shown in Figure 4-1. The actual program has an assertion that fails if the critical sections are interleaved. Codex verifies the correctness of the program under TSO. If the fences are removed, the

```
                    int turn = 0;
               int flag0 = flag1 = false;
Thread 0:                       | Thread 1:
flag0 = true;                   | flag1 = true;
thread_fence();                 | thread_fence();
while (flag1 == true) {         | while (flag0 == true) {
  if(turn != 0) {               |   if(turn != 1) {
    flag0=false;                |     flag1 = false;
    while (turn != 0) {         |     while (turn != 0) {
      //busy wait               |       //busy wait
    }                           |     }
    flag0 = true;               |     flag1 = true;
  }                             |   }
}                               | }
                                |
// critical section             | // critical section
turn = 1;                       | turn = 0;
thread_fence();                 | thread_fence();
flag0 = false;                  | flag1 = false;
```

Figure 4-1: Dekker's algorithm for two threads including necessary fences.

program is correct under Sequential Consistency, but the assertion is violated under TSO. Peterson's algorithm is treated similarly. One small issue that Codex takes care of, in the case of these two algorithms, is that they are allowed to spin indefinitely. Codex automatically contracts loops to avoid infinite schedules. For example, if a loop iterates twice without writing, it is enough to consider the schedule where it only iterates once.

4.1.2 TSO Bug in PostgreSQL WaitLatch

I also used Codex to verify a PostgreSQL TSO bug discovered in 2011 [7]. The bug was in the WaitLatch synchronization primitive. Figure 4-2 shows how the WaitLatch primitive is meant to be used.

The bug is due to the implementation of `SetLatch()`, which returns quickly if the latch is already set. Without memory barriers under TSO, if `SetLatch()` returns without performing a write then it may look to another thread as if `SetLatch();` and `work_to_do=true;` are re-ordered. This can lead to the interleaving in Figure

Waiter:	Waker:
for(;;)	
{	work_to_do = true;
ResetLatch();	SetLatch();
if(work_to_do)	
doStuff();	
WaitLatch();	
}	

Figure 4-2: Usage of WaitLatch

4-3 which results in a lost wake-up for the Waiter thread.

Interleaving:
SetLatch();
ResetLatch();
if(work_to_do) // evaluates to false
work_to_do = true;
WaitLatch();

Figure 4-3: Possible WaitLatch Interleaving under TSO.

Codex finds this bug in the form of finding the violation of the following assertion:

```
assert( !latch_has_been_set or isSetLatch() or work_done );
```

If the assertion is violated, it means that we are about to enter `WaitLatch()` where the latch has been set by the Waker thread, reset by the Waiter, and the work was not done (the if-statement evaluated to false). This results in the Waker thread's signal having been handled by the Waiter thread, the Waiter thread having done no work, and the Waiter thread about to enter the wait call. Thus the wake-up has been effectively skipped.

One solution is to have the WaitLatch user place a lock around the `work_to_do` variable. While solving the issue, this solution is based on adding a convention and changing the WaitLatch contract. Another solution involves getting rid of the optimization, thus having the write to an internal variable occur every time. This also solves the bug. My solution places a memory fence in the `SetLatch()` call implementation. All three solutions prevent the re-ordering in Figure 4-3, although the solution that gets rid of the optimization results in no memory barriers. All three solutions can be verified by Codex.

4.2 Finding Linearizability Violations Using Codex

To show that Codex can find Linearizability violations, I provide two examples. In the first, I consider a very simple lock-free concurrent hash table with both linearizable and non-linearizable implementations. Secondly, I implemented one of the early lock-free algorithms, Valois' lock-free concurrent linked list, and verified that it is linearizable [11].

4.2.1 A Simple Lock-Free Concurrent Hash Table

Consider the most trivial concurrent hash table that stores integers. The C++ class definition looks like:

```
class hash_table {
private:
  const size_t size = 1 << 20;
  atomic<int> _data[size];

public:
  void put(int key, int val);
  int get(int key);
};
```

The `get()` can be implemented as:

```
int get(int key) {
  return _data[key].load(memory_order_relaxed);
}
```

The `memory_order_relaxed` argument tells the C++ atomic library to not put in any fences. Whether or not this hash table will be linearizable depends on the implementation of `put()`. The following implementation is not linearizable:

```
void put(int key, int val) {
  _data[key].store(val,memory_order_relaxed);
}
```

If we run the program in Figure 4-4, Codex finds a Linearizability violation in the TSO run although it passes the sequentially consistent run. Under the TSO run, it is possible to get r_a = false and r_b = false which reveals a Linearizability violation. Codex catches this violation because that result does not occur during the atomic run (it also does not occur during the sequentially consistent run). In order to make this simple hash table linearizable, we can place a fence in the put() call after the store (removing the memory_order_relaxed argument to store is equivalent). This implementation passes our program in Figure 4-4 without violation.

The table values are initialized to 0. x and y are non-zero.	
Thread A	Thread B

table_ptr->put(0,x);
r_a = (table_ptr->get(1) == y);
	table_ptr->put(1,y);
	r_b = (table_ptr->get(0) == x);

Figure 4-4: Program run to test our hash table

4.2.2 Valois' Lock-Free Concurrent Linked List

For a less trivial example, I implemented Valois' lock-free concurrent linked list as described in [11]. Rather than pure loads, stores, and fences the implementation relies on the atomic Compare-and-Swap (CAS) operation. CAS is an atomic operation that is provided by most modern hardware. Its method signature is usually something like this:

```
bool CAS(T* address, T expected_value, T new_value);
```

where T is the appropriate type. CAS will store new_value at address if and only if the current value at address is expected_value. The CAS operation performs a read, a store, and a fence atomically.

The implementation is linearizable when run with programs similar to the one in Figure 4-4. If I instead replace CAS with a conditional store, Codex finds the

Dekker's Algorithm	62,719 schedules
Peterson's Algorithm	24,453 schedules
PostgreSQL WaitLatch Algorithm	63 schedules

Figure 4-5: Number of Schedules Required to Verify User Assertions

	Lock-Free Hash Table	Lock-Free Linked List
Atomic/Linearizable run	4 schedules	4 schedules
Sequential Consistency run	4 schedules	210 schedules
TSO run	14 schedules	701 schedules

Figure 4-6: Number of Schedules Required to Verify Linearizability.

expected violations. This is because the CAS call serves as the point in which the method calls' effect "instantaneously" takes place, often called the linearization point.

4.3 Codex Performance

When considering the performance of Codex, the number of schedules is the most important factor. Each schedule is executed fairly quickly, but the number of total schedules required to verify the program is what increases running time. For instance, when running the programs considered in this thesis, Codex executes 100 schedules per second on average (Codex profiles this information). This performance can be increased on a multi-core machine because executing schedules is parallelizable and Codex's architecture lends itself to running multiple executors with one schedule generator. Figures 4-5 and 4-6 show the number of schedules required to verify the programs discussed in this section. These numbers are relatively small because the programs in question are relatively small. The growth of the number of schedules with the size of the program is a current challenge.

It should be noted that Dekker's algorithm and Peterson's algorithm have a high number of schedules due to the interactions of their nested loops. These numbers also point out how TSO runs require more schedules than Sequential Consistency.

Chapter 5

Conclusions

Dynamic Model Checking provides a way to verify and debug concurrent programs. This is particularly useful and needed when programming lock-free algorithms. Furthermore, it is necessary for the model checker to take into account weaker memory models like TSO, not just Sequential Consistency, when checking lock-free programs. Otherwise the verification is not useful when the goal is to run the code on real machines.

That said, there are still challenges to overcome. Dynamic Model Checking makes it feasible to correctly test and debug code. However, as long as the number of schedules to check remains impractically large for all but the smallest programs, Dynamic Model Checking will not be relevant for industry-sized applications.

5.1 Future Work

An obvious way to proceed with this work would be to better automate Codex with the methodology for finding Linearizability violations. In its current state, the process is fairly manual. One must annotate the method calls and run Codex three separate times under different conditions (atomic, Sequential Consistency, and TSO). Another avenue would be to implement other weak memory models. The C++11 memory model in particular would be useful to be able to verify using Codex. The C++11 memory model will likely gain more use as C++11 becomes the standard C++ and the memory model is notoriously difficult to understand if not working under the default Sequential Consistency settings.

Bibliography

[1] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *ACM SIGPLAN Notices*, volume 46, pages 487–498. ACM, 2011.

[2] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In *ACM Sigplan Notices*, volume 45, pages 330–340. ACM, 2010.

[3] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.

[4] Keir Fraser. *Practical lock-freedom*. PhD thesis, PhD thesis, University of Cambridge Computer Laboratory, 2004.

[5] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2011.

[6] Eddie Kohler. Notes on read-copy update. `http://read.seas.harvard.edu/cs261/2011/rcu.html`. Accessed: 2013-5-18.

[7] Tom Lane. Yes, waitlatch is vulnerable to weak-memory-ordering bugs. Mailing list. pgsql-hackers@postgreSQL.org., 08 2011. Accessed 5, 2013.

[8] Carl Leonardsson. *Thread-Modular Model Checking of Concurrent Programs under TSO using Code Rewriting*. PhD thesis, Uppsala University, 2010.

[9] Madan Musuvathi and Shaz Qadeer. Chess: Systematic stress testing of concurrent software. In *Logic-Based Program Synthesis and Transformation*, pages 15–16. Springer, 2007.

[10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[11] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.