1-1-2006

Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study

Sebastian Burckhardt
University of Pennsylvania

Rajeev Alur
University of Pennsylvania, alur@cis.upenn.edu

Milo Martin
University of Pennsylvania, milom@cis.upenn.edu

Recommended Citation

Sebastian Burckhardt, Rajeev Alur, and Milo Martin, "Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study", *Lecture Notes in Computer Science: Computer Aided Verification* 4144, 489-502. January 2006. http://dx.doi.org/10.1007/11817963_45

From the 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006.

Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study

Abstract

Many multithreaded programs employ concurrent data types to safely share data among threads. However, highly-concurrent algorithms for even seemingly simple data types are difficult to implement correctly, especially when considering the relaxed memory ordering models commonly employed by today's multiprocessors. The formal verification of such implementations is challenging as well because the high degree of concurrency leads to a large number of possible executions. In this case study, we develop a SAT-based bounded verification method and apply it to a representative example, a well-known two-lock concurrent queue algorithm. We first formulate a correctness criterion that specifically targets failures caused by concurrency; it demands that all concurrent executions be observationally equivalent to some serial execution. Next, we define a relaxed memory model that conservatively approximates several common shared-memory multiprocessors. Using commit point specifications, a suite of finite symbolic tests, a prototype encoder, and a standard SAT solver, we successfully identify two failures of a naive implementation that can be observed only under relaxed memory models. We eliminate these failures by inserting appropriate memory ordering fences into the code. The experiments confirm that our approach provides a valuable aid for desigining and implementing concurrent data types.

Disciplines

Computer Engineering | Computer Sciences

Comments

From the 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006.

Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study*

Sebastian Burckhardt, Rajeev Alur, and Milo M.K. Martin

Department of Computer Science
University of Pennsylvania
{sburckha, alur, milom}@cis.upenn.edu

Abstract. Many multithreaded programs employ concurrent data types to safely share data among threads. However, highly-concurrent algorithms for even seemingly simple data types are difficult to implement correctly, especially when considering the relaxed memory ordering models commonly employed by today's multiprocessors. The formal verification of such implementations is challenging as well because the high degree of concurrency leads to a large number of possible executions. In this case study, we develop a SAT-based bounded verification method and apply it to a representative example, a well-known two-lock concurrent queue algorithm. We first formulate a correctness criterion that specifically targets failures caused by concurrency; it demands that all concurrent executions be observationally equivalent to some serial execution. Next, we define a relaxed memory model that conservatively approximates several common shared-memory multiprocessors. Using commit point specifications, a suite of finite symbolic tests, a prototype encoder, and a standard SAT solver, we successfully identify two failures of a naive implementation that can be observed only under relaxed memory models. We eliminate these failures by inserting appropriate memory ordering fences into the code. The experiments confirm that our approach provides a valuable aid for desigining and implementing concurrent data types.

1 Introduction

Shared-memory multiprocessor architectures dominate the server and scientific computing market today and are even finding their way into desktop, laptop and gaming machines. Nevertheless, programming such systems remains a challenge [1]. To cope with the subtleties of concurrent program executions, software architects often introduce abstraction layers in the form of *concurrent data types*.

Concurrent data types provide familiar data abstractions (such as queues, hash tables, or trees) to client programs that have concurrently executing threads. The interface of the data type specifies the operations. The implementation provides the actual code for the operations; it hides the concurrency from the client program, using lower-level synchronization primitives such as locks or semaphores as needed. To allow for more concurrency and better performance, optimized implementations use fine-grained locking or even avoid locks altogether by using lock-free synchronization techniques [2, 3, 4].

* Supported partially by NSF awards CCR 0306352 and CNS 0524059 and donations from Intel Corporation.

T. Ball and R.B. Jones (Eds.): CAV 2006, LNCS 4144, pp. 489–502, 2006.

Writing correct and efficient code for concurrent data types is challenging. To make matters worse, many contemporary shared-memory architectures use *relaxed memory ordering models* [5]. For example, a processor may execute memory accesses in a different order than specified by the program, and stores may take effect locally before becoming visible to remote processors. Although regular "fully synchronized" programs are not sensitive to the memory model, implementations that contain concurrency optimizations (such as intentional data races or lock-free synchronization) become exposed to such ordering and atomicity relaxations. Because the resulting executions are counterintuitive and nondeterministic, even highly skilled engineers are likely to make programming errors when relying on informal reasoning and conventional testing only, which motivates the use of formal verification.

The operations of the concurrent data type are invoked by a multi-threaded client program and may execute concurrently on a multiprocessor. Our correctness criterion is *operation-level sequential consistency*. It requires that all concurrent executions be observationally equivalent to a *serial* execution, that is, an execution in which the operations execute atomically and in the order they are invoked by each thread. As we assume that all serial executions reflect the semantics of the abstract data type correctly (which can be verified independently using standard techniques for sequential programs), correctness in our sense implies that client programs always observe the correct semantics. In particular, the data type is guaranteed to appear sequentially consistent to the client program even if the underlying multiprocessor executions are not sequentially consistent [6] on the instruction level.

To bound the number of threads, the state space, and the depth of the execution, we consider client programs that make a fixed number of operation calls only. We call these bounded instances *symbolic tests*. Furthermore, the user must specify *commit points* [7], that is, single out an instruction within each operation such that the logical order of the operations always matches the execution order of their commit points. We qualify soundness and completeness of our approach as follows: (a) it can prove correctness for all executions of the given symbolic test, and (b) it generates counterexamples that are sound with respect to the chosen memory model and commit point specification.

We encode the existence of a violating execution as a CNF instance that can be solved or refuted by a standard SAT solver (corresponding to cases (b) and (a) above). Our encoding combines several ideas that appear in prior work, such as loop unrolling and SSA transformations [8] and axiomatic memory model encodings [9, 10].

We successfully applied our method to an example that represents optimized implementations of concurrent data types, the two-lock concurrent queue by Michael and Scott [11]. First, we verified that the implementation code is correct for all symbolic tests in our suite when executed on a sequentially consistent memory model. Next, our prototype found two failures that can occur when the same code is executed on a relaxed memory model. Guided by the counterexamples, we identified the problematic instruction reorderings and prevented them by inserting two memory ordering fences. Finally, we verified that with these fences, the code executes correctly on a relaxed memory model for all symbolic tests in the suite.

1.1 Related Work

Most prior work on formal verification of concurrent data types assumes a sequentially consistent memory model [12, 13, 14]. In that context, linearizability [15] is the correctness criterion of choice. Unfortunately, its definition assumes that an execution globally orders the operation invocations and returns, which is not well defined on relaxed memory models because instructions may be reordered across operation boundaries.

Model checking of assembly code snippets for relaxed memory models was first attempted with explicit state enumeration [16, 17] using an operational memory model and interleaving concurrency. More recently, constraint-based encodings of axiomatic memory models have been proposed for memory-model sensitive race detection [9]. Our approach differs because we specifically target concurrent data types and because we use operation-level sequential consistency as our correctness criterion.

2 The Challenge

Our verification target is the two-lock FIFO queue implementation [11] by Michael and Scott (Fig. 1). We chose this example because of its optimized use of locks: the enqueue and dequeue operations can proceed concurrently because they use independent locks. This concurrency improves performance, but it also introduces a race condition if the queue is empty. Race conditions sometimes indicate an improper locking discipline [18], but as we see here, they may also be a side effect of concurrency optimizations.

We encountered several challenges in the course of our case study:

Avoiding State Explosion. An interleaving model of concurrency can lead to large state spaces; relaxed memory models exacerbate this effect because they introduce additional concurrency at the instruction level. Therefore, we decided against unrolling the transition relation and representing executions as global state sequences. Instead, we represent the program executed by each thread as a linear symbolic instruction stream, and we encode the relative order of instructions using SAT variables.

Defining Memory Models. We compared the memory model specifications for the IBM PowerPC [19], Sun SPARC v9 TSO/PSO/RMO [20], Alpha [21], and IBM zArchitecture [22]. Although there are many differences, the specifications use similar rules (axioms) to describe the valid memory orderings. By comparing the axioms, we derived a generic relaxed memory model (to be defined in section 3.4) that provides a common conservative approximation and abstracts unneeded details.

Encoding Memory Models. We can encode the memory model axioms directly because we have explicit representations of the instruction streams for each thread [9, 23]. In contrast, classic interleaving models based on labeled transition systems require a prior conversion of the axiomatic specification into an operational style [16, 17].

Bounding Instances. To achieve a bounded formulation, we approximate admissible client programs using a manually constructed suite of symbolic tests. Each test specifies a fixed, finite sequence of symbolic operation invocations for each thread. Unlike deterministic tests, a symbolic test covers all possible instruction interleavings and

```
structure node_t {
  value: value_t;
  next: ptr to node_t
}

structure queue_t {
  head: ptr to node_t;
  tail: ptr to node_t;
  headlock: lock_t;
  taillock: lock_t;
}

1  initialize(Q: ptr to queue_t)
2    // Make dummy node
3    node = new_node()
4    node->next = NULL
5    Q->head = Q->tail = node
6    Q->headlock = FREE
7    Q->taillock = FREE
8
9  enqueue(Q: ptr to queue_t,
10           value: value_t)
11   node = new_node()
12   node->value = value
13   node->next = NULL
14   lock(&Q->taillock)
15     Q->tail->next = node
16     Q->tail = node
17   unlock(&Q->taillock)

27 dequeue(Q: ptr to queue_t,
28           pvalue: ptr to value_t)
29         : boolean
30   lock(&Q->headlock)
31   node = Q->head
32   new_head = node->next
33   if new_head == NULL
34   // queue empty
35     unlock(&Q->headlock)
36     return false
37   endif
38   *pvalue = new_head->value
39   Q->head = new_head
40   unlock(&Q->headlock)
41   free(node)
42   return true
```

Fig. 1. Michael and Scott's two-lock queue implementation [11]. The queue is represented by a dynamically allocated singly linked list with head and tail pointers, each protected by a separate lock. To simplify the empty queue case, the first node of the linked list is a "dummy" element: its value is not part of the queue.

reorderings and all possible call arguments and return values. The total number of instructions executed during a test is bounded because the operations do not contain loops. As a result, each test has a finite (albeit exponential) number of possible executions, which explains how we avoid the undecidability of sequential consistency [24].

Representing Parameters. The implementation is parameterized by (a) the number of threads, (b) the size of the queue, (c) the size of the instruction reordering window, and (d) the number of distinct data values. As our formulation targets individual symbolic tests with finitely many executions, we can easily find static bounds. For instance, the number of threads is explicitly specified by the test, the queue size and the number of data values never exceed the number of "enqueue" calls, and the instruction reorder window need not be larger than the total number of instructions.

Avoiding Mixed Quantifiers. Our correctness criterion contains alternating quantifiers (we ask if there exists a observationally equivalent serial execution for each concurrent execution), which can not be directly encoded in SAT. We avoid this problem (at the expense of some generality and automation) by asking the user to designate one instruction for each operation to be the *commit point*. If correctly specified, the order in which the commit points execute matches the logical order of the operations. With this additional information, we can construct a deterministic serial reference execution for each concurrent execution. If the two executions are observationally equivalent in all cases, we have shown that the implementation is sequentially consistent. If not, our tool provides a counterexample trace that shows both executions, which may point out an actual defect in the implementation or an incorrect commit point specification.

Making Memory Accesses Explicit. The original algorithm (Fig. 1) uses a pseudo-code notation similar to C. To accurately model synchronization instructions and the effects of the memory model, we require a lower-level representation that makes the loads and stores explicit. Our back-end prototype accepts a loop-free imperative intermediate language that has (a) a small syntax, (b) a well-defined semantics even for weak memory models, and (c) supports modelling of spin loops, atomic blocks, and assertions.

Translating the Code. We envision a tool that includes a front end that accepts a subset of C and performs the translation automatically. However, for this case study, we used a straightforward manual translation of the pseudo-code into our tool's intermediate language.

Modelling Locks and Detecting Deadlocks. The code for the two-lock queue makes calls to `lock()` and `unlock()` without fully specifying their memory ordering semantics. For reference, we use a lock implementation from an architecture manual [20] that contains a spin loop, an atomic load-store primitive, and (partial) memory ordering fences. We use a reduction for side-effect free spin loops that allows us to model a single iteration of the spin loop only, while still covering all executions and detecting all deadlocks caused by an improper locking discipline in the implementation.

Modelling Dynamic Memory Management. To model dynamic memory allocation, we create an array of blocks, each with its own lock. The allocation call nondeterministically selects a free block and locks it. The deallocation call unlocks it again. The array size is bounded by the number of "enqueue" calls in the symbolic test.

3 Solution

In this section, we formalize symbolic tests and our correctness criterion, we show how to prove correctness or provide a counterexample for a given commit point specification, and we formally define our memory model.

3.1 Symbolic Tests

A symbolic test $T(A, B)$ specifies a finite sequence of operation invocations for each thread. A is a set of symbolic variables that represents argument values passed to the

$T(A, B)$	
thread 1:	thread 2:
$(b_1, b_2) = \mathit{dequeue}()$	$\mathit{enqueue}(a_2)$
$\mathit{enqueue}(a_1)$	$(b_5, b_6) = \mathit{dequeue}()$
$(b_3, b_4) = \mathit{dequeue}()$	

$A = \{a_1, a_2\}$ and $B = \{b_1, \ldots, b_6\}$

Meaning of the operations:

— $\mathit{enqueue}(v)$
adds value v to the queue

— $\mathit{dequeue}()$ returns values (r, v)
if queue is empty, returns $r = \mathit{false}$;
otherwise, returns $r = \mathit{true}$ and the dequeued value v

Fig. 2. An example for a symbolic test $T(A, B)$

operations, and B similarly represents values returned by the operations. For our queue example, a symbolic client program $T(A, B)$ may look as in in Fig. 2.

For a given symbolic test $T(A, B)$, let V_A be the set of valuations to the variables in A, and let V_B the set of valuations to the variables in V_B. Given an implementation I, a memory model Y, and a symbolic test $T(A, B)$, we define the set $R_{T,I,Y} \subset V_A \times V_B$ to consist of all tuples (a, b) such that it is possible to observe the output values b when executing the test T with implementation I and input values a on a machine with memory model Y.

Let Π_T be the set of all total orders on the invocations in T. We say an order $o \in \Pi_T$ is *consistent* with T (written *consistent*$_T(o)$) if and only if for all invocations made by the same thread, the order in T matches the order o. Define the function $g_{I,T} : \Pi_T \times V_A \to V_B$ such that $g_{I,T}(o, a)$ describes the return values that result from executing the invocations appearing in T in a single thread, in the order specified by o, and with input values a. We guarantee that $g_{I,T}$ is a well-defined function as follows:

1. We admit only implementations I whose single-threaded executions are deterministic. Where we want nondeterminism (such as for modelling memory allocation), we express it by declaring additional symbolic input values.
2. We assume that executions never deadlock. However, because deadlocks are well possible in practice, we discharge this assumption separately by performing a prior check for deadlocks using an independent SAT instance (which we do not describe further here).

With the formalism introduced above we can now precisely define operation-level sequential consistency for a given test T.

Formulation. The implementation I is correct for a given symbolic test T and a memory model Y if and only if for all $(a, b) \in R_{T,I,Y}$, there exists an invocation order $o \in \Pi_T$ such that o is consistent with T and $b = g_{I,T}(o, a)$.

If an implementation is correct for all symbolic tests T, it is guaranteed to be free of defects that are caused by concurrency; if it contains any other errors, those are guaranteed to manifest themselves in some serial execution, and can therefore be easily covered with conventional verification methods.

3.2 Encoding Concurrent Executions

Our first subgoal is to encode the concurrent executions in a way that is suitable for SAT solving. We show in this section how to define auxiliary variables C, M and a formula $\Phi_{T,I,Y}(A, B, C, M)$ such that for all $(a, b) \in V_A \times V_B$ the following holds:

$$(a, b) \in R_{T,I,Y} \quad \Leftrightarrow \quad \exists C : \exists M : \Phi_{T,I,Y}(a, b, C, M) \tag{1}$$

The variable M represents the memory order; different valuations to M correspond to different instruction interleavings (and possibly reorderings). C is a set of variables that represent intermediate values of the computation. Each variable that represents an input, intermediate, or return value is local to a thread k, and we partition $A = \bigcup_k A_k$, $B = \bigcup_k B_k$, $C = \bigcup_k C_k$ accordingly. The formula $\Phi_{T,I,Y}$ then decomposes

(a) Implementation code for the operation *func*	(b) Symbolic instruction stream for the expanded invocation $y = \mathit{func}(x)$

var arr : **array**[8] **of** int
 	
op func(int index) **returns** int
  if (index < 0) **then**
  return 0
  else
  return arr[index]
  endif
endop
	move (x < 0), c
	[+c] move 0, r1
	[-c] load arr[x], r2
	move (c ? r1 : r2), y

(c) Corresponding formula over $A_k = \{x\}, B_k = \{y\}, C_k = \{c, r1, r2\}$
$\Delta(A_k, B_k, C_k) \;\equiv\; (c = (x < 0)) \;\wedge\; (r1 = 0) \;\wedge\; ((c \wedge (y = r1)) \vee (\neg c \wedge (y = r2)))$

Fig. 3. Example of the thread-local encoding

into subformulas that represent the communication and the thread-local components separately:

$$\Phi_{T,I,Y}(A, B, C, M) \;\equiv\; \Theta_{T,I,Y}(M, C) \;\wedge\; \bigwedge_k \Delta_{T,I,k}(A_k, B_k, C_k) \qquad (2)$$

The Thread-Local Formulas. For each thread k, the formula $\Delta_{T,I,k}$ captures the connection among input values A_k, intermediate values C_k, and return values B_k: the solutions to $\Delta_{T,I,k}(A_k, B_k, C_k)$ correspond to all possible executions of thread k in an unspecified environment (that is, for arbitrary values returned by the load instructions). We obtain the encoding as follows (see Fig. 3 for an example):

- Expand the invocation sequence for thread k specified in $T(A, B)$ by inlining the implementation code I.
- Unroll loop iterations. We can skip this step for this case study (and avoid the associated loss of precision) because the implementation code is already loop-free.
- Compile the code into a linear, finite instruction sequence consisting of loads, stores, fences, and instructions that capture the thread-local computations. We call the latter *move* instructions.
- Create a variable in C_k for each intermediate value produced by a load or move.
- For each move instruction, create constraints on the source and destination values that express the nature of the computation. Take the conjunction of these constraints to get the formula $\Delta_{T,I,k}$.
- If the code contains conditionals, use *predicates* to express conditional execution of instructions. For each instruction i, define the predicate $\pi(i)$ to be a boolean formula over variables in C_k that captures the condition(s) under which this instruction gets executed. Fig. 3 illustrates how to use predicates; we skip the further details of the compilation algorithm here.

The Communication Formula. The formula $\Theta_{T,I,Y}(M, C)$ encodes the valid interactions between the threads as they execute load, store, and fence instructions. It thus captures the shared memory semantics of the multiprocessor, which is defined by the memory model Y.

To encode $\Theta_{T,I,Y}$, we first create predicated instruction streams for each thread as described in section 3.2. Let X be the set of all loads and stores appearing in these streams. Let Π_X be the set of all total orders on X. Define the *memory order* variable M to range over Π_X. We can now encode $\Theta_{T,I,Y}$ such that its solutions have the following properties: (a) the value loaded by a load matches the last value stored to the same address (where "last" is interpreted in terms of the memory order M), and (b) the memory order M follows the ordering axioms of the memory model.

We give a full definition for the formula $\Theta_{T,I,\mathit{Relaxed}}$ describing our relaxed memory model in section 3.4; in the remainder of this section we discuss the similar but somewhat simpler case of a sequentially consistent multiprocessor only. For each memory access $x \in X$, let $\pi(x)$ be its predicate (a boolean formula over the variables in C that captures the condition under which x gets executed), and let $a_x, v_x \in C$ be the variables that represent the address and data value of x, respectively. Let $L \subset X$ be the set of loads, and $S \subset X$ be the set of stores. Let $<_p$ be the program order; that is, $<_p$ is a partial order on X such that $x <_p y$ if and only if x, y are appear in the same stream, and x comes before y. Then

$$\begin{aligned}
&\Theta_{T,I,\mathit{SeqCons}}(M, C) \quad \equiv \\
&\quad \forall x, y \in X : \ (\pi(x) \wedge \pi(y) \wedge x <_p y) \Rightarrow x <_M y \\
&\wedge \ \forall l \in L : \forall s \in S : \ \mathit{sees}\,(l, s) \Rightarrow [\ v_l = v_s \vee (\exists s' \in S : \mathit{sees}\,(l, s') \wedge s <_M s')\ ] \\
&\text{where} \quad \mathit{sees}\,(l, s) \equiv (\pi(l) \wedge \pi(s) \wedge (a_s = a_l) \wedge (s <_M l))
\end{aligned} \tag{3}$$

The second line of (3) expresses that the memory order may not contradict the program order, which is the essence of sequential consistency. The third line of (3) specifies that a load gets the last value "seen", that is, the last value stored to the same address. It uses the subformula $\mathit{sees}\,(l, s)$, which is defined on the last line of (3) and says that a load "sees" a store if and only if it succeeds it in the memory order M, goes to the same address, and both predicates are true.

The formula (3) still contains non-boolean variables and quantifiers. To obtain a CNF representation, we (a) encode non-boolean variables in A, B, or C as bitvectors, (b) expand quantifiers into finite conjunctions or disjunctions, and (c) break M down into boolean variables $\{M_{xy} \mid x, y \in X\}$ such that M_{xy} represents $x <_M y$ and add clauses to express transitivity, antisymmetry and non-reflexivity. The number of variables and clauses is then quadratic and cubic in $|X|$, respectively.

3.3 Encoding Correctness

We now show how to construct a formula Ψ such that (a) Ψ can be solved by a SAT solver, (b) unsatisfiability of Ψ implies correctness, and (c) given a satisfying assignment for Ψ, we can construct a counterexample trace. Such a trace shows a concurrent execution for which the serial reference execution is not observationally equivalent.

For a given test T and implementation I, a commit point specification h is understood as a function $\Pi_X \rightarrow \Pi_T$ that maps a given memory order m to the invocation order $h(m)$ that reflects how m orders the commit points. Now we can define

$$\begin{array}{rcl} \Psi_{T,I,Y,h} & \equiv & \exists A : \exists B : \exists C : \exists M : \quad \Phi_{T,I,Y}(A,B,C,M) \ \wedge \\ & & (\ g_{I,T}(h(M),A) \neq B \quad \vee \quad \neg\, \mathit{consistent}_T(h(M))\ ) \end{array} \tag{4}$$

To encode the subformula $g_{I,T}(h(M),A) \neq B$ in (4), we create a copy $T'(A,B')$ of $T(A,B)$ in which we put each invocation in a separate thread, and we define a special "memory model" *Atomic*, which is similar to sequential consistency but executes each thread atomically. Then $g_{I,T}(h(m),A) \neq B$ if and only if

$$\exists B' : \exists C' : \exists M' : \Phi_{T',I,\mathit{Atomic}}(A,B',C',M') \ \wedge \ h(M') = h(M) \ \wedge \ B \neq B' \tag{5}$$

After substituting (5) into (4), we can move all existential quantifiers to the front as required for SAT solving.

If the SAT solver determines that $\Psi_{T,I,Y,h}$ is unsatisfiable, it follows directly from the definitions that the implementation I is correct for the test T and memory model Y (regardless of h). However, if the SAT solver provides a satisfying assignment for $\Psi_{T,I,Y,h}$, our prototype presents the corresponding concurrent and serial executions to the user. The user can then analyze the counterexample and determine whether there is a defect in the implementation or a mistake in the commit point specification h.

3.4 Encoding Relaxed Memory Models

Relaxed memory models impose fewer ordering restrictions on the instruction streams than sequential consistency; therefore $R_{T,I,\mathit{SeqCons}} \subset R_{T,I,L}$ for all relaxed models L. Finding a uniform specification framework for the puzzling variety of memory models is a challenge of its own [25, 26]. For this case study, we restricted our attention to a selection of memory models (listed in the next paragraph) that are commonly used by hardware. Moreover, we are content with a *conservative approximation*, that is, a model *Relaxed* such that $R_{T,I,Y} \subset R_{T,I,\mathit{Relaxed}}$ for all memory models Y in our selection.

We compared the memory model specifications for the IBM PowerPC [19], Sun SPARC v9 TSO/PSO/RMO [20], Alpha [21], and IBM zArchitecture [22]. Although there are many differences, all of the specifications are based on a similar axiomatic style: they consist of a collection of rules that describe the valid instruction orderings and how values may flow from stores to loads. This non-operational style suits our purpose well; it allows us to compare the different models and derive a common approximation *Relaxed*, which we now describe in detail.

First, let us describe the relaxations with respect to sequential consistency informally. We use the symbols X, M, C, S, L, $\pi(x)$, a_x, v_x, and $<_p$ as defined in section 3.2.

- Accesses to different locations by the same thread may be executed out of order: If $x, y \in X$ and $x <_p y$ and $a_x \neq a_y$, we may have $y <_M x$.
- Loads to the same location by the same thread may be executed out of order: If $l, l' \in L$ and $l <_p l'$ and $a_l = a_{l'}$, we may have $l' <_M l$.

– Stores may be non-atomic: the stored value may be held in a thread-local buffer before becoming visible to other threads. We use $<_M$ to express the time at which a store commits globally, and we adjust the definition of $\mathit{sees}\,(l, s)$ to allow a load to see stores in the buffer. For example, if $s \in S$ and $l \in L$ and $s <_p l$ and $a_s = a_l$, we may have $l <_M s$ and $\mathit{sees}\,(l, s)$.

Our formalization is similar to the Sparc RMO memory model axioms [16]. In fact, our generic model is equivalent to the latter if we remove the RMO-specific axiom (m1) that defines how value and control dependencies influence the memory order.

If a memory ordering fence instruction appears in between two memory accesses in the code, they must execute in order. Fences affect only instructions in the same thread, and there exist specific variations (such as load-load, load-store, store-load or store-store fences) that target a subset of instructions only. Formally, let F to be the set of memory fences appearing in all instruction streams, and for each fence $f \in F$, let $X_f \subset X$ be the set of accesses affected by f. For example, if f is a store-load fence, then $X_f = \{s \in S \mid s <_p f\} \cup \{l \in L \mid f <_p l\}$.

Now we are ready to define *Relaxed* formally. We do so by directly specifying

$$\begin{aligned}
&\Theta_{T,I,\mathit{Relaxed}}(M, C) \quad \equiv \\
&\quad \forall x \in X : \forall s \in S : \ (\pi(x) \wedge \pi(s) \wedge a_x = a_s \wedge x <_p s) \Rightarrow x <_M s \\
&\wedge\ \forall l \in L : \forall s \in S : \ \mathit{sees}\,(l, s) \ \Rightarrow\ v_l = v_s \vee (\exists s' \in S : \mathit{sees}\,(l, s') \wedge s <_M s') \\
&\wedge\ \forall f \in F : \forall x, y \in X_f : \ (\pi(f) \wedge \pi(x) \wedge \pi(y) \wedge (x <_p f <_p y)) \Rightarrow x <_M y \\
&\text{where} \quad \mathit{sees}\,(l, s) \equiv \pi(l) \wedge \pi(s) \wedge (a_s = a_l) \wedge (s <_M l \vee s <_p l)
\end{aligned} \tag{6}$$

The second line of (6) specifies the conditions under which the memory order may not contradict the program order. When compared with the formula (3) for sequential consistency, we see that this line has been weakened to reflect the ordering relaxations we described earlier. The third line specifies that a load gets the last value "seen", that is, the last value stored to the same address. It is the same as for sequential consistency (3), but the definition of $\mathit{sees}\,(l, s)$ on the last line has been modified to allow forwarding. The fourth line of (6) defines the effect of memory fences on the valid memory orderings.

The memory model *Relaxed* is simpler than most memory models used for actual hardware because (a) it consistently relaxes the order, for example, even data- or control-dependent instructions may be reordered, and no special measures are taken to prevent circular value flow, (b) it uses a single, generic memory ordering fence construct, (c) it does not contain specific synchronization primitives, but allows them to be expressed as atomic blocks (we omitted atomic blocks from the formalization above, but they can introduced easily by adding suitable constraints on $<_M$), and (d) it omits unneeded details such as the behavior of instruction caches and I/O, special flushing operations, or unaligned and non-atomic memory accesses.

This (relative) simplicity makes *Relaxed* a good model for studying the algorithms: even though it may exhibit executions that are not possible on a specific target architecture, we are made aware of all issues by verifying our code on *Relaxed*. Once we understand which instructions need to stay in order, it is comparatively easy to pick the right fences for a specific target architecture.

Specialized algorithms to insert memory fences automatically during compilation have been proposed [27, 28]. However, these methods are based on a conservative program analysis, and they enforce sequential consistency on the instruction level rather than the operation level. These characteristics make them unattractive for optimized implementations, because redundant fences imply suboptimal performance [29].

4 Results

We implemented a prototype that encodes SAT instances as described in the previous chapter, solves them using zChaff [30], and converts satisfying assignments into human-readable execution traces. We first tested our prototype on some smaller examples (including the spinlock [16]). Then we hand-translated the pseudo-code (Fig. 1) into the intermediate language accepted by our back-end prototype. Next, we created a suite of symbolic tests (Fig. 4) and made an initial guess at the commit points (lines 15 and 31 in Fig. 1).

Running our prototype, we found five problems (numbered 1–5 below). First, we ran T0 on a sequentially consistent memory model, finding problem 1. Then, we ran T0 on our relaxed memory model, finding problems 2–4. Next, we ran on T1 on the relaxed model and found problem 5. After that, no more problems were found. The tests T0 and T1 alone (neither of which took more than a few seconds) therefore uncovered all the bugs found.

1. **Incorrect commit point specification.** We had guessed line 31 to be the commit point. The tool produced a counterexample revealing a race between the store on line 15 and the load on line 32. The outcome of this race determines the logical order of the operations, so we changed the commit point for the dequeue to be line 32 instead of line 31.
2. **Incorrect modelling of dynamic memory.** Our initial model for dynamic memory allocation was incorrect for relaxed memory models: the trace showed a load from a storage location inside a dynamically allocated block that took effect only after the block was freed, re-allocated by another thread, and then overwritten. This situation caused the load to get the wrong value. We fixed this problem by inserting fences into the alloc() and free() calls.

Program name	T0	T1	T5-3	T5-4	T5-5	T5-6	Tpc4	Tpc6
Thread 1 sequence	e	e	e e e e	e e e	e e	e	e e e e	e e e e e e
Thread 2 sequence	d	e	d	e	e	e	d d d d	d d d d d d
Thread 3 sequence		d	d	d	e	e		
Thread 4 sequence		d		d	d	e		
Thread 5 sequence					d	d		
Thread 6 sequence						d		

Fig. 4. A selection of the symbolic tests we used. The letters e and d represent calls to the enqueue and dequeue operation (with symbolic arguments). All calls operate on the same queue object.

	Program Characteristics					SAT encoding		Requirements	
	threads	operations	instructions	loads	stores	variables	clauses	memory [kB]	time [s]
T0	2	2	65	12	18	551	4,081	332	0.004
T1	4	4	119	23	30	1,514	44,479	4,165	0.87
T5-3	3	6	163	31	44	3,380	160,516	16,246	9.33
T5-4	4	6	163	31	44	3,400	167,456	16,308	21.1
T5-5	5	6	163	31	44	3,413	173,324	16,357	35.4
T5-6	6	6	163	31	44	3,419	179,109	16,401	42.8
Tpc2	2	4	119	23	30	1,504	42,829	4,151	0.139
Tpc3	2	6	173	34	42	3,717	170,116	16,320	5.23
Tpc4	2	8	227	45	54	5,797	430,445	33,372	45.7
Tpc5	2	10	281	56	66	8,315	877,624	100,462	300.0
Tpc6	2	12	335	67	78	11,271	1,549,090	131,087	886.3
Tpc7	2	14	389	78	90	12,394	2,438,721	n/a	> 1000

Fig. 5. Some experimental data. All resource requirements are reported by the zChaff solver (version 2004/11/15) and refer to unsatisfiable instances using a relaxed memory model. The tests were run on a 3 GHz Pentium 4 desktop Linux PC.

3. **Missing store-store fence.** On a relaxed model, the store instruction that updates the queued value (line 12) may be ordered after the load that is supposed to read it (line 38). To force the store to take effect by the time the node is linked into the list, we insert a store-store fence before the store on line 15.
4. **Missing load-load fence.** Symmetrically, we need to make sure that the load of the queued value (line 38) does not take effect before the load of its address on line 32. This may seem automatic — but some weak memory models (such as Alpha [21]) do not enforce in-order execution of loads, even if there is a value dependency [31]. Therefore, we insert a load-load memory fence after the load on line 32.
5. **Incorrect modelling of locks.** During the translation, we had misplaced one of the fences within the code for `unlock()`. It appeared after instead of before the committing store, where it is useless. Without proper fences in `lock()` and `unlock()`, memory accesses can "escape" from the critical section.

Analysis. The results indicate that our method is efficient at finding errors in highly concurrent programs, but does not scale to long program executions. As expected, zChaff was much quicker at solving satisfiable instances than at refuting unsatisfiable ones, but the choice of the memory model seemed to have a negligible effect on the runtime. We show some statistics about the programs and the resources required (for unsatisfiable instances and the relaxed memory model) in Fig. 5. The results show that making the programs longer (Tpc series, see Fig. 4 for definition) is more challenging for the solver than making them more concurrent (T5 series). This result is not surprising because we chose an encoding that specializes on highly concurrent executions.

5 Conclusions

Verifying the sequential consistency of a concurrent data type implementation on a relaxed memory model presents a challenge because of the high degree of concurrency

at the instruction level and the infinite state space. In this case study, we developed a new SAT-based method that can solve a bounded formulation of this problem (using finite symbolic tests and commit point annotations) and demonstrated its practical value by applying it successfully to Michael and Scott's two-lock queue implementation.

Future work includes exploring more example data structure implementations, eliminating the need for commit point specifications, automating the creation of a symbolic test suite, improving the scalability with more efficient or incremental SAT encodings, and developing a front end for the tool that would accept a subset of C as the specification of the implementation.

References

[1] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[2] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[3] G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

[4] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[5] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[6] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

[7] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: verifying concurrent programs by runtime refinement-violation detection. In *Programming Language Design and Implementation (PLDI)*, pages 27–37, 2005.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2988, pages 168–176. Springer, 2004.

[9] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory-model-sensitive data race analysis. In *International Conference on Formal Engineering Methods (ICFEM)*, LNCS 3308, pages 30–45. Springer, 2004.

[10] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Computer-Aided Verification (CAV)*, LNCS 3114, pages 401–413, 2004.

[11] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[12] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 129–136, 2006.

[13] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Computer-Aided Verification (CAV)*, LNCS 3576, pages 82–97. Springer, 2005.

[14] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 34–41, 1995.

[17] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[18] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comp. Sys.*, 15(4):391–411, 1997.

[19] B. Frey. *PowerPC Architecture Book v2.02*. International Business Machines Corporation, 2005.

[20] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

[21] Compaq Computer Corporation. *Alpha Architecture Reference Manual*, 4th edition, January 2002.

[22] International Business Machines Corporation. *z/Architecture Principles of Operation*, first edition, December 2000.

[23] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Correct Hardware Design and Verification Methods (CHARME)*, LNCS 2860, pages 81–95. Springer, 2003.

[24] R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *Logic in Computer Science (LICS)*, pages 219–228, 1996.

[25] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

[26] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[27] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[28] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *International Conference on Supercomputing (ICS)*, pages 285–294, 2003.

[29] C. von Praun, T. Cain, J. Choi, and K. Ryu. Conditional memory ordering. In *International Symposium on Computer Architecture (ISCA)*, 2006.

[30] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535, 2001.

[31] M. Martin, D. Sorin, H. Cain, M. Hill, and M. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *International Symposium on Microarchitecture (MICRO)*, pages 328–337, 2001.