CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models *

Sebastian Burckhardt
University of Pennsylvania
sburckha@cis.upenn.edu

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

Milo M. K. Martin
University of Pennsylvania
milom@cis.upenn.edu

Abstract

Concurrency libraries can facilitate the development of multi-threaded programs by providing concurrent implementations of familiar data types such as queues or sets. There exist many optimized algorithms that can achieve superior performance on multiprocessors by allowing concurrent data accesses without using locks. Unfortunately, such algorithms can harbor subtle concurrency bugs. Moreover, they require memory ordering fences to function correctly on relaxed memory models.

To address these difficulties, we propose a verification approach that can exhaustively check all concurrent executions of a given test program on a relaxed memory model and can verify that they are observationally equivalent to a sequential execution. Our *CheckFence* prototype automatically translates the C implementation code and the test program into a SAT formula, hands the latter to a standard SAT solver, and constructs counterexample traces if there exist incorrect executions. Applying *CheckFence* to five previously published algorithms, we were able to (1) find several bugs (some not previously known), and (2) determine how to place memory ordering fences for relaxed memory models.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — Formal Methods, Model Checking; D.1.3 [*Programming Techniques*]: Concurrent Programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — Mechanical Verification

General Terms Verification

Keywords Concurrent Data Structures, Multi-Threading, Shared-Memory Multiprocessors, Memory Models, Lock-Free Synchronization, Sequential Consistency, Software Model Checking

1. Introduction

Shared-memory multiprocessors and multi-core chips are now ubiquitous. Nevertheless, programming such systems remains a challenge [44]. Concurrency libraries such as the java.util.concurrent package JSR-166 [27] or the Intel Threading Building Blocks [22] support the development of safe and efficient multi-threaded programs by providing *concurrent data types*, that is, concurrent implementations of familiar data abstractions such as queues, sets, or maps.

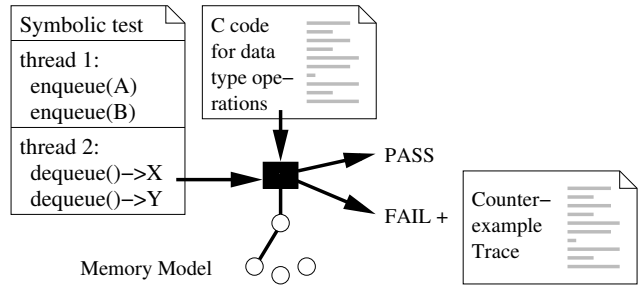

Figure 1. Black-box view of the *CheckFence* tool.

Client programs with threads that execute concurrently on a multiprocessor can benefit from implementations that are optimized for concurrency. Many sophisticated algorithms that use lock-free synchronization have been proposed for this purpose [17, 18, 31, 33, 34, 43]. Such implementations are not race-free in the classic sense because they allow concurrent access to shared memory locations without using locks for mutual exclusion.

Algorithms with lock-free synchronization are notoriously hard to verify, as witnessed by many formal verification efforts [6, 14, 46, 50] and by bugs found in published algorithms [10, 32]. Many more interleavings need to be considered than for implementations that follow a strict locking discipline. Moreover, the deliberate use of races prohibits the use of classic race-avoiding design methodologies and race-detecting tools [1, 12, 19, 24, 36, 39, 41, 49].

To make matters worse, most commonly used multiprocessor architectures use *relaxed memory ordering models* [2]. For example, a processor may reorder loads and stores by the same thread if they target different addresses, or it may buffer stores in a local queue. Whereas fully lock-based programs are insensitive to the memory model (because the lock and unlock operations are designed to guarantee the necessary memory ordering), implementations that use lock-free synchronization require explicit *memory ordering fences* to function correctly on relaxed memory models. Fences counteract the ordering relaxations by selectively enforcing memory order between preceding and succeeding instructions. A lack of fences leads to incorrect behavior, whereas an overzealous use of fences impacts performance. Nevertheless, fence placements are rarely published along with the algorithm.

To help designers and implementors develop correct and efficient programs for relaxed models, we present a method that can statically check the consistency of a data type implementation for a

* This research was partially supported by NSF awards CPA 0541149 and CSR-EHS 0509143 and by donations from Intel Corporation.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

given bounded test program and memory model (Fig. 1). Given the implementation code, a small test program representing the client program, and a choice of memory model, our *CheckFence* prototype verifies for all concurrent executions of the test that the observed results are consistent with the expected semantics of the data type. If the check fails, a counterexample trace is presented to the user who can then analyze and fix the problem.

We build upon and further automate the general technique described in our prior case study [4]. At the heart of this method is an encoding that represents the executions of the test program as solutions to a propositional formula. To obtain this formula, we first compile the code for each thread into a bounded sequence of instructions. Then we separately encode the thread-local program semantics (in the style of CBMC [5]) and the memory model (in axiomatic form). Once the encoding is complete, we can use a standard SAT solver to solve for erroneous executions.

Our method proceeds in two steps. First, we perform a *specification mining*: we automatically create a specification for the given test by enumerating the set of correct observations. An *observation* is a combination of argument and return values, and it is *correct* if it is consistent with some atomic interleaving of the operations. For example, for the test in Fig. 1, $(A = 0, B = 1, X = 0, Y = 1)$ is a correct observation, whereas $(A = 0, B = 1, X = 0, Y = 0)$ is not.

After mining the specification, we then check all executions of the test on the chosen memory model to see that the observed values are contained in the specification. This step is called *inclusion check*. If the inclusion check fails, we produce a counterexample trace. The trace presents the details of the execution to the user, who can then analyze and fix the problem.

We implemented this method in a prototype called *CheckFence* and applied it to five previously published algorithms, writing test programs and C code that closely follows the published pseudocode. We were thus able to

1. Reproduce two known bugs in a concurrent deque algorithm known as "snark" [8, 10, 26].
2. Uncover a not-previously-known bug in a lazy list-based set implementation. The published pseudocode [18] fails to initialize a field, which was missed by a prior formal verification using the PVS interactive proof system [6].
3. Show numerous failures on architectures with relaxed memory models (the original algorithms assume sequentially consistent architectures), and fix them by inserting appropriate memory ordering fences.

2. Problem Formulation

In this section, we describe the parameters of the verification problem (test programs, correctness condition, and memory models) more concretely.

2.1 Test Programs

To exercise the implementation code, we use *test programs*. A test program specifies a finite sequence of operation calls for each thread. It may choose to leave the argument values unspecified, which conveniently allows us to cover many scenarios with a single test. It may also specify initialization code to be performed prior to the concurrent execution of the threads.

2.2 Correctness Condition

The basic idea of our method is to compare the set of concurrent executions with the set of serial executions. Given a test program T that makes invocations to the operations of some abstract data type, and an implementation I, we define

- $E_{T,I,\mathit{Serial}}$ is the set of serial executions. Serial executions are executions by a single processor that interleaves the threads and treats the operations as atomic, that is, does not switch threads within operations.
- $E_{T,I,Y}$ is the set of multiprocessor executions for memory model Y. The model Y may use relaxed memory ordering rules.

We define these sets more formally below in Section 2.3.1 and proceed first to a description of how they relate to our correctness condition. For a given execution e we define the *observation vector* $\mathit{obs}(e)$ to consist of the argument and return values to the operations that occur in e. For test program T and implementation I, we define the *observation set* as

$$S_{T,I} = \{\mathit{obs}(e) \mid e \in E_{T,I,\mathit{Serial}}\}$$

The observation set S captures the intended behavior of the data type, and serves as a specification in the following sense. For test T and observation set S, we say that the implementation I *satisfies* S *on memory model* Y if and only if

$$\forall e \in E_{T,I,Y} : \mathit{obs}(e) \in S$$

Because the argument and return values are all that the client program observes, implementations that satisfy the specification are guaranteed to appear to the client program as if they executed the operations atomically.

Note that we need not necessarily use the same implementation when extracting the specification S and when performing the inclusion check. In practice, it is often sensible to write a reference implementation (which need not be concurrent and is thus simple) to construct the observation set.

2.3 Memory Models

We currently support two hardware-level memory models. One is the classic *sequential consistency* [25], which requires that the loads and stores issued by the individual threads are interleaved in some global, total order. Sequential consistency is easiest to understand; however, it is not guaranteed by most multiprocessors [2].

The other model is *Relaxed* [4]. It allows the hardware to relax the ordering and atomicity of memory accesses. Specifically, it permits (1) reorderings of loads and stores to different addresses, (2) buffering of stores in local queues, (3) forwarding of buffered stores to local loads, (4) reordering of loads to the same address, and (5) reordering of control- or data-dependent instructions.

2.3.1 Set of Executions

For a test T with n threads, an implementation I, and a memory model Y, we define the set of executions $E_{T,I,Y}$ to consist of all execution traces $e = (w_1, \ldots, w_n)$ such that (1) each w_i is a finite sequence of basic machine instructions (loads, stores, assignments, and fences) within which all instructions are annotated with the execution values, (2) each instruction sequence w_i corresponds to a sequential execution (under standard semantics) of the code for thread i as specified by T, I, and (3) the trace e satisfies the conditions of the memory model Y as defined in the next section.

2.3.2 Memory Model Axioms

We now present the core of our axiomatic formulations for sequential consistency and *Relaxed*. First, we need some common notation. Let A be a set of addresses, and V be a set of values. For a given execution trace $e = (w_1, \ldots, w_n)$, let $X_e = L_e \cup S_e$ be the set of memory accesses in the trace, with L_e being the set of loads and S_e being the set of stores. For an access $x \in X_e$, let $a(x) \in A$ be the address of the accessed location, and $v(x) \in V$ be the value

loaded or stored. For an address $a \in A$, let $i(a) \in V$ be the initial value of the memory location a. Let $<_p$ be the program order, that is, a partial order on X_e such that $x <_p y$ whenever x precedes y within some sequence w_i.

Sequential Consistency. An execution trace e is sequentially consistent if there exists a total order $<_M$ over X_e (the memory order) subject to the following conditions. For each load $l \in L_e$, let $S(l)$ be the set of stores that are "visible" to l:

$$S(l) = \{s \in S_e \mid a(s) = a(l) \wedge (s <_M l)\}$$

Then the following axioms must be satisfied:

1. if $x <_p y$, then $x <_M y$
2. if $l \in L_e$ and $S(l) = \emptyset$, then $v(l) = i(a)$
3. if $l \in L_e$ and $s \in S(l)$ and $v(l) \neq v(s)$, then there exists a store $s' \in S(l)$ such that $s <_M s'$.

Relaxed. An execution trace e is allowed by the memory model *Relaxed* if there exists a total order $<_M$ over X_e (the memory order) subject to the following conditions. For each load $l \in L_e$, let $S(l)$ be the set of stores that are "visible" to l:

$$S(l) = \{s \in S_e \mid a(s) = a(l) \wedge ((s <_M l) \vee (s <_p l))\}$$

Then the following axioms must be satisfied:

1. if $x <_p y$, $a(x) = a(y)$, and $y \in S_e$, then $x <_M y$
2. if $l \in L_e$ and $S(l) = \emptyset$, then $v(l) = i(a)$
3. if $l \in L_e$ and $s \in S(l)$ and $v(l) \neq v(s)$, then there exists a store $s' \in S(l)$ such that $s <_M s'$.

The model *Relaxed* differs from sequential consistency in two places. For one, axiom 1 has been weakened to allow the memory order to be different from the program order. Secondly, the set $S(l)$ has been modified to allow forwarding of values from stores sitting in a local store queue to subsequent loads by the same processor: $S(l)$ may contain stores that precede a load in program order ($s <_p l$) but are performed globally only after the load is performed ($l <_M s$).

Seriality. We can conveniently formalize serial executions (executions that interleave the operations atomically) by defining seriality as a special kind of "memory model" as follows. Given T, I and an execution trace e, define an equivalence relation $\sim$ over X_e such that $x \sim y$ whenever x, y are part of the same operation. Then, we say e is serial if and only if (1) it is sequentially consistent, and (2) if $x \sim x'$ and $y \sim y'$, then $(x <_M y) \Leftrightarrow (x' <_M y')$.

2.3.3 Comparison to Other Memory Models

Memory models can be compared in terms of the execution traces they allow. We call a model Y *stronger* than another model Y' if every execution trace that is allowed by model Y is also allowed by Y'. For example, seriality is stronger than sequential consistency, and sequential consistency is stronger than *Relaxed*.

The purpose of *Relaxed* is to provide a common, conservative approximation of several memory models (Sun SPARC v9 TSO/PSO/RMO [48], Alpha [7], and IBM 370/390/zArchitecture [23]). All of these models are stronger than *Relaxed*, which implies that once code runs correctly on *Relaxed*, it will run correctly on the former.

However, *Relaxed* is not strictly weaker than the official PowerPC [13], IA-64 [21] and IA-32 [20] models because it globally orders all stores (the execution in Fig. 2 illustrates this point). Even so, *Relaxed* still captures the most important relaxations of those models and is useful to determine where to place fences. This limitation is not fundamental to our methodology, and we are actively working on formalizing a weaker version of *Relaxed* to close the gap.

Initially, $x = y = 0$			
thread 1	thread 2	thread 3	thread 4
store x, 1	store y, 1	load x, 1 load-load fence load y, 0	load y, 1 load-load fence load x, 0

Figure 2. An execution trace that is not possible on *Relaxed*, but not ruled out on PPC, IA-32, and IA-64. On the latter, we represent the load-load fence as follows: (PPC) lwsync, (IA-32) lfence, (IA-64) replace load that precedes the fence with load-acquire.

3. Solution

We now describe how we implemented and applied our method. See Fig. 3 for a schematic view of the internal structure of the tool.

3.1 The Front-End

CheckFence has a front-end that compiles the C code into an intermediate representation. This intermediate representation uses a custom language called *load-store language* (LSL) which precisely defines the possible instruction sequences of stores, loads, fences, and synchronization instructions for each thread.

The front-end is based on the CIL framework [37] which parses C and provides us with a cleaned-up and somewhat simplified abstract syntax tree. From there, compilation into LSL is relatively straightforward for most programs. *CheckFence* translates concurrent data type implementations of realistic detail precisely and automatically, but it may refute some programs if they contain unsupported features. We discuss some of the choices we made in the following paragraphs. See Fig. 4 for the abstract syntax of LSL.

C multiprocessor semantics. The C language does not specify a memory model (standardization efforts for C/C++ are still under way). Therefore, executing memory-model sensitive C code on a multiprocessor can have unpredictable effects [3]. On the machine language level, however, the memory model is officially defined by the hardware architecture. It is therefore possible to write C code for relaxed models by exerting direct control over the C compilation process to prevent optimizations that would alter the program semantics. The details of how to do this (for example, volatile declarations, compiler pragmas, or command line options) are compiler-dependent and beyond the scope of this work. Here, we simply assume a "vanilla" compilation without optimizations, and we apply the hardware-level memory model to the resulting machine-level program.

Values and types. We found that the types present at C source level can not be relied upon (due to the presence of casts). Therefore, we chose to keep LSL untyped; however, we do track the type

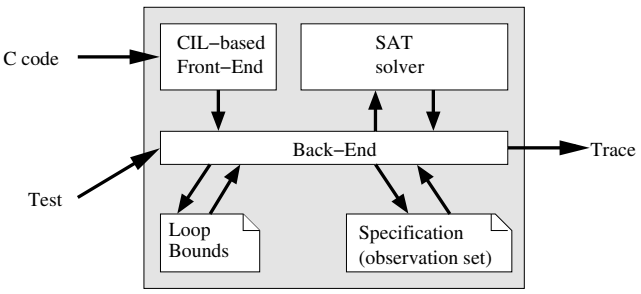

Figure 3. Schematic view of the components.

(number)	$n \in \mathbb{N}$
(value)	v ::= undefined $\mid$ n $\mid$ [$\overline{n}$]
(register)	r
(primitive op)	f
(procedure name)	p
(block tag)	t
(statement)	s ::=
(constant)	$r = v$
(primitive op)	$\mid r = f(\overline{r})$
(store)	$\mid *r = r$
(load)	$\mid r = *r$
(fence X)	$\mid$ fence$_X$
(atomic block)	$\mid$ atomic { $\overline{s}$ }
(procedure call)	$\mid p(\overline{r})(\overline{r})$
(labeled block)	$\mid t$: { $\overline{s}$ }
(leave block)	$\mid$ if (r) break t
(repeat block)	$\mid$ if (r) continue t
(assertion)	$\mid$ assert (r)
(assumption)	$\mid$ assume (r)

Figure 4. The abstract syntax of LSL

```
struct {
  long a;
  int b[3];
} x;
int y;
```

Pointer	C value	LSL value
&(x)	0x000	[0]
&(x.a)	0x000	[0 0]
&(x.b)	0x008	[0 1]
&(x.b[0])	0x008	[0 1 0]
&(x.b[1])	0x00C	[0 1 1]
&(x.b[2])	0x010	[0 1 2]
&(y)	0x014	[1]

Figure 5. Representing C pointers in LSL.

of values at runtime, by distinguishing between undefined, integer, and pointer values. The back-end also recovers some static type information directly from the untyped code by performing a range analysis (see Section 3.4). The runtime types help to automatically detect bugs. For example, we detect if a program uses an undefined value in a computation or a condition.

Pointer values. We represent pointer values as a sequence of natural numbers $[n_1 \ldots n_k]$ where $(k \geq 1)$, representing the base address n_1 and sequence of offsets $n_2, \ldots, n_k$. The offsets may be field or array offsets, providing a unified way of handling arrays and structs. See Fig. 5 for an example of how C pointers can be represented in this manner. The advantage of keeping the offsets separate from the base address is that we can avoid addition or multiplication when encoding pointer operations in the back-end. Moreover, our range analysis (Section 3.4) can often determine that large portions of the sequence are statically fixed.

Control flow. To facilitate a minimalistic unrolling of loops in the back end, we retain the nested block structure of the source program. Conditionals are represented by conditional breaks and continues which can exit or repeat an enclosing block identified by its tag.

Fences. Fences are special machine instructions that guarantee some ordering among the memory accesses that precede and follow it. We currently support four kinds of memory ordering fences: load-load, load-store, store-load and store-store (as used by the Sparc RMO memory model [48]). An X-Y fence guarantees that all accesses of type X that appear before the fence will be ordered before all accesses of type Y that appear after the fence. Fences can guarantee some ordering among the memory accesses without enforcing full sequential consistency.

```
bool cas(unsigned *loc,
         unsigned old, unsigned new) {
  atomic {
    if (*loc == old) {
      *loc = new;
      return true;
    } else {
      return false;
    }
  }
}
```

Figure 6. Pseudocode for the compare-and-swap (CAS) operation.

```
typedef enum { free, held } lock_t;

void lock(lock_t *lock) {
  lock_t val;
  do {
    atomic {
      val = *lock;
      *lock = held;
    }
  } while (val != free);
  fence("load-load");
  fence("load-store");
}

void unlock(lock_t *lock) {
  fence("load-store");
  fence("store-store");
  atomic {
    assert(*lock == held);
    *lock = free;
  }
}
```

Figure 7. Pseudocode for the lock and unlock operations.

Synchronization. We currently model all synchronization in LSL using atomic blocks. The instructions within an atomic block are guaranteed to execute in program order, and they are never interleaved with instructions in other threads. See Fig. 6 for a pseudocode example of how we model the compare-and-swap instructions using an atomic block. Our lock and unlock operations are based on code from the SPARC v9 architecture manual [48] and use a spin loop, an atomic load-store primitive, and partial memory ordering fences (Fig. 7). To avoid an unbounded unrolling of the spin loop, we use a custom reduction for side-effect free spin loops.

C features. The C language has many features, not all of which are supported by our *CheckFence* prototype. We are adding features as needed to handle the implementations we wish to study. Already supported are pointers, structs, arrays, full integer arithmetic, limited pointer arithmetic, nested loops, limited gotos, and packed structures.[1]

3.2 The Back-End

The back-end first transforms the test program T and implementation I by inlining the operation calls and unrolling the loops (more on this in Section 3.3). As a result, the code for each thread is a simple sequence of machine-level instructions comprising only loads, stores, register assignments, fences, and forward branches. We now encode the possible executions as a propositional formula $\Phi_{T,I,Y}(Z)$ over boolean variables Z such that each solution of Φ corresponds to an execution $e \in E_{T,I,Y}$ (more on this in Sec-

[1] Many implementations pack structures into a single machine word with the intent of accessing the entire structure atomically (for example, using a compare-and-swap operation).

tion 3.2.1). Once we have Φ thus encoded, we can perform the specification mining and inclusion check using a standard SAT solver as follows.

Specification mining. To construct the observation set $S_{T,I}$ we use the following iterative procedure. First, we provide the formula $\Phi_{T,I,\mathit{Serial}}(Z)$ as an input to the SAT solver. Next, we run the solver which will return a solution for Z, corresponding to some serial execution $e \in E_{T,I,\mathit{Serial}}$. Let o_1 be the observation of this execution. Now, we add additional constraints to the solver to exclude executions that have the observation o_1 and run the solver again. If there is another solution, it gives us a new observation o_2. By continuing this process (adding constraints to rule out observations we already saw) until the SAT solver determines insatisfiability (say, after k steps), we obtain the observation set $S_{T,I} = \{o_1, o_2, \ldots, o_k\}$.

Our practical experience suggests that although the set of serial executions $E_{T,I,\mathit{Serial}}$ can be quite large (due to nondeterministic memory layout and interleavings), the observation set $S_{T,I}$ contains no more than a few thousand elements for the testcases we used. Therefore, the iterative procedure described above is sufficiently fast, especially when used with a SAT solver that supports incremental solving.

Inclusion check. For a given test T, implementation I, memory model Y, and finite observation set S, we check the inclusion $\mathit{obs}(E_{T,I,Y}) \subset S$ by asking the SAT solver to find a solution for the variables Z subject to the constraints

$$\Phi_{T,I,Y}(Z) \ \wedge \bigwedge_{o \in S} \mathit{obs}(Z) \neq o$$

If the SAT solver finds a satisfying assignment, then the corresponding execution is a counterexample because its observation is not equal to any observation in S. On the other hand, if the SAT instance is unsatisfiable, the inclusion check passes.

3.2.1 Encoding Concurrent Executions

After inlining I in T and unrolling the loops, the code resembles a machine-level program consisting only of loads, stores, register assignments, fences, and forward branches. Following the definition of $E_{T,I,Y}$ in Section 2.3.1, we can now encode the possible execution traces $e = (w_1, \ldots, w_n)$ by introducing variables to represent the execution values and writing constraints over these variables to capture the conditions (2) and (3). To encode the thread-local semantics (condition 2), we use a formula $\Delta_{T,I,k}$ for each thread k. To encode the memory model (condition 3), we use a formula $\Theta_{T,I,Y}$.

The thread-local formulae. To obtain $\Delta_{T,I,k}$, we follow a technique similar to the CBMC tool [5]; specifically, we use a register SSA (single static assignment) form that guarantees that for each program point and for each register there is statically known, unique instruction that assigned it last.

For each thread k, we introduce a set of variables V_k containing one variable for each instruction, representing the LSL value produced by that instruction. Next, we introduce a set C_k of boolean variables containing one variable for each forward branch, representing whether the branch is taken or not. We now create constraints for each assignment to express the relationship between the consumed and produced values, and for each branch to express how the branch condition depends on the value of some register. By taking the conjunction of all these constraints we get a formula $\Delta_{T,I,k}(V_k, C_k)$ that captures the possible executions of the thread in an unspecified environment (that is, for unspecified values returned by the loads).

The memory model formula. We construct a formula $\Theta_{T,I,Y}$ to represent the memory model. In our case, Θ is simply the conjunction of the memory model axioms for Y (Section 2.3.2). If we represent the memory order $<_M$ by a variable M (ranging over all total orders of X), we thus get a formula $\Theta_{T,I,Y}(M, V, C)$ where $V = \bigcup_k V_k$ and $C = \bigcup_k C_k$. This formula depends on the variables in V because the axioms make reference to the addresses and values used by instructions. It depends on the variables in C because the axioms apply to executed memory accesses only (an access that is skipped over by a branch is not part of the set X_e).

The combined formula. We combine the thread-local and communication formulae as follows

$$\Phi_{T,I,Y}(M, V, C) \ \equiv \ \Theta_{T,I,Y}(M, V, C) \ \wedge \bigwedge_k \Delta_{T,I,k}(V_k, C_k)$$

Finally, we need to transform Φ down to the level of the SAT solver, which requires conjunctive normal form $\Phi_{T,I,Y}(Z)$ for some set of boolean variables Z. We thus need to replace all quantifiers by finite conjunctions or disjunctions and break M and V down to boolean variables. During this process we introduce auxiliary variables, as follows.

1. To encode the memory order M, we introduce auxiliary variables $\{M_{xy} \mid x, y \in X\}$ such that M_{xy} represents $x <_M y$. To express antisymmetry, we represent M_{xy} and M_{yx} by the same SAT variable (adjusting the sign of literals). To express transitivity, we add explicit clauses.
2. To encode the value variables V, we use bitvectors. To get a conservative estimate on the required width, we perform a range analysis (Section 3.4).
3. For each pair of values $v_1, v_2 \in V$, we introduce auxiliary variables to represent the equalities $v_1 = v_2$. We use separate clauses to break the equalities down to the bit level (which we need only do for equality literals that appear in the formula).
4. We use for each $l \in L$ an auxiliary variable Init_l that represents whether $S(l) = \emptyset$, and for each $s \in S$ and $l \in L$ an auxiliary variable Flow_{sl} that represents whether s is the maximal store in $S(l)$.

The resulting CNF encoding is polynomial: the number of SAT variables and clauses is quadratic and cubic in the size of the unrolled test program, respectively.

3.3 Loop Bounds

For the implementations and tests we studied, all loops are statically bounded. However, this bound is not necessarily known in advance. We therefore unroll loops *lazily* as follows. For the first run, we unroll each loop exactly once. We then run our regular checking, but restrict it to executions that stay within the bounds. If an error is found, a counterexample is produced (the loop bounds are irrelevant in that case). If no error is found, we run our tool again, solving specifically for executions that exceed the loop bounds. If none is found, we know the bounds to be sufficient. If one is found, we increment the bounds for the affected loop instances and repeat the procedure.

3.4 Range Analysis

To reduce the number of boolean variables, we perform a range analysis before encoding Φ. Specifically, we use a simple lightweight flow-insensitive analysis to calculate for each SSA register r and each memory location m sets S_r, S_m that conservatively approximate the values that r or m may contain during a valid execution. We can sketch the basic idea as follows. First, initialize S_r and S_m to be the empty set. Then, keep propagating values as follows until a fixpoint is reached:

ms2	Two-lock queue [33]	Queue is represented as a linked list, with two independent locks for the head and tail.
msn	Nonblocking queue [33]	Similar, but uses compare-and-swap for synchronization instead of locks (Fig. 9).
lazylist	Lazy list-based set [6, 18]	Set is represented as a sorted linked list. Per-node locks are used during insertion and deletion, but the list supports a lock-free membership test.
harris	Nonblocking set [16]	Set is represented as a sorted linked list. Compare-and-swap is used instead of locks.
snark	Nonblocking deque [8, 10]	Deque is represented as linked list. Uses double-compare-and-swap.

Table 1. The implementations we studied. We use the mnemonics on the left for quick reference.

- constant assignments of the form $r = c$ propagate the value c to the set S_r.
- assignments of the form $r = f(r_1, \ldots, r_k)$ propagate values from the sets $S_{r_1}, \ldots, S_{r_k}$ to the set S_r (applying the function).
- stores of the form $*r' = r$ propagate values from the set S_r to the sets $\{S_m \mid m \in S_{r'}\}$.
- loads of the form $r = *r'$ propagate values from the sets $\{S_m \mid m \in S_{r'}\}$ to the set S_r.

This analysis is sound for executions that do not have circular value dependencies. To ensure termination, we need an additional mechanism. First, we count the number of assignments in the test that have unbounded range. That number is finite because we are operating on the unrolled, finite test program. During the propagation of values, we tag each value with the number of such functions it has traversed. If that number ever exceeds the total number of such functions in the test, we can discard the value.

We use the sets S_r for four purposes: (1) to determine a bitwidth that is sufficient to encode all integer values that can possibly occur in an execution, (2) to determine a maximal depth of pointers, (3) to fix individual bits of the bitvector representation (such as leading zeros), and (4) to rule out as many aliasing relationships as possible, thus reducing the size of the memory model formula.

4. Results

We studied the five implementations shown in Table 1. All of them make deliberate use of data races. Although the original publications contain detailed pseudocode, they do not indicate where to place memory ordering fences. Thus, we set out to (1) verify whether the algorithm functions correctly on a sequentially consistent memory model, (2) find out what breaks on the relaxed model and (3) add memory fences to the code as required.

First we wrote symbolic tests (Fig. 8). To keep the counterexamples small, we started with small and simple tests, say, two to four threads with one operation each. All memory model-related bugs were found on such small testcases. We then gradually added larger tests until we reached the limits of the tool. Fig. 10 shows the tests we ran for each implementation.

4.1 Bugs Found

We found several bugs that are not related to relaxations in the memory model. The snark algorithm has two known bugs [10, 26]. We found the first one quickly on test D0. The other one requires a fairly deep execution. We found it with the test Dq, which took about an hour.

We also found a not-previously-known bug in the lazy list-based set: the pseudocode fails to properly initialize the 'marked' field when a new node is added to the list. This simple bug went undetected by a formal correctness proof [6] because the PVS source code did not match the pseudocode in the paper precisely [28]. This confirms the importance of using actual code (rather than pseudocode and manual modeling) for formal verification.

Queue tests: (e,d for enqueue, dequeue)

T0	= (e | d)	Ti2	= e (ed | de)
T1	= (e | e | d | d)	Ti3	= e (de | dde)
Tpc2	= (ee | dd)	T53	= (eeee | d | d)
Tpc3	= (eee | ddd)	T54	= (eee | e | d | d)
Tpc4	= (eeee | dddd)	T55	= (ee | e | e | d | d)
Tpc5	= (eeeee | ddddd)	T56	= (e | e | e | e | d | d)
Tpc6	= (eeeeee | dddddd)		

Set tests: (a, c, r for add, contains, remove)

Sac	= (a | c)	Sar	= (a | r)
Sacr	= (a | c | r)	Saacr	= a (a | c | r)
Sacr2	= aar (a | c | r)	Saaarr	= aaa (r | rc)
S1	= (a' | a' | c' | c' | r' | r')	Sarr	= (a | r | r)

Deque tests: (a_l, a_r, r_l, r_r for add/remove left/right)

D0	= $(a_l\ r_r \mid a_r\ r_l)$	Db	= $(r_r\ r_l \mid a_r \mid a_l)$
Da	= $a_l\ a_l\ (r_r\ r_r \mid r_l\ r_l)$		
Dm	= $(a'_l\ a'_l\ a'_l \mid r'_r\ r'_r\ r'_r \mid r'_l \mid a'_r)$		
Dq	= $(a'_l \mid a'_l \mid a'_r \mid a'_r \mid r'_l \mid r'_l \mid r'_r \mid r'_r)$		

Figure 8. The tests we used. We show the invocation sequence for each thread in parentheses, separating the threads by a vertical line. Some tests include an initialization sequence which appears before the parentheses. If operations need an input argument, it is chosen nondeterministically out of $\{0, 1\}$. Primed versions of the operations are restricted forms that assume no retries (that is, retry loops are restricted to a single iteration).

4.2 Missing Fences

As expected, our testcases revealed that all five implementations require extra memory fences to function correctly on relaxed memory models.

To give a concrete example, we show the source code for the non-blocking queue with appropriate fences in Fig. 9. To our knowledge, this is the first published version of Michael and Scott's non-blocking queue that includes memory ordering fences. We verified that on *Relaxed* these fences are sufficient and necessary for the tests in Fig. 10. Of course, our method may miss some fences if the tests do not cover the scenarios for which they are needed. An interesting observation is that the implementations we studied required only load-load and store-store fences. On some architectures (such as Sun TSO or IBM zSeries), these fences are automatic and the algorithm therefore works without inserting any fences on these architectures.

4.3 Description of Typical Failures

Incomplete initialization. A common failure occurs with code sequences that (1) allocate a new node, (2) set its fields to some value and (3) link it into the list. On relaxed memory models, the stores to the fields (in step 2) may be delayed past the pointer store

```
1  typedef struct node {
2    struct node *next;
3    value_t value;
4  } node_t;
5  typedef struct queue {
6    node_t *head;
7    node_t *tail;
8  } queue_t;
9
10 extern void assert(bool expr);
11 extern void fence(char *type);
12 extern int cas(void *loc,
13                unsigned old, unsigned new);
14 extern node_t *new_node();
15 extern void delete_node(node_t *node);
16
17 void init_queue(queue_t *queue)
18 {
19   node_t *node = new_node();
20   node->next = 0;
21   queue->head = queue->tail = node;
22 }
23 void enqueue(queue_t *queue, value_t value)
24 {
25   node_t *node, *tail, *next;
26   node = new_node();
27   node->value = value;
28   node->next = 0;
29   fence("store-store");
30   while (true) {
31     tail = queue->tail;
32     fence("load-load");
33     next = tail->next;
34     fence("load-load");
35     if (tail == queue->tail)
36       if (next == 0) {
37         if (cas(&tail->next,
38                 (unsigned) next, (unsigned) node))
39           break;
40       } else
41         cas(&queue->tail,
42             (unsigned) tail, (unsigned) next);
43   }
44   fence("store-store");
45   cas(&queue->tail,
46       (unsigned) tail, (unsigned) node);
47 }
48 bool dequeue(queue_t *queue, value_t *pvalue)
49 {
50   node_t *head, *tail, *next;
51   while (true) {
52     head = queue->head;
53     fence("load-load");
54     tail = queue->tail;
55     fence("load-load");
56     next = head->next;
57     fence("load-load");
58     if (head == queue->head) {
59       if (head == tail) {
60         if (next == 0)
61           return false;
62         cas(&queue->tail,
63             (unsigned) tail, (unsigned) next);
64       } else {
65         *pvalue = next->value;
66         if (cas(&queue->head,
67                 (unsigned) head, (unsigned) next))
68           break;
69       }
70     }
71   }
72   delete_node(head);
73   return true;
74 }
```

Figure 9. C code for the non-blocking queue [33], with fences added. It is slightly simplified: the original code stores a counter along with each pointer, which we omit because it is not required in all contexts. No such modifications were made to the other algorithms.

(in step 3). If so, operations by other threads can read the node fields before they contain the correct values, with fatal results. All five implementations showed this behavior. The fix is the same in all cases: adding a store-store fence between steps (2) and (3). For example, the store-store barrier on line 29 of Fig. 9 was added for this reason.

Reordering of value-dependent instructions. Some weak architectures (such as Alpha [7]) allow loads to be reordered even if they are value dependent. For example, the common code sequence (1) read a pointer `p` to some structure and (2) read a field `p->f` is (somewhat surprisingly) susceptible to out-of-order execution: the processor may perform the load of `p->f` before the load of `p` by speculating on the value of `p` and then confirming it afterward [30]. We found this behavior to cause problems in all five implementations. To avoid it, we add a load-load fence between the two instructions. For example, the load-load fence on line 32 in Fig. 9 was inserted for this reason.

Reordering of CAS operations. We model the compare-and-swap operation without any implied fences (Fig. 6). As a result, two CAS instructions to different addresses may be reordered. We observed this behavior only for the nonblocking queue, where it causes problems in the dequeue operation (Fig. 9) if the tail is advanced (line 45) before the node is linked into the list (line 37). To fix this problem, we added a store-store fence on line 44.

Reordering of load sequences. The nonblocking queue uses simple load sequences to achieve some synchronization effects. For example, `queue->tail` is loaded a first time on line 31; next, `tail->next` is loaded (line 33); then, `queue->tail` is loaded a second time (line 35) and the value is compared to the previously loaded value. If the values are the same, the implementation infers that the values that were loaded for `queue->tail` and `tail->next` are consistent (that is, can be considered to have been loaded atomically). A similar load sequence is used in the enqueue operation (lines 52 and 58). For this mechanism to work, we found that the loads in the sequence must not be reordered, and we added a number of load-load fences to achieve this effect (lines 32, 34, 53, 55, 57). The other implementations did not exhibit this behavior.

4.4 Quantitative Results

The performance results confirm that our observation set method provides an efficient way to check bounded executions of concurrent C programs (with up to about 200 memory accesses). Furthermore, they indicate that for our encoding, the choice of the memory model has no significant impact on the tool execution time.

Inclusion check statistics. To illustrate the character of the inclusion checks, we show statistics and graphs in Fig. 10. As described in Section 3.2.1, *CheckFence* encodes the inclusion problem as a CNF formula which is then refuted by the zChaff SAT solver [35] (version 2004/11/15). To keep the trends visible, we do not include the time required for the lazy loop unrolling because it varies greatly between individual tests and implementations.

Specification mining statistics. We show information about the specification mining in Fig. 11a. Most observation sets were quite small (less than 200 elements). The time spent for the specification mining averaged about a third of the total runtime (Fig. 11b). However, in practice, much less time is spent for observation set enumeration because (1) observation sets need not be recomputed after each change to the implementation, and (2) we can often compute observation sets much more efficiently by using a small, fast reference implementation (as shown by the data points for "refset").

	Test Name	Unrolled code instrs	loads	stores	Encoding time [s]	CNF formula vars	clauses	Zchaff [MB]	time [s]	Total time [s]
ms2	T0	142	13	20	0.1	433	5,077	<1	<0.1	0.1
	T1	256	25	33	0.6	1,266	40,454	4	0.2	0.8
	T53	346	33	47	2.0	2,445	125,831	12	1.9	3.9
	T54	346	33	47	1.8	2,378	125,745	12	5.1	6.9
	T55	346	33	47	1.6	2,331	125,788	12	7.2	8.7
	T56	346	33	47	1.6	2,304	123,660	12	3.3	4.9
	Ti2	301	29	40	0.8	1,409	42,418	4	0.2	1.0
	Ti3	370	37	46	1.7	2,116	90,066	12	0.7	2.4
	Tpc2	256	25	33	0.7	1,294	41,553	4	0.1	0.8
	Tpc3	370	37	46	2.4	2,591	142,742	15	1.6	4.0
	Tpc4	484	49	59	7.4	4,324	342,446	47	12.0	19.4
	Tpc5	598	61	72	16.4	6,493	677,390	94	91.6	108.0
	Tpc6	712	73	85	35.0	9,098	1,178,842	186	367.0	402.0
msn	T0	214	22	14	0.2	1,004	12,151	1	<0.1	0.2
	T1	1000	115	55	22.0	14,848	1,597,115	189	314.0	336.0
	T53	966	107	57	22.0	14,536	1,426,104	188	105.0	127.0
	Ti2	843	93	48	8.5	10,407	709,416	94	7.2	15.7
	Ti3	1086	122	60	25.6	16,344	1,633,713	190	35.4	61.0
	Tpc2	454	48	27	1.4	3,496	126,013	12	0.7	2.0
	Tpc3	694	74	40	5.4	7,638	468,274	48	6.5	11.9
	Tpc4	934	100	53	16.4	12,979	1,165,993	186	74.0	90.4
	Tpc5	1174	126	66	42.0	19,679	2,347,472	372	455.0	497.0
	Tpc6	1414	152	79	130.0	27,747	4,133,783	610	1930.0	2060.0
lazylist	Sac	254	29	23	0.5	1,396	24,658	2	<0.1	0.5
	Sar	435	56	39	4.0	4,521	210,799	24	0.8	4.8
	Sacr	505	65	39	5.2	5,435	280,223	47	0.7	5.9
	Saa	543	69	48	8.7	7,120	424,579	80	2.1	10.8
	Saacr	747	97	58	11.7	9,233	504,304	81	3.5	15.2
	Sacr2	1071	141	81	16.3	14,364	555,692	93	11.6	27.9
	Sarr	842	114	67	103.0	15,941	1,731,774	318	47.3	150.3
	S1	821	107	56	18.2	13,610	1,201,727	186	66.4	84.6
	Saaarr	1183	158	93	93.6	23,474	1,945,051	321	86.4	180.0
harris	Sac	406	34	14	0.4	1,882	25,456	2	<0.1	0.5
	Sar	575	51	18	1.4	3,670	85,824	12	0.1	1.6
	Saa	896	77	28	5.2	8,629	333,128	48	1.0	6.3
	Sacr	1349	125	32	28.0	16,411	1,157,264	187	6.9	34.9
snark	Da	760	77	51	4.1	5,229	230,292	24	0.8	4.9
	D0	810	89	65	19.9	9,254	1,075,792	121	9.3	29.2
	Db	980	107	75	47.4	12,278	1,815,494	191	49.6	97.0
	Dm	748	77	57	8.1	8,086	698,752	62	28.7	36.8
	Dq	748	77	57	7.0	8,015	710,252	62	123.0	130.0

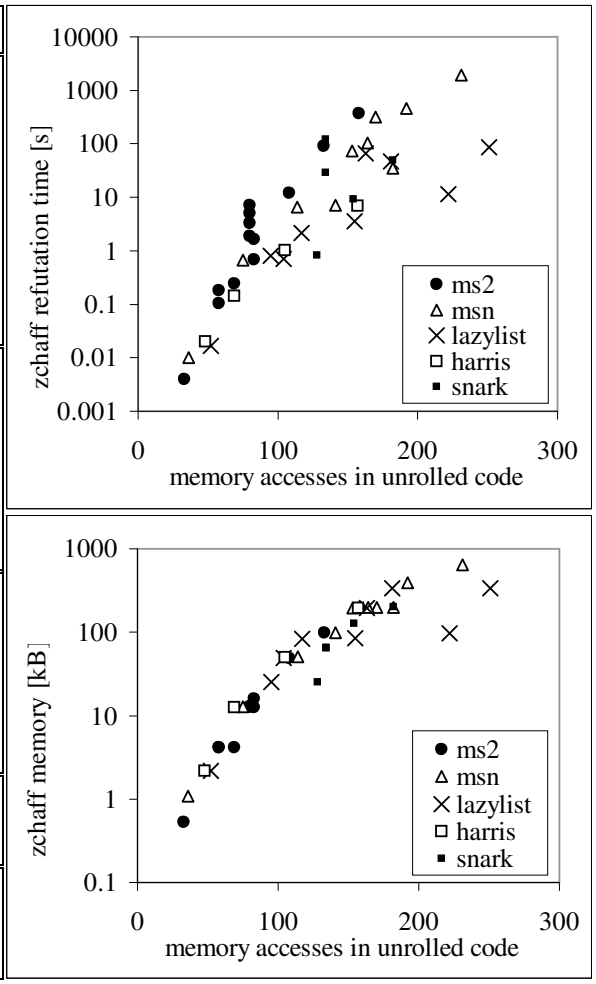

Figure 10. (a) left: statistics about the inclusion checks. For a given implementation (listed in Table 1) and test (listed in Fig. 8), we show (from left to right): the size of the unrolled code, the time required to create the SAT instance, the size of the SAT instance, the resources required by the SAT solver to refute the SAT instance, and the overall time required. All measurements were taken on a 3 GHz Pentium 4 desktop PC with 1GB of RAM, using zchaff [35] version 2004/11/15. **(b)** right: charts show (on a logarithmic scale) how time and memory requirements increase sharply with the number of memory accesses in the unrolled code. The data points represent the individual tests, grouped by implementation.

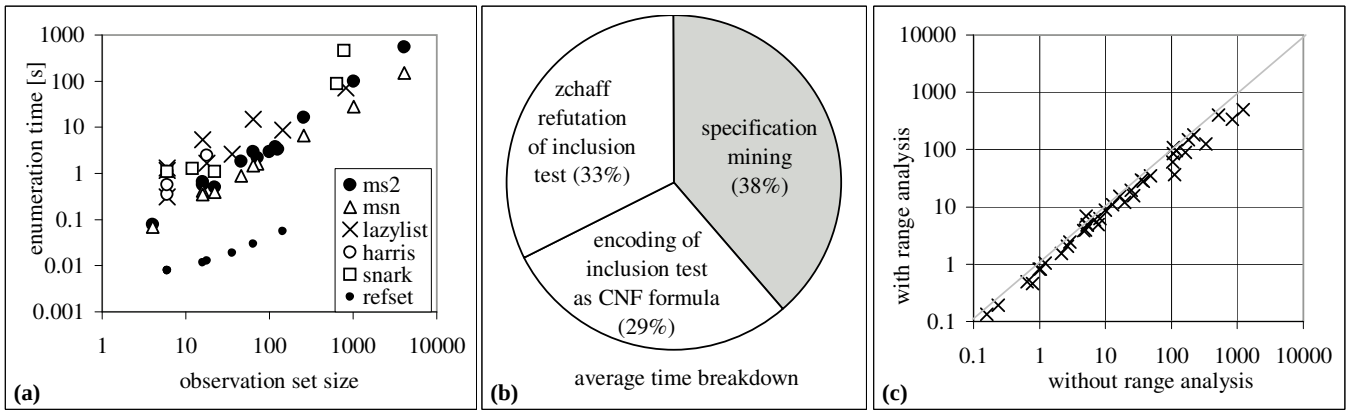

Figure 11. (a) characterization of specification mining. The data points represent the individual tests (Fig. 8), grouped by implementation. **(b)** average breakdown of total runtime. **(c)** impact of range analysis on runtime. The data points represent the individual tests.

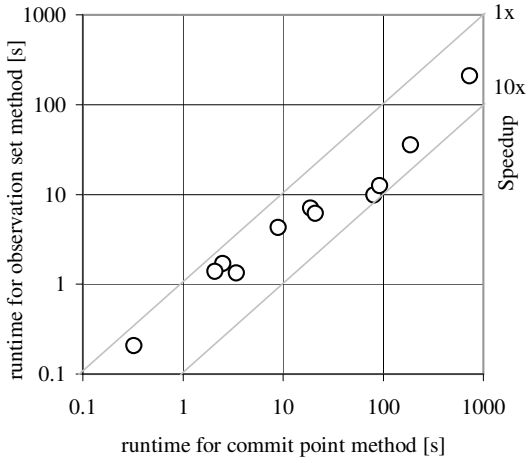

Figure 12. Speed comparison between our observation set method and the commit point method [4]. Each data point corresponds to an individual test. Both axes are logarithmic, diagonal lines show constant ratios. The average speedup is $2.61\times$.

Impact of range analysis. As described earlier, we perform a range analysis prior to the encoding to obtain data bounds, alias analysis, and range information. This information is used to improve the encoding by reducing the number of boolean variables. Fig. 11c shows the effect of the range analysis on runtime. On average, the performance improvement was about 42%. On larger testcases (where we are most concerned), the positive impact is more pronounced (the tool finished up to $3\times$ faster).

Choice of memory model. All tests use the memory model *Relaxed* (see Sections 2.3 and 3.2.1). To find out if the choice of memory model has an effect on the runtime, we separately evaluated the runtime for a sequentially consistent memory model. The results indicate that on average, performance is about 4% faster for sequential consistency, which is insignificant.

5. Related Work

Most prior work on verification of concurrent data types is based on interactive proof construction and assumes a sequentially consistent memory model [6, 14, 40, 46, 50]. To our knowledge, analogous proof strategies for relaxed memory models have not been investigated.

Specialized algorithms to insert memory fences automatically during compilation have been proposed early on [11, 42]. However, these methods are based on a conservative program analysis, which makes them less attractive for highly optimized implementations: fences can have a considerable performance impact [45, 47] and should be used sparingly.

Most previous work on model checking executions on relaxed memory models has focused on relatively small and hand-translated code snippets (such as spinlocks or litmus tests). It can be divided into two categories: explicit-state model checking combined with operational memory models [9, 38], and constraint solving combined with axiomatic memory models [4, 15, 51].

We prefer the latter approach for two reasons: (1) axiomatic models can more easily capture official specifications because the latter use an axiomatic style, and (2) constraint-based encodings can leverage the advances in SAT solving technology.

When compared to our earlier case study [4], the method presented in this paper differs as follows:

- We allow the operations of the implementation to be written as C code (rather than requiring a manual translation). This improves the degree of automation and the precision, at the expense of a somewhat larger encoding.
- Our observation set method is more automatic and more general, because it does not require commit point specifications. Implementations such as the lazy list-based set [18] are not known to have commit points [6, 46].
- Our observation method is faster. Fig. 12 shows a direct speed comparison on a logarithmic scale; the diagonal lines show constant speed ratios. On average, the speedup was about $2.61\times$, but it approached an order of magnitude on some tests.

6. Conclusions

Verifying concurrent data type implementations that make deliberate use of data races and memory ordering fences is challenging because of the many interleavings and counterintuitive instruction reorderings that need to be considered. Conventional verification tools for multithreaded programs are not sufficient because they make assumptions on the programming style (race-free programs) or the memory model (sequential consistency).

Our *CheckFence* prototype fills this gap and provides a valuable aid to algorithm designers and implementors because it (1) accepts implementations written as C code, (2) supports relaxed memory models, memory ordering fences, and lock-free synchronization and (3) can verify that the implementation behaves correctly for a given bounded test or will produce a counterexample trace if it does not.

Future work includes (1) enhancements to the front-end to support more C features and data type implementations from the literature, and (2) the use of SMT solvers and customized decision procedures to improve the efficiency of the back end. We are also working on applying our method to memory models that are weaker than *Relaxed* (such as the PowerPC model [13]) or defined at the language level (such as the new Java Memory Model [29]).

References

[1] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

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

[3] H.-J. Boehm. Threads cannot be implemented as a library. In *Programming Language Design and Implementation (PLDI)*, pages 261–268, 2005.

[4] S. Burckhardt, R. Alur, and M. Martin. Bounded verification of concurrent data types on relaxed memory models: a case study. In *Computer-Aided Verification (CAV)*, LNCS 4144, pages 489–502. Springer, 2006.

[5] 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.

[6] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Computer-Aided Verification (CAV)*, LNCS 4144, pages 475–488. Springer, 2006.

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

[8] D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent deques. In *Conference on Distributed Computing (DISC)*, LNCS 1914, pages 59–73. Springer, 2000.

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

[10] S. Doherty, D. Detlefs, L. Grove, C. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS is not a silver bullet for nonblocking algorithm design. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 216–224, 2004.

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

[12] C. Flanagan and S. Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000.

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

[14] H. Gao and W. Hesslink. A formal reduction for lock-free parallel algorithms. In *Computer-Aided Verification (CAV)*, LNCS 3114, pages 44–56. Springer, 2004.

[15] 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.

[16] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *Conference on Distributed Computing (DISC)*, LNCS 2180, pages 300–314. Springer, 2001.

[17] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *Conference on Distributed Computing (DISC)*, LNCS 2508, pages 265–279. Springer, 2002.

[18] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems (OPODIS)*, 2005.

[19] T. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Programming language design and implementation (PLDI)*, pages 1–13, 2004.

[20] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, November 2006.

[21] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual, Book 2, rev. 2.2*, January 2006.

[22] Intel Corporation. *Intel Threading Building Blocks*, September 2006.

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

[24] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. In *TV'06 Workshop, Federated Logic Conference (FLoC)*, pages 66–77, 2006.

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

[26] L. Lamport. Checking a multithreaded algorithm with $^{+}$CAL. In *Conference on Distributed Computing (DISC)*, LNCS 4167, pages 151–163. Springer, 2006.

[27] D. Lea. The java.util.concurrent synchronizer framework. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, 2004.

[28] V. Luchangco. Personal communications, October 2006.

[29] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, pages 378–391, 2005.

[30] 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.

[31] M. Michael. Scalable lock-free dynamic memory allocation. In *Programming Language Design and Implementation (PLDI)*, pages 35–46, 2004.

[32] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

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

[34] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Principles of distributed computing (PODC)*, pages 219–228, 1997.

[35] 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.

[36] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Programming Language Design and Implementation (PLDI)*, pages 308–319, 2006.

[37] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conf. on Compiler Constr. (CC)*, 2002.

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

[39] P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Programming Language Design and Implementation (PLDI)*, pages 320–331, 2006.

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

[41] 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.

[42] 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.

[43] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

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

[45] O. Trachsel, C. von Praun, and T. Gross. On the effectiveness of speculative and selective memory fences. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[46] 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.

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

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

[49] M. Xu, R. Bodik, and M. Hill. A serializability violation detector for shared-memory server programs. In *Programming Language Design and Implementation (PLDI)*, 2005.

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

[51] 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.