The 'Test Model-Checking' Approach to the Verification of Formal Memory Models of Multiprocessors*

Ratan Nalumasu, Rajnish Ghughal,
Abdel Mokkedem and Ganesh Gopalakrishnan

Department of Computer Science, University of Utah,
Salt Lake City, UT 84112–9205

Contact email: ratan,ghughal,mokkedem,ganesh@cs.utah.edu

Abstract. We offer a solution to the problem of verifying formal memory models of processors by combining the strengths of model-checking and a formal testing procedure for parallel machines. We characterize the formal basis for abstracting the tests into test automata and associated memory rule safety properties whose violations pinpoint the ordering rule being violated. Our experimental results on Verilog models of a commercial split transaction bus demonstrates the ability of our method to effectively debug design models during early stages of their development.

1 Introduction

The fundamentally important problem [1] of verifying whether a given *memory system model* (or “a memory system”) provides a *formal memory model* (or “memory model”) appears in a number of guises. CPU designers are interested in knowing whether some of the aggressive execution techniques such as speculative issue of memory operations violate sequential consistency; I/O bus designers are interested in knowing the exact semantics of shared accesses provided by split I/O transactions [9]; even language designers of multi-threaded languages such as Java that support shared updates [14] are interested in this problem. Formal verification methods are ideally suited for this problem because: (i) the semantics of memory orderings are too subtle to be fathomed through informal reasoning alone; (ii) *ad hoc* testing methods cannot provide assurance that the desired memory model has been implemented. Unfortunately, despite the central importance of this problem and the large body of formal methods research in this area, there is still no single formally based method that the designer of a realistic multiprocessor system can use on his/her detailed design model to *quickly* find violations in the design. In this paper we describe such a method called *test model-checking*.

* Supported in part by ARPA Order #B990 Under SPAWAR Contract #N0039-95-C-0018 (Avalanche), DARPA under contract #DABT6396C0094 (Utah Verifier), and NSF MIP MIP–9321836.

Test model-checking formally adapts to the realm of model-checking a formally based architectural testing method called ARCHTEST. ARCHTEST has been successfully used on a number of commercial multiprocessors [7] by running a suite of test-programs on them. ARCHTEST is an *incomplete* testing method in that it does not, under all circumstances, detect violations of memory orderings [8]. Nevertheless, its tests have been shown to be incisive in practice [7]. Most importantly, the formal theory of memory ordering rules developed by Collier in [8] forms the basis for ARCHTEST, which means that whenever a violation is detected by ARCHTEST, there is a formal line of reasoning leading back to the precise cause.

Being based on ARCHTEST, test model-checking is also incomplete. However, none of the (presumed) complete alternatives to date have been shown to be practical for verifying large designs. For example [25] involves the use of manually guided mechanical theorem proving. Even approaches based on *conventional* model-checking are impossibly difficult to use in practice. For example, the assertions pertaining to the sequential consistency of lazy caching [11], a simple memory system, expressed in various temporal logics (by [15] in $\forall\text{CTL}^*$ [6] and [19] in TLA [21]) are highly complex. We do not believe that descriptions of this style will scale up. On the other hand, the test model-checking method has not only been able to comfortably handle the memory system defined by the symmetric multiprocessor (SMP) bus called *Runway* [4] used by Hewlett-Packard in their high-end machines, but also it discovered many subtle bugs in early models describing this bus that *we created*. Our Utah Runway Model(URM) includes a number of details such as split transactions, out of order transaction completions, and even an element of speculative execution. The errors we made in capturing these details could well have been made in an actual industrial context. We believe that with growing system complexity, the role of debugging methods that are effective and are formally based will only grow in significance, regardless of whether the methods are complete or not.

Our specific contributions in this paper are: (i) the adaptation of a formal testing method for memory models to model-checking, that can be applied during the design of modern microprocessors whose memory systems are very complex; (ii) a formal characterization (accompanied by proofs) of *how* the tests of the testing method are abstracted and turned into a fixed set of safety properties that are then model-checked; and (iii) experimental results on three examples using the VIS model-checker, the last example being much larger than any previously reported in this context.

Related Work

A technique very similar to test model-checking was proposed in [22] under the section heading ‘Sequential Consistency’. To give a historic perspective, our test model-checking idea originated in our attempt to answer the following two questions: (i) which memory ordering rule(s) is [22] really verifying? (ii) is this a general technique? *i.e.* can other memory ordering rules be verified in the same fashion? We still have not found a satisfactory answer to the first question

because the test in [22] uses only one location which then couldn't make it a test for *sequential consistency*; it could plausibly be a test for coherence—which again does not correspond to what Collier formally proves in [8]. One of our contributions is that we answer these questions by elaborating on the theoretical as well as practical aspects of test model-checking.

Alur et al [3] showed that the problem of finding if there is a sequentially consistent string σ in a regular expression r is undecidable. The problem is undecidable as the actions that follow a read instruction can depend on the value returned by a read instruction. This result is not applicable in our case, because the models we consider do not make decisions based on the value returned by a read instruction. This is detailed in Section 3.1.

In [25], the authors use a method called *aggregation* on a distributed shared memory coherence protocol used in an experimental multiprocessor, to arrive at a simplified model of system behavior. Their technique involves manual theorem proving. The work in [17] as well as [10] are aimed at verifying that synchronization routines work correctly under various memory models, where the memory models themselves are described using finite-state operational models. They do not address the problem of establishing the memory models provided by detailed memory subsystem designs, which is our contribution. In [13], the authors analyze the problem of deciding whether a given set of traces are sequentially consistent. Our approach differs in that we prove properties about design level models of memory system rather than traces (presumably obtained from actual machines) and hence designers can use our approach in early stages of design cycle.

2 Overview of ARCHTEST

ARCHTEST is based on the theory presented in [8] that formally defines and characterizes architectural *rules* obeyed by memory subsystems of multiprocessors. Although these rules are *elemental*, in realistic memory systems the rules manifest in *compound* form. Obeying a compound rule is tantamount to obeying *all* the constituent elemental rules; violating a compound rule is tantamount to violating *any* of the constituent elemental rules. Some of the elemental ordering rules are:

Rule of Computation (CMP): This is a basic rule defining how the terminal value of each operand is calculated from the initial values of the operands.

Though most of the literature on memory architectures implicitly assumes this rule, we will often keep it explicit in our discussions.

Rule of Read Order (RO): For any pair of read events a and b in the same process, if a comes before b in program order then a happens before b .

Rule of Write Order (WO): Same as RO with 'write' substituted for 'read'.

Rule of Program Order (PO): For any pair of events a and b in the same process, if a comes before b in program order then a happens before b . Event a or b can be either read or write events. So, both RO and WO are special

Fig. 1. $Test_{ROWO}$ - ARCHTEST test and test automata for $A(CMP, RO, WO)$

Fig. 2. $Test_{WA}$: ARCHTEST test for $A(CMP, RO, WO, WA)$

cases of PO. This is one of the strongest ordering rules and is essential for sequential consistency.

Rule of Write Atomicity (WA): A write operation becomes visible to all processes instantaneously. More precisely, one conceptual *store* S_i is associated with each processor node P_i . Then, for each write operation W , one write event W_i is defined per store S_i . Then, WA guarantees that there are no i, j and no event e such that e is before W_i and is after W_j .

$Test_{ROWO}$: ARCHTEST test for $A(CMP, RO, WO)$ The test of ARCHTEST for the *compound* rule consisting of the elemental rules CMP , RO , and WO , denoted $A(CMP, RO, WO)$, is shown in Figure 1(a). (Figure 1(b) will be discussed later.) Process P_1 executes a sequence of write instructions (intended to check for WO), and P_2 executes a sequence of read instruction (intended to check for RO). If the memory system correctly realizes $A(CMP, RO, WO)$, then Condition 1 produces a positive outcome:

Condition 1 (Monotonic) *The sequence of X values is monotonically increasing, i.e.: $\forall i, j : 1 \leq i \leq j \leq k : X[i] \leq X[j]$ or equivalently $\forall i : 1 \leq i \leq k - 1 : X[i] \leq X[i + 1]$.*

If MONOTONIC condition is violated then at least one of the CMP , RO and WO rules is violated.

$Test_{WA}$: ARCHTEST test for $A(CMP, RO, WO, WA)$ $Test_{WA}$, shown in Figure 2 tests for $A(CMP, RO, WO, WA)$, with the conditions checked being: (i) the MONOTONIC condition (suitably modified for arrays U, V, X, Y), and (ii) ATOMIC, which is:

Initially $A = B = 0$

$L_{11} : A := 1;$	$L_{11} : B := 1;$
$L_{12} : Y[1] := B;$	$L_{12} : X[1] := A;$
$L_{21} : A := 2;$	$L_{21} : B := 2;$
$L_{22} : Y[2] := B;$	$L_{22} : X[2] := A;$
...	...
$L_{k1} : A := k;$	$L_{k1} : B := k;$
$L_{k2} : Y[k] := B;$	$L_{k2} : X[k] := A;$

(a) $Test_{PO}$ of ARCHTEST(b) Test automata for $Test_{PO}$ **Fig. 3.** $Test_{PO}$: ARCHTEST test for $A(CMP, PO)$

Condition 2 (Atomic) $\forall i, j : 1 \leq i, j \leq k : V[i] \geq X[j] \vee Y[j] \geq U[i].$

ATOMIC watches for the possibility that a write operation from P_1 and a write operation from P_4 became visible in different orders to P_2 and P_3 .

$Test_{PO}$: ARCHTEST test for $A(CMP, PO)$ $Test_{PO}$, shown in Figure 3(a) tests for $A(CMP, PO)$, with the conditions checked being: (i) the MONOTONIC condition (suitably modified for arrays X, Y), and (ii) PO_CROSS, which is:

Condition 3 (PO_Cross) $\forall i, j : 1 \leq i, j \leq k : (X[i] \geq j \vee Y[j] \geq i) \wedge (X[i] \leq j \vee Y[j] \leq i).$

All ARCHTEST test programs such as $Test_{WA}$, $Test_{PO}$ etc. are meant to be run on real machines and there cannot be any real guarantees that the particular interleavings that reveal violations (such as for memory ordering rule WA watched by condition ATOMIC in $Test_{WA}$) will indeed happen. To allow for as many interleavings as possible, ARCHTEST recommends that its tests be run for large values of k . With test model-checking, we effectively run the tests for $k = \infty$, as will be elaborated shortly.

3 Test model-checking

Test model-checking converts the tests of ARCHTEST to corresponding *memory rule test automata* (“test automata”) that drive the model of the memory system being examined. In our experiments, we use the Verilog language supported by VIS [26] to capture the memory system models as well as the test automata. Precisely, the automata are modeled as (Verilog) processes which run in parallel with the memory system and simply enqueue test automata instructions in the instruction stream of the processors. The CONDITIONS corresponding to each compound memory rule being tested are turned into corresponding *memory rule safety properties* that are checked by the VIS tool.

3.1 Assumptions about memory systems realized in hardware

Memory systems realized in hardware as well as finite-state models thereof are assumed to be *data independent*; i.e., the control logic of the system moves data around, and does not base its control-point settings on the data values themselves. We also assume that the system is address *semi-dependent* [16], i.e. the control logic can at most compare two addresses for equality or inequality and base its actions on the outcome of this test. These assumptions are standard, and form the basis for defining test automata as well as memory rule safety properties.

3.2 Creation of test automata

As illustrated in Figure 1(b), we obtain test automata for various memory models by finitely abstracting the data used in ARCHTEST tests, using non-determinism to justify the abstraction. For example, we abstract the specific activities of process P_1 of Figure 1(a) into that of (non-deterministically) writing *all possible* ascending values over $\{0,1\}$, as shown in P_1 of Figure 1(b). Also, since we cannot store infinite arrays in creating process P_2 , we turn P_2 and the corresponding memory rule safety property into an automaton that checks that the array values read are monotonically increasing. This, in turn, can be performed using just two *consecutive* array values $x1$ and $x2$ that are nondeterministically recorded by P_2 . Hence, the memory rule safety property we model-check for is: P_2 in final state $\Rightarrow x2 \geq x1$.

We now provide a justification that these abstractions preserve the memory rule safety properties, i.e., for the same memory system model, a violation of a condition occurs in a test of ARCHTEST for $k = \infty$ iff the same violation will occur in model-checking the corresponding memory rule safety property when test automata are used to drive the memory system model. To keep the presentation simple, we formally argue how the test automata finds every violation present in the test of ARCHTEST with $k = \infty$; the opposite direction of *iff*, i.e. how a test of ARCHTEST with $k = \infty$ finds violations found by the test automata is easy to see because the test automata just appears as a “stuttering” of the test of ARCHTEST. For example, the actions of P_1 in Figure 1(a) can be viewed as repeating the initialization and then repeating the instruction at label L_1 of P_1 of Figure 1(a). Our proof sketches are illustrated on the three tests presented in Section 2.

3.3 Abstracting Test_{ROWO}

We show that if the test program in Test_{ROWO} shows that MONOTONIC is violated, then the test automata also reveals the error. Since MONOTONIC is violated,

$$\begin{aligned}
 & \exists i : \quad 1 \leq i < k : X[i] > X[i+1] \\
 \Leftrightarrow & \exists i, \alpha : 1 \leq i < k : (X[i] > \alpha) \wedge (X[i+1] \leq \alpha) \\
 \Leftrightarrow & \exists i, \alpha : 1 \leq i < k : (X[i] > \alpha) \wedge \neg(X[i+1] > \alpha)
 \end{aligned}$$

Fig. 4. Abstraction of $Test_{ROWO}$

Since, the last formula compares $X[i]$ and $X[i+1]$ only to α , we can rewrite the test program as shown in Figure 4(a) *assuming data independence*, and rewrite the last formula as $\exists i : 1 \leq i < k : X[i] = 1 \wedge X[i+1] = 0$. Note that in Figure 4(a) all reads of A occur in the expression $A > \alpha$. Hence, we can replace every $A := v$ with $A := (v > \alpha)$ and $X[i] := (A > \alpha)$ with $X[i] := A$ without affecting MONOTONIC again, *if data independence holds*, to obtain Figure 4(b). Figure 4(c) is obtained by simplifying Figure 4(b): each $v > \alpha$ evaluates to 0 for $v \leq \alpha$ and 1 otherwise. This figure is generalized to obtain the test automata in Figure 1(b). Intuitively the automata finds the violation as follows. P_1 remains in the initial state for α iterations (executing $A:=0$) and then switches to second state (executing $A:=1$). Also, P_2 remains in the initial state for $i-1$ iterations and then switches to second state recording x_1 and then x_2 (dashed edges show when these variables are recorded). Thus the test automata's execution is identical to that in Figure 4(c) except that the test automata gives the effect of taking k to ∞ . Also notice that x_1 and x_2 get the values corresponding to $X[i]$ and $X[i+1]$. Also, corresponding to $X[i] = 1 \wedge X[i+1] = 0$, we have $x_1 = 1 \wedge x_2 = 0$. Hence the memory rule safety property corresponding to condition MONOTONIC is found violated by the test automata exactly when $Test_{ROWO}$ for $k = \infty$ detects a violation. Note that the nondeterminism employed in constructing test automata enables P_1 and P_2 to *guess* the right value of α and i corresponding to the violation.

3.4 Abstracting $Test_{WA}$

Test automata for $Test_{WA}$ is shown in Figure 5. In this automata P_1 and P_4 write all possible ascending sequences of $\{0, 1\}$ in A and B respectively. Each processor *independently* and *nondeterministically* decides to switch from writing 0 to writing 1. Modifications similar to those in $Test_{ROWO}$ are applied to P_2 and P_3 also, to (nondeterministically) decide which $U[i], V[i]$ pair and $X[j], Y[j]$ pair are recorded in u, v and x, y . The memory rule safety property corresponding to condition ATOMIC is: P_2 and P_3 in their final states $\Rightarrow v \geq x \vee y \geq u$. As was explained in Section 3.2 for $Test_{ROWO}$, our abstraction avoids having to

Fig. 5. $Test_{WA}$ test automata : Test Automata for $A(CMP, RO, WO, WA)$

remember the entire extent of the arrays U , V , X , and Y . (In $Test_{WA}$, one has to check for MONOTONIC also; this is done similarly to $Test_{ROWO}$.)

To show that the abstraction preserves ATOMIC, let ATOMIC be violated in $Test_{WA}$ of ARCHTEST. Hence

$$\begin{aligned} & \exists i, j : \quad U[i] > Y[j] \wedge X[j] > V[i] \\ \iff & \exists i, j, \alpha, \beta : \quad Y[j] = \alpha \wedge U[i] > \alpha \wedge V[i] = \beta \wedge X[j] > \beta \end{aligned}$$

Similar to $Test_{ROWO}$, assuming *data-independence*, we have an execution of the test automata (Figure 5) in which P_1, P_2, P_3, P_4 iterates for $\alpha, i-1, j-1, \beta$ times (respectively) in their initial states before switching to their final states. This test automata execution detects violations of ATOMIC exactly when $Test_{WA}$ for $k = \infty$ would. A violation of ATOMIC happens exactly when $u = 1 \wedge v = 0 \wedge x = 1 \wedge y = 0$.

3.5 Abstracting $Test_{PO}$

We now discuss $Test_{PO}$ for the elemental ordering rule Program Order (PO), which is somewhat more complex than the previous two tests.

We obtain the test automata and the memory rule safety property for $Test_{PO}$ of Figure 3(a) as illustrated in Figure 3(b). P_1 executes a pair of instructions: write to A followed by read from B , infinitely often. The value written to A is 0 for some iterations and is nondeterministically changed to 1. P_2 runs similarly. P_1 nondeterministically selects a pair of write instruction followed by read instruction. It assigns the value written to A to j and the value read from B to y . Similarly, P_2 updates i and x . The dashed edges in Figure 3(b) show when x, y, i, j are updated. The memory rule safety property corresponding to condition PO_CROSS is: P_1 and P_2 in their final states $\Rightarrow (x \geq j \vee y \geq i) \wedge (x \leq j \vee y \leq i)$. We can show that this abstraction preserves PO_CROSS by an argument similar to that for $Test_{ROWO}$ and $Test_{WA}$.

4 Case Studies

To demonstrate the effectiveness of our approach, we verified three different memory systems, namely serial memory, lazy caching, and URM, all using a symbolic model-checker VIS (See [26] for more information on VIS). These three

Event	Action or condition
$R_i(d, a)$	if $\text{Mem}[a] = d$
$W_i(d, a)$	$\text{Mem}[a] := d$

Fig. 6. Serial memory transaction rules

memory systems are described in some detail below, along with some of the subtle bugs that we could detect using test model-checking. Details of all our experiments can be obtained from the Web [23] or by contacting the authors.

4.1 Sequential Consistency

A sequentially consistent memory system [20] requires that there be a single self-consistent trace t of memory operations that when projected onto the memory operations of each individual processor P_i ($R_i(a, d)$ and $W_i(a, d)$ for processor i) is according to program order for P_i . As suggested in [8], one's intuition about sequential consistency matches the behaviour described by $A(CMP, PO, WA)$.

As [8] does not list a single compound test to check for $A(CMP, PO, WA)$, we can use the following two tests that are available: $Test_{WA}$ which tests for $A(CMP, RO, WO, WA)$ and $Test_{PO}$ which tests for $A(CMP, PO)$. This combination is exactly equivalent to testing for sequential consistency because PO implies RO and WO (as formally defined in [8]). While sequential consistency matches the behavior described by $A(CMP, PO, WA)$, successful test model-checking outcomes are only necessary but not sufficient conditions to ensure sequential consistency. As part of our future work, we are exploring the ways to arrive at sufficient conditions for such tests. For every memory system we consider, these two tests are model-checked separately and summarized in Figure 7.

4.2 Serial memory and Lazy caching

The **serial memory** protocol for n processors and a memory is shown in Figure 6. Serial memories are often used to define sequential consistency(SC) operationally. The **lazy caching** protocol [2, 11], (Figure 4 in [11] describes the transition system of the protocol), also implements sequential consistency, and is geared towards a bus based architecture. The memory interface still consists of reads and writes; however, caches C_i are interposed between the shared memory Mem and the processors P_i . Each cache C_i contains a part of the memory Mem and has two queues associated with it: an out-queue Out_i in which P_i write requests are buffered and an in-queue IN_i in which the pending cache updates are stored. These queues model the asynchronous behavior of write events in a sequentially consistent memory. A write event $W_i(a, d)$ doesn't have an immediate effect. Instead, a request (d, a) is placed in Out_i . When the write request is taken out of the queue, by an internal memory-write event $MW_i(a, d)$, the memory is updated and a cache update request (d, a) is placed in every in-queue. This cache update is eventually removed by an internal cache update event $CU_j(a, d)$ as a result of which the cache C_j gets updated. Cache evictions are modeled by

internal cache invalidate events: CI_i can arbitrarily remove locations from cache C_i . Caches are filled both as the delayed result of write events and through internal memory-read events, $MR(a, d)$. The latter events model the effect of a cache-miss: in that case the read event stalls until the location is copied from the memory. A read event $R_i(a, d)$, predictably, stalls until a copy of location a is present in C_i but also until the copy contains a correct value in the following sense: SC demands that a processor P_i reads the value at a location a that was recently written by P_i unless some other processor updated a in the meantime. Hence, a read event $R_i(a, d)$ cannot occur unless all pending writes in Out_i are processed as well as the cache updates requests from In_i that corresponds to writes of P_i . For this reason, such cache update requests are marked (with a $\star$).

4.3 Utah Runway Model (URM)

Our third example, called URM, is modeled after a commercial bus used to interconnect processors and memory controller together to form a multiprocessor system. The behavior of this memory system is described in some detail in [4,5,18]. The complexity of this protocol stems from many sources, a few of which are elaborated here (see [24] for more details). First, the queues in the clients introduce decoupled execution, leading to a large number of “otherwise equivalent” states. Next, the control mechanism is very complex, owing to many reasons, including: (i) lines can be obtained in various sharing modes such as read-shared-private and read-private; (ii) line states can be eagerly promoted to *private* before the data actually arrives (concurrent dirtying are merged into when the data arrives); (iii) hit after miss situations can be speculatively processed and unrolled when invalidated. Though we did not try to model each of these features in their full glory, we *did* include a modicum of these aggressive features into our URM, which in fact occupies more than 2,000 lines of VIS Verilog code[23].

4.4 Verification results

The tables in figure 7 show execution time for model-checking our Serial memory, Lazy caching and URM models for tests of A(CMP, PO) and A(CMP,RO,WO,WA) (recall that A(CMP, PO, WA) implies SC). The three models running separately the two tests $Test_{WA}$ and $Test_{PO}$ are model-checked for the following conditions: (Figure 3(b) does not show some of these states)

$$\begin{aligned}
 Test_{WA}: \text{MONOTONIC: } & \wedge (P_2.inS_2) \Rightarrow (P_2.U_1 \leq P_2.U_2) \wedge (P_2.inS_2) \Rightarrow (P_2.V_1 \leq P_2.V_2) \\
 & \wedge (P_3.inS_2) \Rightarrow (P_3.X_1 \leq P_3.X_2) \wedge (P_3.inS_2) \Rightarrow (P_3.Y_1 \leq P_3.Y_2) \\
 & \text{ATOMIC: } (P_2.inS_1 \wedge P_3.inS_1) \Rightarrow (P_2.V \geq P_3.X \vee P_3.Y \geq P_2.U) \\
 Test_{PO}: \text{PO_CROSS: } & (P_1.inS_3 \wedge P_2.inS_3) \Rightarrow \\
 & (P_1.Y \geq P_2.I \vee P_2.X \geq P_1.J) \wedge (P_1.Y \leq P_2.I \vee P_2.X \leq P_1.J)
 \end{aligned}$$

As can be seen, all these conditions are safety properties, and independent of the model itself, which is a distinct advantage over other methods.

A(CMP,PO)	#states	#bdd nodes	conditions verified	runtime (mn:sec)
serial memory	7229	7145	Vacuity PO_CROSS	00:02 00:09
lazy caching	7.80248e+06	306692	Vacuity PO_CROSS	01:12 36:33
URM	953675	1657308	Vacuity PO_CROSS	14:23 27h28:30

A(CMP,WO,RO,WA)	#states	#bdd nodes	conditions verified	runtime (mn:sec)
serial memory	21242	10084	Vacuity MONOTONIC, ATOMIC	00:04 00:34
lazy caching	1.90736e+06	513655	Vacuity MONOTONIC, ATOMIC	02:02 59:33
URM	985236	1695092	Vacuity MONOTONIC, ATOMIC	17:24 40h17:33

Vacuity: Antecedent of $\Rightarrow$ is not always false

Fig. 7. Verification results using VIS on a SPARC ULTRA-1 with 512 MB Memory

The size of the state space and number of nodes in BDDs are also reported. Note that lazy caching has more states than URM due to the queues present in the model. However, the complexity of the URM protocol is much higher, which results in large BDD size and higher run time. However, in all our experiments, whenever there was any memory ordering rule violation in our model, test model-checking detected it quickly (in the order of minutes). A very desirable feature one can provide in a tool based on test model-checking is a *menu* of previously generated test automata for the various compound rules in [8], using which designers can probe their model.

Our Verilog models captures quite faithfully the cache coherence protocol and the ordering rules of the three memory systems. After an extensive debugging using test model-checking driven by $Test_{PO}$ and $Test_{WA}$, we have a high confidence that the memory systems built based on the Lazy caching model or URM would be sequentially consistent.

Description of a Bug found in a preliminary model of lazy caching: The following bug in our model of Lazy Caching was caught by a violation of PO_CROSS in $Test_{PO}$. The bug was in the queues used by Lazy Caching, which were implemented as shift registers. We forgot to shift the $\star$ -bit in In_i when the processor P_i receives a cache-update from In_i queue. With this bug it is possible that In_i queue is not $\star$ -ed when it should be, and consequently reads in P_i may bypass writes. This results in a violation of PO. This is a difficult bug to catch because its detection involves understanding the complex feedback from all components of the protocol to each other (queues, memory, and caches). Moreover, this bug is interesting because it violates PO but doesn't violate WA. This is so because only write-read (WR) order is affected by this bug. Our technique effectively caught this bug: the PO_CROSS condition does not pass when we model-checked the model for $Test_{PO}$. However, $Test_{WA}$ (note that it doesn't involve PO) *passes!* This shows the futility of *ad hoc* testing methods:

one could apply subjective criteria to consider a test similar to $Test_{WA}$ to be sufficiently incisive, when in fact it fails to account for a crucial ordering relation such as PO.

We found a more complex bug in URM. Details are too involved to report here, but can be easily obtained from the web site [23, 12].

5 Conclusion and Future Plans

We presented a new approach to verify multi-processors for formal memory models, which combines two existing powerful techniques: model-checking, and the testing method of ARCHTEST. From our results, we conclude that test model-checking can be of great value in detecting bugs during early stages of the design cycle of modern microprocessors whose memory subsystems are complex. Our results on our URM of the HP PA/Runway bus attest to this.

So far we have identified the rules and corresponding tests for sequential consistency. We are currently working on identifying similar rules and tests for other well-known formal memory models such as TSO, PSO, and RMO [1] that are described in the SPARC V9 architecture manual [27]. This work may involve considering memory barrier operations and defining new rules as well as new tests. We are currently working to formulate some reasonable assumptions about the memory system model under which the tests administered by our test automata can be rendered complete. Also, for a limited class of models, model-checking the test for some small value of k might actually be sufficient. Our initial attempts in this direction are encouraging.

Acknowledgments: We would like to thank W.W. Collier for his help in explaining his work, his very informative emails and providing ARCHTEST, Dr. Narendran for many fruitful discussions, and Dr. Al Davis and his Avalanche team for offering us the unique opportunity to work on state-of-art processors and busses.

References

1. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.
2. Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.
3. Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 219–228, July 1996.
4. William R. Bryg, Kenneth K. Chan, and Nicholas S. Fiduccia. A high-performance, low-cost multiprocessor bus for workstations and midrange servers. *Hewlett-Packard Journal*, pages 18–24, February 1996.
5. Albert Camilleri. A hybrid approach to verifying liveness in a symmetric multiprocessor. In *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ*, pages 49–67, August 1997. Springer-Verlag LNCS 1275.

6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.
7. W. W. Collier. Multiprocessor diagnostics. <http://www.infomall.org/diagnostics/archtest.html>.
8. W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
9. Francisco Corella, April 1997. Invited talk at Computer Hardware Description Languages 1997, Toledo, Spain, on Verifying I/O Systems.
10. David L. Dill, Seungjoon Park, and Andreas Nowatzky. Formal specification of abstract memory models. In Gaetano Borriello and Carl Ebeling, editors, *Research on Integrated Systems*, pages 38–52. MIT Press, 1993.
11. Rob Gerth. Introduction to sequential consistency and the lazy caching algorithm. *Distributed Computing*, 1995.
12. R. Ghughal, A. Mokkedem, R. Nalumasu, and G. Gopalakrishnan. Using “test model-checking” to verify the Runway-PA8000 memory model. In *Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 1998.
13. Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, August 1997.
14. James Gosling, Bill Joy, and Guy Steele. *The JavaTM Language Specification*. Sun Microsystems, 1.0 edition, August 1996.
15. S. Graf. Verification of a distributed cache memory by using abstractions. *Lecture Notes in Computer Science*, 818:207–220, 1994.
16. R. Hojati and R. Brayton. Automatic datapath abstraction of hardware systems. In *Conference on Computer-Aided Verification*, 1995.
17. R. Hojati, R. Mueller-Thuns, P. Loewenstein, and R. Brayton. Automatic verification of memory systems which service their requests out of order. In *CHDL*, pages 623–639, 1995.
18. Gerry Kane. *PA-RISC 2.0 Architecture*. Prentice Hall, 1996.
19. P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in TLA. *Distributed Computing*, 1997.
20. Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. Technical report, Digital Equipment Corporation, Systems Research Center, February 1993.
21. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994. Also appeared as SRC Research Report 79.
22. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.
23. A. Mokkedem. Verification of three memory systems using test model-checking. <http://www.cs.utah.edu/~mokkedem/vis/vis.html>.
24. R. Nalumasu, R. Ghughal, A. Mokkedem, and G. Gopalakrishnan. The ‘test model-checking’ approach to the verification of formal memory models of multiprocessors. Technical Report UUCS-98-008, University of Utah, 1998. also available in <http://www.cs.utah.edu/~mokkedem/frames/tr98.ps.gz>.
25. Seungjoon Park and David L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *SPAA*, pages 288–296, Padua, Italy, June 24–26, 1996.
26. Vis-1.2 release. <http://www-cad.eecs.berkeley.edu/Respep/Research/vis/>.
27. David L. Weaver and Tom Germond. *The SPARC Architecture Manual – Version 9*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.