

Using “test model-checking” to verify the Runway-PA8000 memory model *

Rajnish Ghughal, Abdel Mokkedem, Ratan Nalumasu, and Ganesh Gopalakrishnan
Department of Computer Science, University of Utah,
Salt Lake City, UT 84112-9205
Contact email: {ghughal,mokkedem}@cs.utah.edu

Abstract

We have developed a formal technique called *test model-checking* for debugging claimed conformance to formal memory models by realistic memory systems and multiprocessor machines. Test model-checking is an embedding of a formal testing method called ARCHTEST in the *model-checking* framework. In this paper, we describe our technique and illustrate it on the problem of checking sequential consistency of (a model of) the HP PA8000 symmetric multiprocessing (SMP) bus called Runway. Our experiments show that test model-checking is an effective method for use in the typically iterative design cycle of complex memory systems to quickly detect ordering violations and pinpoint their cause.

1 Introduction

The fundamentally important problem [AG96] of verifying whether a given memory system conforms to a formal memory model appears in a number of contexts, including CPU design [Col92, GK94], I/O bus design [Cor97], and even in multi-threaded language design [GJS96]. As the semantics of memory orderings are too subtle to be fathomed through informal reasoning alone, formal verification methods have a natural role to play here. Unfortunately, previous solutions to this problem have either demanded valuable human expertise and time or have required complex temporal assertions to be made. Consequently, they are unsuitable for use in the typically *iterative* design cycle of memory systems to *quickly* detect ordering violations and to *pinpoint* the source of the violations. In this paper we describe a method that meets these objectives, called *test model-checking*.

Test model-checking formally adapts to the realm

^oSupported 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.

of *model-checking* a formal architectural testing method called ARCHTEST that has previously been successfully used on a number of commercial multiprocessors. While ARCHTEST is an *incomplete* testing method in that it does not, under all circumstances, detect violations of memory orderings [Col92], its tests have been shown to be sufficiently incisive to be useful in practice [Col]. 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 [PD96] 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 [Ger95], a simple memory system, expressed in various temporal logics (by [Gra94] in $\forall\text{CTL}^*$ [CES86] and [LLOR97] in TLA [Lam94]) are quite complex. In [Gra94], abstract interpretation [CC77] is employed to reduce infinite-system verification to finite $\forall\text{CTL}^*$ model-checking. They apply this technique to verify the sequential consistency of lazy caching with unbounded queues. They recognize that to get an exact characterization of sequential consistency involving only the observable event names, one needs full second order logic [Gra94]. To be able to express sequential consistency in $\forall\text{CTL}^*$, they give a stronger characterization of sequential consistency, the temporal formula describing which is very complex. We do not believe that such descriptions 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* [BCS96, GGH⁺97] used by Hewlett-Packard in their high-end machines, but also it discovered many subtle bugs in early Utah Runway Models (URMs) describing this bus that *we created*. Our 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.

In [McM93], a CTL property that encodes a test for memory orderings has been discussed under the section heading ‘Sequential Consistency’. To give a historic perspective, the test model-checking idea originated in our attempt to answer the question of which memory ordering rule(s) the test of [McM93] is really verifying. We still have not found a satisfactory answer because the test of [McM93] 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 [Col92]. In this context, our contribution has been to not only answer the

above question (see Figure 1 which captures the test of [McM93]), but also to describe formally the finite-state abstractions that justify porting other tests from [Col92] to a model-checking framework.

Alur et al [AMP96] showed that the problem of finding if there is a sequentially consistent string σ in a regular expression r is undecidable. In the model they consider r is made up of instructions $read(p, x, v)$ and $write(p, x, v)$ where p is the process that issued the instruction, x is the shared variable, and v is the value the read instruction returns or the value to write. 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 2.1.

In [PD96], 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 manually assisted theorem proving. The work in [HMTLB95] as well as [DPN93] 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. In [GK97, GK94], the authors study the problem of deciding whether a given set of traces are sequentially consistent. These works do not address the problem we study, namely that of establishing that a detailed memory system model written in an HDL conforms to a formal memory model.

Test model-checking has a number of desirable features. It involves model-checking a *fixed* set of safety properties for each formal memory model, that are very nearly independent of the actual memory system model being tested. This greatly facilitates its use in the typically *iterative* design cycle where debugging is most effective, design changes are frequent, and time-consuming alterations to the properties being verified following design changes would be frowned upon. The formal adaptation of the tests of ARCHTEST made in test model-checking can be verified once and for all, thanks to the fixed set of tests used in test model-checking (we describe and argue the correctness of these abstractions later). Finally, in test model-checking, a memory model is viewed as a collection of simpler ordering rules, and for each constituent ordering rule, a specific property is tested on the memory system. We found that this significantly helps *compartmentalize* errors, as opposed to producing non-intuitive error traces that could result during conventional model-checking, which can be very difficult to understand for non-trivial memory systems.

Test model-checking is also a more effective debugger for memory models than ARCHTEST in a formal sense. The tests of ARCHTEST are straight-line programs of length k , one per node. Such programs execute on various nodes of the multiprocessor concurrently. The recommendation accompanying ARCHTEST is that users run the tests for as large a k that is feasible, because then the chances of being scheduled according to different interleavings (by the underlying operating system, memory controller arbiter, etc.) increase. In adapting the tests of ARCHTEST, test model-checking gives the effect of choosing $k = \infty$. Thus, we cover *all possible schedules*.

2 Test Model-checking

ARCHTEST is based on the theory presented in [Col92] 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. For read operations there is one read event per each read operation. However, for write operations, there is one write event per process per write operation which captures the effect of a write operation becoming visible to different processors at different times. Five important 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. Some of the orderings imposed by this rule include: (i) for statements such as $A := B$, read event for operand B happens before all write events for operand A and the values of write events for operand A match the value of read event for operand B, and (ii) if a process P_1 sees a value written for some operand A by a process P_2 then P_2 's write event for operand A (in P_1 's store) happens before the read event for operand A by process P_1 and their values match. 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): For any pair of write events a and b in the same process, if a comes before b in program order then a happens before b .

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 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 .

2.1 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* [HB95], *i.e.*, the control logic can at most compare two addresses for equality or inequality

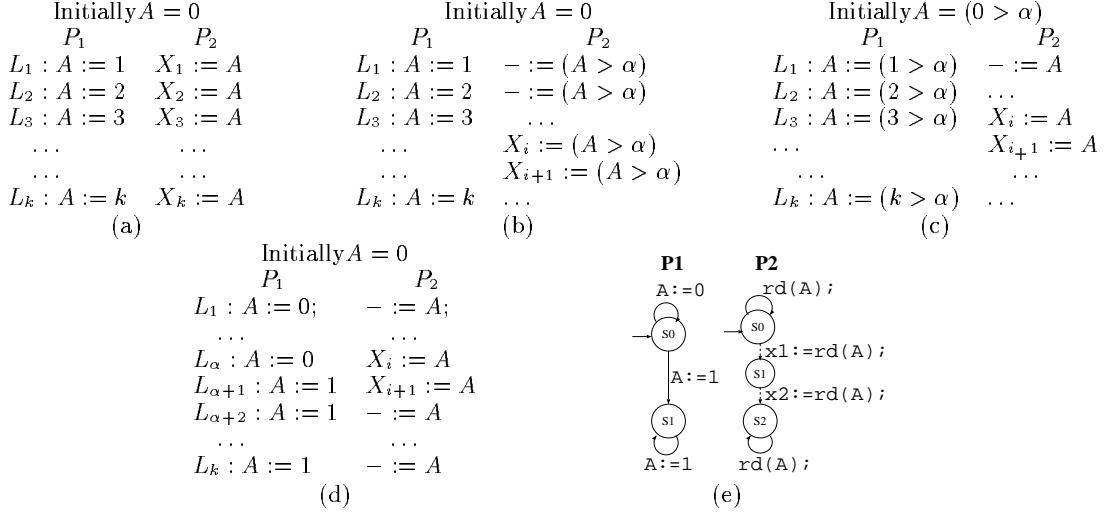


Figure 1: $Test_{ROWO}$: A program to test for $A(CMP, RO, WO)$ and its abstraction

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*, as will be discussed shortly.

2.2 Testing for architecture rules

The test $Test_{ROWO}$ 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). 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 holds:

CONDITION 1 (MONOTONIC) The sequence of X values is monotonically increasing, *i.e.*,

$$\forall i : 1 \leq i < k : X_i \leq X_{i+1}.$$

ARCHTEST recommends that the parameter k be made as large as possible to increase the chances that real machines running programs P_1 and P_2 interleave them in more ways. In test model-checking, we obtain the effect of $k = \infty$ by abstracting $Test_{ROWO}$ using *data independence*, as follows. Assume that Condition 1 is violated, *i.e.*,

$$\begin{aligned} & \exists i : 1 \leq i < k : X_i > X_{i+1} \\ \iff & \exists i, \alpha : 1 \leq i < k : (X_i > \alpha) \wedge \\ & (X_{i+1} \leq \alpha) \\ (*) \iff & \exists i, \alpha : 1 \leq i < k : (X_i > \alpha) \wedge \\ & \neg(X_{i+1} > \alpha) \end{aligned}$$

Since (*) depends only on the predicates $X_i > \alpha$ and $X_{i+1} > \alpha$, we can rewrite the test as shown in Figure 1(b), and rewrite (*) as $X_i = 1 \wedge X_{i+1} = 0$. In the figure we show “ $- := (A > \alpha)$ ” to show that the statement is executed only for obtaining the side effects associated with reading A . Since, in this figure, all reads of A occur in $(A > \alpha)$, we can replace the statements $A := v$ and $\bullet := (A > \alpha)$ with $A := (v > \alpha)$ and $\bullet := A$ respectively, without affecting the correctness of the property. The effect of such a replacement is shown in Figure 1(c).

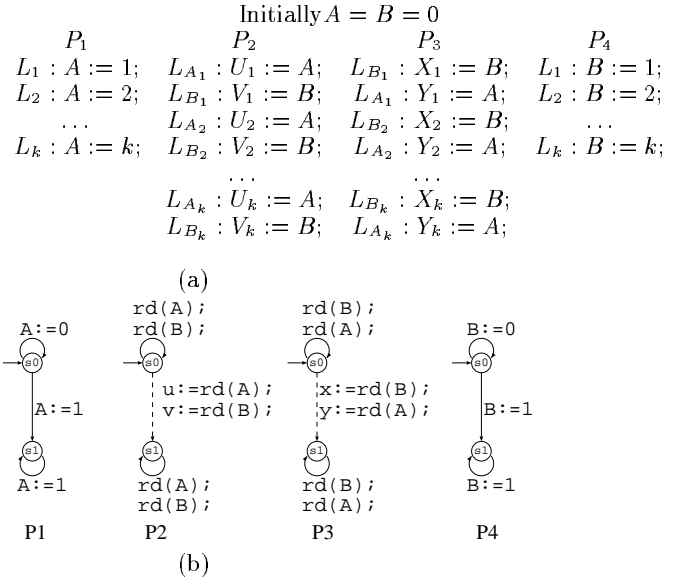


Figure 2: $Test_{WA}$: ARCHTEST test and test automata for $A(CMP, RO, WO, WA)$

Figure 1(d) is essentially Figure 1(c) where every $v > \alpha$ is replaced by 0 or 1 as the case might be. Figure 1(e) is the *test automata* obtained from Figure 1(d) by making P_1 and P_2 guess the values of α and i nondeterministically. MONOTONIC is checked as the *memory rule safety property*: P_2 at $S2 \Rightarrow x2 > x1$. The dotted edges in test automata indicate when the values used in memory rule safety property are assigned.

The test $Test_{WA}$ of ARCHTEST for $A(CMP, RO, WO, WA)$ is shown in Figure 2(a). When this program is run on a machine that obeys $A(CMP, RO, WO, WA)$, MONOTONIC must hold for arrays U, V, X and Y , and *in addition*, condition ATOMIC must hold:

CONDITION 2 (ATOMIC) $\forall i, j : 1 \leq i, j \leq k : V_i \geq X_j \vee Y_j \geq U_i.$

Initially $A = B = 0$

P_1	P_2
$L_{11} : A := 1;$	$L_{11} : B := 1;$
$L_{12} : Y_1 := B;$	$L_{12} : X_1 := A;$
$L_{21} : A := 2;$	$L_{21} : B := 2;$
$L_{22} : Y_2 := B;$	$L_{22} : X_2 := A;$
...	...
$L_{k1} : A := k;$	$L_{k1} : B := k;$
$L_{k1} : Y_k := B;$	$L_{k1} : X_k := A$

(a)

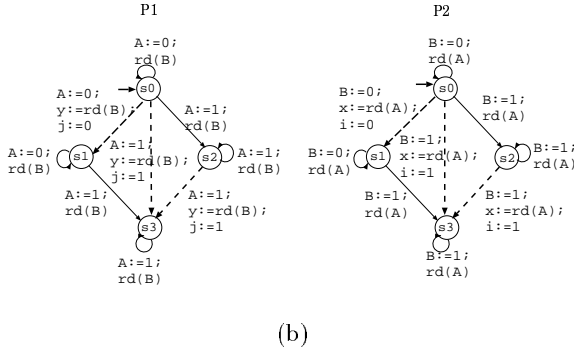


Figure 3: $Test_{PO}$: ARCHTEST test and test automata for $A(CMP, PO)$

ATOMIC watches for the possibility that a write of P_1 and a write of P_4 appear to have completed in different orders to P_2 and P_3 . Figure 2(b) shows the corresponding test automata which checks for violation of ATOMIC.

The memory rule safety property corresponding to condition ATOMIC is: P_2 and P_3 in their final states $\Rightarrow v \geq x \vee y \geq u$. (We also test for violation of MONOTONIC using test automata (not shown here) similar to one shown in Figure 1(e) using two variables for each of the arrays U, V, X and Y .)

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

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

Similar to $Test_{ROWO}$, assuming *data-independence*, we have an execution of the test automata (Figure 2(b)) in which P_1, P_2, P_3, P_4 iterate 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 $u = 1 \wedge v = 0 \wedge x = 1 \wedge y = 0$.

The test $Test_{PO}$ of ARCHTEST for $A(CMP, PO)$ is shown in Figure 3(a). When this program is run on a machine obeying $A(CMP, PO)$, in addition to MONOTONIC for arrays X and Y , condition PO_CROSS also must hold:

CONDITION 3 (PO_CROSS) $\forall i, j : 1 \leq i, j \leq k : (X_i \geq j \vee Y_j \geq i) \wedge (X_i \leq j \vee Y_j \leq i)$.

Figure 3(b) shows the corresponding test automata. The memory rule safety property corresponding to condition PO_CROSS is: P_1 and P_2 in their final states $\Rightarrow (x \geq j \vee y \geq i) \wedge (x \leq j \vee y \leq i)$.

To show that this abstraction preserves PO_CROSS , let PO_CROSS be violated in ARCHTEST test $Test_{PO}$.

$$\begin{aligned} & \exists i, j : (X_i < j \wedge Y_j < i) \vee \\ & (X_i > j \wedge Y_j > i) \\ \iff & \exists i, j, \alpha, \beta : ((X_i = \alpha) \wedge (j > \alpha) \wedge \\ & (Y_j = \beta) \wedge (i > \beta)) \vee \\ & ((X_i > \alpha) \wedge (j = \alpha) \wedge \\ & (Y_j > \beta) \wedge (i = \beta)) \end{aligned}$$

Similar to the case of $Test_{WA}$, if $\exists i, j : X[i] < j \wedge Y[j] < i$, then we can get a case in the test automata where $x = 0 \wedge j = 1 \wedge y = 0 \wedge i = 1$. Similarly, if $\exists i, j : X[i] > 0 \wedge Y[j] > i$, then we can get a case in the test automata where $x = 1 \wedge j = 0 \wedge y = 1 \wedge i = 0$. Hence, the memory rule safety property corresponding to PO_CROSS will be violated in test automata if and only if PO_CROSS will be violated in ARCHTEST test $Test_{PO}$ for $k = \infty$.

Summarizing test model-checking, there cannot be any real guarantees for ARCHTEST tests such as $Test_{WA}$, $Test_{PO}$ etc. 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 . Test model-checking effectively run ARCHTEST tests for $k = \infty$ by transforming each ARCHTEST test into a test automata using non-determinism. Also, the model-checking framework guarantees that we explore all possible interleavings rather than only particular interleavings.

3 Runway-PA8000 Memory System

Figure 4 shows a simplified view of 2 HP PA8000 CPUs and a memory controller (HOST) interconnected by HP Runway Bus[BCS96, Cam97, Kan96]¹. We will describe the Runway-PA8000 system in some detail to facilitate a clear description of some of the subtle bugs in URM unearthed by the test model-checking technique. Runway is a synchronous, split-transaction bus which is responsible for providing a coherent view of shared memory to the processors (*clients*) while still allowing the clients to maintain private copies of memory lines in their caches. Cache Coherency is maintained by a snoopy coherency protocol described below.

Snoopy Coherency Protocol

Each cache line in a client can be in one of the four states²: invalid, shared, private-clean or dirty. If a

¹We have purposefully avoided arbitration lines and other details for the sake of clarity. The actual Runway allows up to four CPUs and one I/O processor and also many more transactions including coherent, non-coherent and I/O transactions than we describe here. We provide a simplified view of its operation which captures the essential complexity of its behavior.

²There are also transient states that the cache line may assume when it is changing from one of these clean states to another.

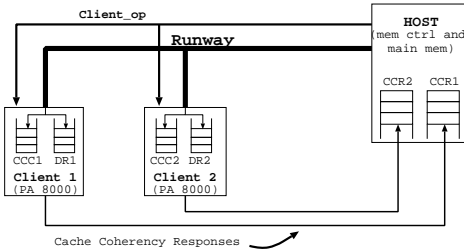


Figure 4: Simplified View of Runway-PA8000 Memory System

Transaction	Generated by	State	<i>ccr</i>
-	self	-	<i>coh_ok</i>
-	other	invalid	<i>coh_ok</i>
<i>rsp</i>	other	private-clean	<i>coh_shared</i>
<i>rsp</i>	other	shared	<i>coh_shared</i>
<i>rp</i>	other	shared	<i>coh_ok</i>
<i>rp</i>	other	private-clean	<i>coh_ok</i>
-	other	dirty	<i>coh_copyout</i>

Figure 5: *ccr* generated when a transaction gets to the head of CCC queue

client suffers a read miss in cache, it generates a *rsp* (read shared or private) transaction; if it suffers a write miss, it generates a *rp* (read private) transaction. The transaction is broadcast on the Runway when it wins the bus mastership. All clients snoop the transaction into their CCC (cache coherency check) queues and process the entries in CCC queue at their own speed. When a transaction gets to the head of CCC of client C_i , it sends a *ccr* (cache coherency response) to HOST according to Figure 5, and also changes its state to reflect the transaction; for example, if the transaction is *rp* generated by C_i , it would assume “invalid-private-clean” transient state. If a client generates a *coh_copyout* as *ccr*, it would later issue a *c2cw* (cache to cache write) to supply the data. HOST enters the *ccr*'s into its CCR queue, and after all clients have responded to a transaction, the HOST determines if the data would be supplied by another client. If no client is going to supply the data, the HOST would generate a *hdr* (host data return) transaction on the Runway to supply the data to the requester. It would also drive Client_op lines to indicate whether the data must be shared (i.e., at least one of the *ccr*s is *coh_shared*). When a client notices a data return (a *hdr* or *c2cw*) targeted towards it, it enters the information into data return (DR) queue. Note that a client might receive a data return before it generates the corresponding *ccr*. In this case, the client keeps the data in data return queue until the *ccr* is sent out.

Delay in *ccr* generation

If a client has a *c2cw* transaction for a line yet to go on Runway, then it delays generating any more *ccr*'s for that line. To see why this is necessary, consider the following. Suppose a client C1 has a dirty line. Client C2 requests this line by issuing *rsp* transaction on bus. C1 will generate *coh_copyout* in response to C2's request, invalidate its own line, and create a *c2cw* transaction for C2. Note that the most recent data for this line is with C1 and

not HOST. Now, a client C3 requests the same line by issuing *rsp*. C2 and C3 generate respectively *coh_shared* and *coh_ok* *ccr*s in response to C3's request. C1's *ccr* will be *coh_ok* in response to C3's request. If C1 sends *coh_ok* to HOST before its *c2cw* goes on the bus then HOST can provide stale data to C3 by its *hdr* transaction. To avoid this, C1 delays generating *ccr* until the *c2cw* goes on the bus.

Arbitration

Runway follows a complex pipelined arbitration algorithm to determine the bus master. Here, we only present an approximation of the algorithm. Every bus user (client or HOST) must become the bus master before it can drive the bus. Bus mastership at cycle $N+2$ is acquired by initiating the arbitration in cycle N by driving the request through dedicated arbitration lines (not shown in the figure). During cycle $N+1$, every potential bus user evaluates the others' drives and, in conjunction with round-robin pointers for arbitration priorities, determines who wins bus mastership for cycle $N+2$. Those who do not win bus mastership keep off the bus. Bus arbitration proceeds in a pipelined manner concurrently with transaction processing.

PA8000 Runway interface

In addition to the Runway specifics described above, PA8000 Runway interface (PARI) also adheres to the following constraints in order to ensure Program Order and Write Atomicity [BCS96, Kan96]. PARI allows a client to initiate Runway transactions for various cache misses; it is possible that these transactions complete out of order. However, all instructions strictly *complete* in program order. PARI guarantees that the client will stall the coherency response for any cache line which it has an outstanding miss for (i.e., it has initiated a Runway transaction, has assumed the ownership but is still waiting for the data). The coherency response will be generated only after the client has received the data and has used it to make forward progress at least one instruction. PARI guarantees that if a client receives data for its Runway transaction before it assumed the ownership, then it will not modify or use the data until it processes its own transaction (and thus assumes ownership). PARI guarantees that if a client has a *c2cw* transaction enabled to go to Runway then it gets the highest priority to go to the Runway.

4 Verification using Test Model-checking

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 [Ver] for more information). These three 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 [Mok] or by contacting the authors.

4.1 How do we check for sequential consistency?

A sequentially consistent memory system [Lam93] requires that there be a single self-consistent trace t of

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

Figure 6: Serial memory transaction rules

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 [Col92], one’s intuition about sequential consistency matches the behavior described by $A(CMP, PO, WA)$.

As [Col92] 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 [Col92]). 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 8.

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 SC operationally. The **lazy caching** protocol [ABM93, Ger95], shown in Figure 7, 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 .

Event	Allowed if	Action
$R_i(d, a)$	$C_i(a) = d \wedge Out_i = \{\}$ \wedge no $*-ed$ entries in In_i	
$W_i(d, a)$		$Out_i := append(Out_i, (d, a))$
$MW_i(d, a)$	$head(Out_i) = (d, a)$	$Mem[a] := d;$ $Out_i := tail(Out_i);$ $(\forall k \neq i :: In_k :=$ $append(In_k, (d, a)));$ $In_i := append(In_i, (d, a, *))$
$MR_i(d, a)$	$Mem[a] = d$	$In_i := append(In_i, (d, a))$
$CU_i(d, a)$	$head(In_i)$ is either (d, a) or $(d, a, *)$	$In_i := tail(In_i);$ $C_i := update(C_i, d, a)$
CI_i		$C_i := restrict(C_i)$
Initially:	$\forall a Mem[a] = 0$ $\wedge \forall i = 1 \dots n C_i \subset Mem \wedge In_i = \{\} \wedge Out_i = \{\}$	
Fairness:	no action other than CI_i can be always enabled but never taken	
W—write	MW—memory write	CU—cache update
R—read	MR—memory read	CI—cache invalidate

Figure 7: Gerth’s version of the lazy caching algorithm, from Figure 4 of [Ger95].

that corresponds to writes of P_i . For this reason, such cache update requests are marked (with a $*$).

4.3 URM in VIS Verilog

We constructed a Verilog Utah Runway Model (URM) and the two abstractions of $Test_{PO}$ and $Test_{WA}$ to verify that its memory model is sequentially consistent. The complexity of the system stems from a number of sources: (a) multiple outstanding transactions for each processor, (b) out-of-order completion of the Runway transactions, but in-order completion of instructions, (c) eager assumption of ownership without receiving the corresponding data, (d) “equivalent” states introduced by decoupled execution due to coherency queues, (e) speculative execution features of the processor to ensure performance in spite of in-order completion of the instructions, (f) an involved distributed pipelined arbitration algorithm. We did not try to model each of these features in their full glory, but 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 (see [Mok]). For instance all essential features of (a), (b), (c), and (e) are included, (f) is abstracted by using nondeterminism. (d) is abstracted as explained below.

Abstraction of Queues Additional abstraction effort was necessary to make our URM digestible by VIS. This essentially consists in getting rid of the CCC, CCR, and DR queues which are the main cause of state explosion, but retain HDR queue in the HOST and C2CW queues in the HOST and clients.

In Runway, most of the conflicts are detected and resolved by the HOST. There is one situation where a client detects conflict: the client has a pending $c2cw$ transaction. The client resolves this by delaying its coherency response; the net result of this delay is that the HOST would not generate hdr transactions until the $c2cw$ goes on the Runway. Since we abstracted away the CCR queues, in our URM the clients send the coherency re-

sponse for a coherent transaction immediately after its occurrence on the bus. Hence, in our URM the clients cannot resolve conflicts by *delaying* the coherency response; instead the HOST *computes* if the coherency response needs to be delayed, and if so, delays the *hdrs* appropriately. This is achieved as follows. A counter is associated with each HDR queue entry. If the counter is non-zero, then it is waiting for some *c2cw* transactions for that line from the clients, hence the *hdr* needs to be delayed. After all the pending *c2cw* transactions for that line go on the bus, the counter becomes zero, and hence the *hdr* transaction can go on the bus. In our URM, we used a two-bit counter, which allows up to four processors.

In Runway, all clients save the data returns (*hdr* and *c2cw* transactions) in DR queue until the corresponding request appears at the head of its CCC queue. This is necessary to enforce in-order completion of instructions. We abstract away the CCC queues and the data return queues by associating one bit of information with each cache line in each client. This bit is set for an address *a* whenever a data return happens for *a*, but a preceding instruction is not yet completed. After all preceding instructions are completed, the data is used, and the bit is reset indicating the completion of the instruction.

4.4 Verification results

The tables in figure 8 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)

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

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

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 URM 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 [Col92], 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.

Description of a Bug found in preliminary URM:

Similarly, another corner-case bug was caught by *test model-checking* in our URM by a violation of PO_CROSS condition using $Test_{PO}$. This bug generated a long counterexample trace, due to the depth of the sequential logic of the model. The trace revealed the following situation:

- (1) $client_i$ has removed its own read transaction from the bus, then
- (2) $client_i$ sends *coh_ok* in response to a subsequent coherent transaction for the same line before getting the data for its transaction (by *hdr* or *c2cw*).

This problem was fixed using the counter in the HOST's HDR entries to record the pending *c2cws* and the one-bit information in the client's cache lines to record whether the data is supplied, as explained in paragraph 4.3. After fixing the bug the PO condition passed.

5 Discussion and Future Plans

Using three memory systems, one of which very complex, we have evaluated a new approach to verify multi-processor machines 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. In effect, test automata offer good specifications to check for formal memory models such as sequential consistency.

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

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

Figure 8: Verification results using VIS on a SPARC ULTRA-1 with 512 MB Memory

RMO [AG96] that are described in the SPARC V9 architecture manual [WG94]. 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 for providing ARCHTEST under a no-fee, research-only license. We would like to thank Dr. Paliath Narendran for many fruitful discussions. We would like to thank Dr. Al Davis and his Avalanche team for offering us the unique opportunity to work on state-of-the-art processors and busses.

References

- [ABM93] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.
- [AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.
- [AMP96] 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, New Brunswick, New Jersey, July 1996.
- [BCS96] 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.
- [Cam97] 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.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of 4th POPL*, pages 238–252, Los Angeles, CA, ACM Press, 1977.
- [CES86] 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.
- [Col] W. W. Collier. Multiprocessor diagnostics. <http://www.infomall.org/diagnostics/archtest.html>.
- [Col92] W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [Cor97] Francisco Corella, April 1997. Invited talk at Computer Hardware Description Languages 1997, Toledo, Spain, on Verifying I/O Systems.
- [DPN93] 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.
- [Ger95] Rob Gerth. Introduction to sequential consistency and the lazy caching algorithm. *Distributed Computing*, 1995. Also can be found in <http://www.research.digital.com/SRC/tla/papers.html#Lazy>.

- [GGH⁺97] G. Gopalakrishnan, R. Ghughal, R. Hosabettu, A. Mokkedem, and R. Nalumasu. Formal modeling and validation applied to a commercial coherent bus: A case study. In Hon F. Li and David K. Probst, editors, *CHARME*, Montreal, Canada, 1997.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The JavaTM Language Specification*. Sun Microsystems, 1.0 edition, August 1996. appeared also as book with same title in Addison-Wesleys 'The Java Series'.
- [GK94] Phillip B. Gibbons and Ephraim Korach. On testing cache-coherent shared memories. In *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, pages 177–188, New York, NY, USA, June 1994. ACM Press.
- [GK97] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, August 1997.
- [Gra94] S. Graf. Verification of a distributed cache memory by using abstractions. *Lecture Notes in Computer Science*, 818:207–??, 1994.
- [HB95] R. Hojati and R. Brayton. Automatic datapath abstraction of hardware systems. In *Conference on Computer-Aided Verification*, 1995.
- [HMTLB95] 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.
- [Kan96] Gerry Kane. *PA-RISC 2.0 Architecture*. Prentice Hall, 1996. ISBN 0-13-182734-0.
- [Lam93] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. Technical report, Digital Equipment Corporation, Systems Research Center, February 1993.
- [Lam94] 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.
- [LLOR97] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in tla. *Distributed Computing*, 1997.
- [McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.
- [Mok] A. Mokkedem. Verification of three memory systems using test model-checking. <http://www.cs.utah.edu/~mokkedem/vis/vis.html>.
- [PD96] 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.
- [Ver] Vis-1.2 release. <http://www-cad.eecs.berkeley.edu/Respep/Research/vis/index.html>.
- [WG94] David L. Weaver and Tom Germond. *The SPARC Architecture Manual – Version 9*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.