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

Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem and Ganesh Gopalakrishnan

UUCS-98-008

Department of Computer Science, University of Utah,
Salt Lake City, UT 84112-9205
Contact email: {ratan, ganesh}@cs.utah.edu

This technical report combines work reported in CAV 98 and SPAA 98.

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.

Keywords: Formal memory models, shared memory multiprocessors, formal testing, model-checking.

1 Introduction

The fundamentally important problem [AG96] 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 [Cor97]; even language designers of multi-threaded languages such as Java that support shared updates [GJS96] 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.*

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 [Col] by running a suite of test-programs on them. ARCHTEST is an *incomplete*

*Supported in part by ARPA Order #B990 under SPAWAR Contract #N0039-95-C-0018 (Avalanche), DARPA under contract #DABT6396C0094 (UV).

testing method in that it does not, under all circumstances, detect violations of memory orderings [Col92]. Nevertheless, its tests have been shown to be incisive in practice [Col]. Most importantly, the formal theory of memory ordering rules developed by Collier in [Col92] 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 [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 $\forall$CTL* [CES86] and [LLOR97] in TLA [Lam94]) 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* [BCS96, GGH$^+$97] used by Hewlett-Packard in their high-end machines, but also it discovered many subtle bugs in our early Utah Runway Model (URM) 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. 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.

Test model-checking has a number of other 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 fixed nature greatly facilitates the use of test model-checking *within the 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 (test model-checking will not need such alterations). Also, 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*. The subtle bugs detected by test model-checking on realistic examples that are reported in Section 5 corroborate our intuition that test model-checking is indeed an effective debugging tool for memory models.

To reiterate, 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.

$$
\begin{array}{ll}
(C1) & \forall (a,d) \in \mathit{address} \times \mathit{datum}\ \forall i \in \mathit{index} : \mathit{init} \implies \mathbf{AG}(\mathit{enable}(\mathit{read}_i(a,d)) \implies \mathit{avail}_i(a,d)) \\
(C2) & \forall (a,d),(a,d') \in \mathit{address} \times \mathit{datum}.\ d \neq d'\ \forall i \in \mathit{index} : \\
 & \mathit{init} \implies \mathbf{AG}((\mathit{avail}_i(a,d) \wedge \mathbf{EF}(\mathit{enable}(\mathit{read}_i(a,d)))) \implies \mathbf{A}[\neg \mathit{avail}_i(a,d)\ \mathbf{W}\ \mathbf{AG}(\neg \mathit{avail}_i(a,d))]) \\
(C3) & \forall (a,d) \in \mathit{address} \times \mathit{datum}\ \forall i,k \in \mathit{index} : \mathit{init} \implies \mathbf{AG}[\mathit{after}(\mathit{write}_k(a,d)) \implies \mathbf{AF}(\mathit{avail}_i(a,d))] \\
(S1) & \forall (a,d) \in \mathit{address} \times \mathit{datum}\ \forall i \in \mathit{index} : \\
 & \quad \mathit{init} \implies \mathbf{AG}[\mathit{after}(\mathit{write}_i(a,d)) \implies \mathbf{A}(\neg \mathit{enabled}(\mathit{read}_i(a,d))\ \mathbf{W}\ \mathit{avail}_i(a,d))] \\
 & \vdots \\
(S4) & \forall (a,d),(a,d') \in \mathit{address} \times \mathit{datum}.\ d \neq d'\ \forall i,k \in \mathit{index} : \\
 & \quad \mathit{init} \implies \mathbf{A}([\neg \mathit{avail}_i(a,d)\ \mathbf{W}\ (\mathit{avail}_i(a',d') \wedge \neg \mathit{avail}_i(a,d))] \implies [\neg \mathit{avail}_k(a,d)\ \mathbf{W}\ \mathit{avail}_k(a',d')])
\end{array}
$$

Figure 1: Part of the specification of Sequential Consistency, from [Gra94]

Related Work

In [Gra94], abstract interpretation [CC77] is employed to reduce infinite-system verification to finite $\forall$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 $\forall$CTL*, they give a stronger characterization of sequential consistency. For this stronger characterization, the expression of sequential consistency is very complex, as shown in figure 1 (this figure shows only part of their sequential consistency expression). A technique very similar to test model-checking was proposed in [McM93] 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 [McM93] 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 [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]. One of our contributions is that we answer these questions by elaborating on the theoretical as well as practical aspects of test model-checking.

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 manual 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. They do not address the problem of establishing the memory models provided by detailed memory subsystem designs, which is our contribution. In [GK97, GK94], the authors analyze the problem of deciding whether a given set of traces are sequentially consistent. Our approach differs in two respects. First, we are interested in proving that detailed models of memory systems are correct, while they obtain traces (presumably from actual machines) and analyze them for sequential consistency. Second, our method is more useful for CPU designers as it can give feedback during early phases of the design pinpointing which ordering rules are violated (if any).

2 Overview of ARCHTEST

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

$$\begin{array}{cc} \multicolumn{2}{c}{\text{Initially} A = 0} \\ Process P_1 & Process P_2 \\ L_1 : A := 1; & X[1] := A; \\ L_2 : A := 2; & X[2] := A; \\ L_3 : A := 3; & X[3] := A; \\ \dots & \dots \\ L_k : A := k & X[k] := A; \end{array}$$

Figure 2: $Test_{\mathbf{ROWO}}$: ARCHTEST test for $A(CMP, RO, WO)$

tantamount to obeying *all* the constituent elemental rules; violating a compound rule is tantamount to violating *any* of the constituent elemental rules. Each such elemental rule describes a constraint on the order in which various read and write events can occur. For read operations there is one read event per each read operations. 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. 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 operand. 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 than a happens before b. Event a or b can be either read or write event. So, both RO and PO 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 is no i, j and no event e such that e is before W_i and is after W_j.

In order to check memory subsystems for a compound rule, ARCHTEST provides a test for each compound rule along with a set of conditions to be checked for. If any of the conditions is violated then a violation to obey the compound rule is detected.

$Test_{\mathbf{ROWO}}$: 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 2. 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.*:
$\forall i, j : 1 \le i \le j \le k : X[i] \le X[j]$ or equivalently $\forall i : 1 \le i \le k - 1 : X[i] \le X[i+1]$.

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

Initially $A = B = 0$

P_1	P_2	P_3	P_4
$L_1 : A := 1;$	$L_{A_1} : U[1] := A;$	$L_{B_1} : X[1] := B;$	$L_1 : B := 1;$
$L_2 : A := 2;$	$L_{B_1} : V[1] := B;$	$L_{A_1} : Y[1] := A;$	$L_2 : B := 2;$
$\ldots$	$L_{A_2} : U[2] := A;$	$L_{B_2} : X[2] := B;$	$\ldots$
$L_k : A := k;$	$L_{B_2} : V[2] := B;$	$L_{A_2} : Y[1] := A;$	$L_k : B := k;$
	$\ldots$	$\ldots$	
	$L_{A_k} : U[k] := A;$	$L_{B_k} : X[k] := B;$	
	$L_{B_k} : V[k] := B;$	$L_{A_k} : Y[k] := A;$	

Figure 3: $Test_{\mathrm{WA}}$: ARCHTEST test for $A(CMP, RO, WO, WA)$

Initially $A = B = 0$

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

Figure 4: $Test_{\mathrm{PO}}$: ARCHTEST test for $A(CMP, PO)$

$Test_{\mathbf{WA}}$: ARCHTEST **test for** $A(CMP, RO, WO, WA)$

$Test_{\mathrm{WA}}$, shown in Figure 3 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:

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

The ATOMIC condition watches for the possibility that a write operation from P_1 and a write operation from P_4 appear to have finished in different orders to P_2 and P_3.

$Test_{\mathbf{PO}}$: ARCHTEST **test for** $A(CMP, PO)$

$Test_{\mathrm{PO}}$, shown in Figure 4 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 $Test_{\mathrm{WA}}$, $Test_{\mathrm{PO}}$ etc. are meant to be run on real machines and there can't be any real guarantees that the particular interleavings that reveal violations (such as for memory ordering rule WA watched by condition ATOMIC in $Test_{\mathrm{WA}}$) 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 $k = \infty$. Test model-checking achieves this by transforming each ARCHTEST test into a test automata which exploits non-determinism to effectively check for $k = \infty$. Also, the model-checking framework guarantees that we explore all possible interleavings than a particular interleaving.

3 Test model-checking

Test model-checking converts the tests of ARCHTEST to corresponding *memory rule test automata* ("test automata") that drive model of the memory system being examined. In our experiments, we use the Verilog language supported by VIS [Ver] to capture the memory system models as well as the test automata. 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. The reader may take a peek at Section 4.1 to know which compound rules define sequential consistency [Lam79]. In the remainder of this section, we explain the assumptions under which we formally derive *test automata* as well as *memory rule safety properties*, followed by a description of how test automata as well as memory rule safety properties are derived for specific cases.

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* [HB95], *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 5, we obtain test automata for various memory models by finitely abstracting the data used in test of ARCHTEST, using non-determinism to justify the abstraction. For example, we abstract the specific activities of process P_1 of Figure 2 into that of (non-deterministically) writing *all possible* ascending values over $\{0,1\}$, as shown in P_1 of Figure 5. 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, i.e. 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 2 can be viewed as repeating the initialization and then repeating the instruction at label L_1 of P_1 of Figure 2. Our proof sketches are illustrated on the two tests presented in Section 2 and another test described in this section.

3.3 Abstracting $Test_{\mathbf{ROWO}}$

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

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

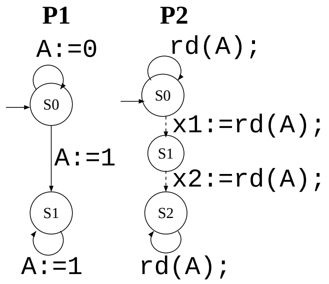

Figure 5: $Test_{\text{ROWO}}$ test automata : Test automata for $A(CMP, RO, WO)$

Initially $A = 0$

$Process P_1$	$Process P_2$
$L_1 : A := 1;$	$X[1] := (A > \alpha);$
$L_2 : A := 2;$	$X[2] := (A > \alpha);$
$L_3 : A := 3;$	$X[3] := (A > \alpha);$
$\ldots$	$\ldots$
$L_k : A := k$	$X[k] := (A > \alpha);$

(a)

Initially $A = (0 > \alpha)$

$Process P_1$	$Process P_2$
$L_1 : A := (1 > \alpha);$	$X[1] := A;$
$L_2 : A := (2 > \alpha);$	$X[2] := A;$
$L_3 : A := (3 > \alpha);$	$X[3] := A;$
$\ldots$	$\ldots$
$L_k : A := (k > \alpha)$	$X[k] := A;$

(b)

Initially $A = 0$

$Process P_1$	$Process P_2$
$L_1 : A := 0;$	$X[1] := A;$
$\ldots$	$\ldots$
$L_\alpha : A := 0$	$X[\alpha] := A$
$L_{\alpha+1} : A := 1$	$X[\alpha+1] := A$
$L_{\alpha+2} : A := 1$	$X[\alpha+2] := A$
$\ldots$	$\ldots$
$L_k : A := 1$	$X[k] := A;$

(c)

Figure 6: Abstraction of $Test_{\text{ROWO}}$

Since, the last formula compares $X[i]$ and $X[i+1]$ only to α, we can rewrite the test program as shown in Figure 6(a) *assuming data independence*, and rewrite the last formulae as

$$\exists i : 1 \leq i < k : X[i] = 1 \wedge X[i+1] = 0$$

Note that in Figure 6(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 6(b). Figure 6(c) is obtained by simplifying Figure 6(b): each $v > \alpha$ evaluates to 0 for $v \leq \alpha$ and 1 otherwise. This figure is generalized to obtain the test automaton in Figure 2(b). Intuitively the automaton 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 $x1$ and then $x2$ (dashed edges show when these variables are recorded). Thus the test automaton's execution is identical to that in Figure 6(c) except that the test automaton gives the effect of taking k to ∞. Also notice that $x1$ and $x2$ get the values corresponding to $X[i]$ and $X[i+1]$. Also, corresponding to $X[i] = 1 \wedge X[i+1] = 0$, we have $x1 = 1 \wedge x2 = 0$. Hence the memory rule safety property corresponding to condition MONOTONIC is found violated by the test automaton exactly when $Test_{\text{ROWO}}$ 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 $Test_{\text{WA}}$

Test automaton for $Test_{\text{WA}}$ is shown in Figure 7. In this automaton P_1 and P_4 write all possible ascending sequences of $\{0, 1\}$ in A and B respectively. Each processor *independently* and *non-deterministically* decides to switch from writing 0 to writing 1. Modifications similar to those in $Test_{\text{ROWO}}$ are applied to P_2 and P_3 also, to (nondeterministically) decide which $U[i]$,$V[i]$ pair and

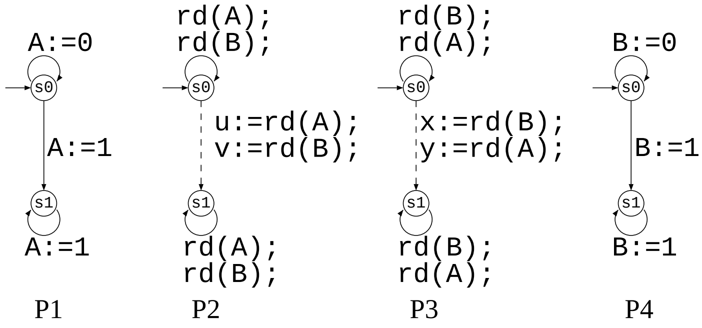

Figure 7: $Test_{\mathrm{WA}}$ test automata : Test Automata for $A(CMP, RO, WO, WA)$

$X[j]$,$Y[j]$ 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 $Test_{\mathrm{ROWO}}$, our abstraction avoids having to remember the entire extent of the arrays U, V, X, and Y. (In $Test_{\mathrm{WA}}$, one has to check for MONOTONIC also; this is done similarly to that in $Test_{\mathrm{ROWO}}$.)

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

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

Similar to $Test_{\mathrm{ROWO}}$, assuming *data-independence*, we have an execution of the test automaton (Figure 7) 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 automaton execution detects violations of ATOMIC exactly when $Test_{\mathrm{WA}}$ 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 $Test_{\mathbf{PO}}$

We now discuss a test for the elemental ordering rule Program Order (PO), which is somewhat more complex than the previous two tests. PO requires that two events of the same process occur in the order specified by the program. ARCHTEST provides the test for the compound rule $A(CMP, PO)$ shown in Figure 8. Violation of $A(CMP, PO)$ is detected if Condition 3 fails: We obtain the test automaton and the memory rule safety property for $Test_{\mathrm{PO}}$ of Figure 4 as illustrated in Figure 8. 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 followed by read instruction. It assigns the value written to A to j and the value read from B to y. Similarly, processor 2 updates i and x. The dashed edges in Figure 8 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* $\Rightarrow (x \geq j \vee y \geq i) \wedge (x \leq j \vee y \leq i)$.

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

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

Similar to the case of $Test_{\mathrm{WA}}$, 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

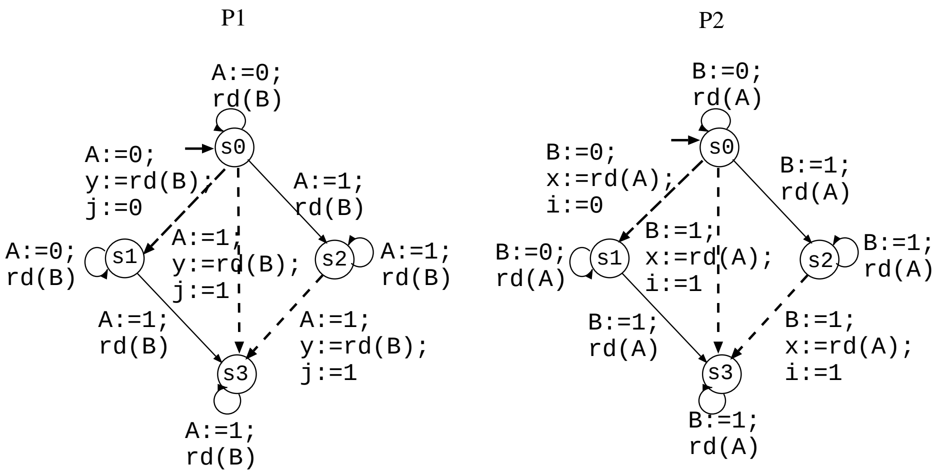

Figure 8: $Test_{\mathrm{PO}}$ test automata: Test automata for $A(CMP, PO)$

Event	Action or condition
Ri(d, a)	if Mem[a] = d
Wi(d, a)	Mem[a] := d

Figure 9: Serial memory transaction rules

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 $Test_{\mathrm{PO}}$ for $k = \infty$.

4 Case Studies

To demonstrate the effectiveness of our approach, we verified three different memory systems, namely serial memory, lazy caching, and a simplified version of the Runway bus, all using VIS [Ver]. 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 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], we can show that sequential consistency is $A(CMP, PO, WA)$.

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: $Test_{\mathrm{WA}}$ which tests for $A(CMP, RO, WO, WA)$ and $Test_{\mathrm{PO}}$ which tests for $A(CMP, PO)$. This combination is exactly equivalent to testing sequential consistency because PO implies RO and WO (as formally defined in [Col92]). For every memory system we consider, these two tests are model-checked separately and summarized in Figure 14.

Event	Allowed if	Action
$\mathsf{R}_i(d,a)$	$C_i(a) = d \wedge Out_i = \{\}$ $\wedge$ no $*$-ed entries in In_i	
$\mathsf{W}_i(d,a)$		$Out_i := append(Out_i, (d,a))$
$\mathsf{MW}_i(d,a)$	$head(Out_i) = (d,a)$	$Mem[a] := d$; $Out_i := tail(Out_i)$; $(\forall k \neq i \;::\; In_k := append(In_k, (d,a)))$; $In_i := append(In_i, (d,a,*))$
$\mathsf{MR}_i(d,a)$	$Mem[a] = d$	$In_i := append(In_i, (d,a))$
$\mathsf{CU}_i(d,a)$	$head(In_i)$ 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: $\forall a \; Mem[a] = 0$
$\wedge \forall i = 1 \ldots n \; C_i \subset Mem \wedge In_i = \{\} \wedge Out_i = \{\}$

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

4.2 Serial memory and Lazy caching

The **serial memory** protocol for n processors and a memory is shown in Figure 9. Serial memories are often used to define SC operationally. The **lazy caching** protocol [Ger95], shown in Figure 10, 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 caches 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 updates requests are marked (with a $\star$). Figure 11 shows the structure of the Verilog model we created for the memory model verification we shall discuss in section 4.5.

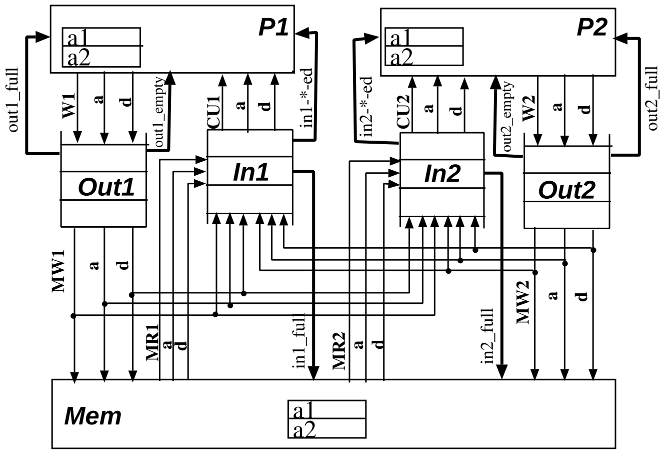

Figure 11: Verilog architecture of two processors Lazy Caching parallel machine

4.3 Runway-PA8000 Memory System

Figure 4.3.1 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.

4.3.1 Snoopy Coherency Protocol

Each cache line in a client can be in one of the four states[2]: invalid, shared, private-clean or dirty. If a 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 bust 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 13, 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

[1] We 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.

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

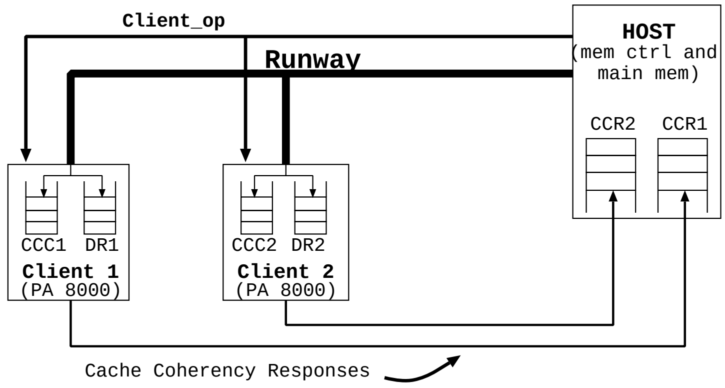

Figure 12: Simplified View of Runway-PA8000 Memory System

Transaction	Generated by	State	*ccr*
–	self	–	*coh_ok*
–	other	invalid	*coh_ok*
rsp	other	private-clean	*coh_shared*
rsp	other	shared	*coh_shared*
rp	other	shared	*coh_ok*
rp	other	private-clean	*coh_ok*
–	other	dirty	*coh_copyout*

Figure 13: *ccr* generated when a transaction gets to the head of CCC queue

the data in data return queue until the *ccr* is sent out.

4.3.2 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 generates 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 a stale data to C3 by its *hdr* transaction. To avoid this, C1 delays generating *ccr* until the *c2cw* goes on the bus.

4.3.3 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.

4.3.4 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. 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 *c2cw* transaction then it gets the highest priority to go to the Runway.

4.4 The Runway-PA8000 in VIS Verilog

We constructed a Verilog model of the Runway-PA8000 system, Utah Runway Model (URM), and the two abstractions of $Test_{\mathrm{PO}}$ and $Test_{\mathrm{WA}}$ to verify that its memory model is sequential 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 response for a coherent transaction immediately after its occurrence on the bus. Hence, in our URM the clients can't resolves conflicts by *delaying* the coherency response; instead the HOST *computes* if the coherency response needed to be delayed, and if so, delays the *hdr*s 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 a one-bit information with each cache line in each client. This bit is set for an address

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

A(CMP,WO,RO,WA)	#states	#bdd nodes	conditions verified	runtime (mn:sec)
serial memory	21242	10084	Vacuity Cond1 – Cond3	00:04 00:34
lazy caching	1.90736e+06	513655	Vacuity Cond1 – Cond3	02:02 59:33
URM	985236	1695092	Vacuity Cond1 – Cond3	17:24 40h17:33

Figure 14: Verification results using VIS on a SPARC ULTRA-1 with 512 MB Memory

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.

4.5 Verification results

The tables in figure 14 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 $Test_{\mathrm{WA}}$ and $Test_{\mathrm{PO}}$ are model-checked for the following conditions: (Figure 8 does not show some of these states)

$$
\begin{array}{lrl}
Test_{\mathrm{WA}}: & \textsc{Monotonic}: & \wedge\,(P_2.inS_2) \implies (P_2.U_1 \leq P_2.U_2) \\
 & & \wedge\,(P_2.inS_2) \implies (P_2.V_1 \leq P_2.V_2) \\
 & & \wedge\,(P_3.inS_2) \implies (P_3.X_1 \leq P_3.X_2) \\
 & & \wedge\,(P_3.inS_2) \implies (P_3.Y_1 \leq P_3.Y_2) \\
 & \textsc{Atomic}: & (P_2.inS_1 \wedge P_3.inS_1) \implies (P_2.V \geq P_3.X \vee P_3.Y \geq P_2.U) \\
 & & \\
Test_{\mathrm{PO}}: & \textsc{PO\_Cross}: & (P_1.inS_3 \wedge P_2.inS_3) \implies (P_1.Y \geq P_2.I \vee P_2.X \geq P_1.J) \wedge (P_1.Y \leq P_2.I \vee P_2.X \leq P_1.J)
\end{array}
$$

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 Runway due to the queues present in the model. However, the complexity of the Runway 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 [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 $Test_{\mathrm{PO}}$ and $Test_{\mathrm{WA}}$, we have a high confidence that the memory model provided by Lazy caching and Runway-PA8000 is sequentially consistent. The verification of serial memory was straightforward.

Description of a Bug found in preliminary model of lazy caching: The following bug in our model of Lazy Caching was caught by a violation of PO_CROSS in $Test_{\mathrm{PO}}$. 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 $Test_{\mathrm{PO}}$. However, $Test_{\mathrm{WA}}$ (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 $Test_{\mathrm{WA}}$ 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 $Test_{\mathrm{PO}}$. This bug generated a long counter-example 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 *c2cw*s and the one-bit information in the client's cache lines to record whether the data is supplied, as explained in paragraph 4.4. After fixing the bug the PO condition passed.

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 [AG96] that are described in the SPARC V9 architecture manual [WG94]. This work may involve defining new rules as well as new tests corresponding to them.

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

References

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 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 intepretation: 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 Nowatzyk. 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 Java^{TM} 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.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9(29):690–691, 1979.

[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.