Design Verification of the S3.mp Cache-Coherent Shared-Memory System

Fong Pong, *Member*, *IEEE Computer Society*,
Michael Browne, Günes Aybay,
Andreas Nowatzyk, *Member*, *IEEE*,
and Michel Dubois, *Senior Member*, *IEEE*

Abstract—This paper describes the methods used to formulate and validate the memory subsystem of the cache-coherent *Sun Scalable Shared-memory MultiProcessor (S3.mp)* at three levels of abstraction: the *memory consistency model*, the *cache coherence protocol*, and the *implementation*.

Index Terms—Cache-coherent shared-memory multiprocessors, distributed directory-based protocols, formal verification, debugging.

1 Introduction

In this paper, we describe the methods used to formulate and verify the memory subsystem of the cache-coherent *Sun Scalable Shared-memory MultiProcessor (S3.mp)* [11] at three layers: the *memory consistency model* [1], the *cache coherence protocol* [19], and the *implementation*. At the most abstract level, S3.mp supports the *Total Store Ordering* (TSO) memory model [20], which disallows processors to read two write updates as having occurred in a different order. After defining the cache coherence property in the context of TSO, we verify the distributed directory-based cache coherence protocol by state enumeration-based tools. All protocol components are modeled as communicating finite state machines. Finally, we apply three simulation strategies: random-driven, directed script-driven, and application-driven, to catch different implementation errors.

Learning from our experience, efficient tools and well-planned simulation strategies reduce engineering effort in getting correct designs by eliminating ambiguities in the transition from high-level abstraction to low-level implementation. Moreover, complex design alternatives can be tried out more quickly. This is particularly important for cache protocols, in which design complexity is the major obstacle.

2 Overview of the S3.mp Architecture

The S3.mp implements a *CC-NUMA (Cache-Coherent Non-Uniform Memory Access)* multiprocessor system on a network of commodity Sparcstation-10 and / or Sparcstation-20 workstations (Fig. 1). Each node includes several processors with private caches and maintains a fraction of a globally shared address space. Processor caches within a node are kept coherent via the *Mbus* snooping protocol. For remote memory accesses, the memory controller (MC) translates bus transactions into a messages that are sent across the network to the remote memory controllers. Part of the physical memory at each node is allocated as a large *InterNode Cache (INC)*. The S3.mp system maintains inclusion between the InterNode Cache and the processor caches.

To maintain coherence among internode caches, the S3.mp uses a distributed directory-based protocol. Every memory block is mapped to a *home* node, which contains a directory entry for the block [3]. The directory entry maintains the global state of the block as well as a pointer to the first remote node having a copy of the block. Each INC block also has a pointer to the next node sharing the block. Thus, nodes which share a memory block form a linked list.

3 Total Store Ordering and Cache Coherence

The current S3.mp design supports the *Total Store Ordering* (TSO) model [20] which is illustrated in Fig. 2a. In this model, each processor is associated with a first-in-first-out (FIFO) store buffer. All stores are inserted into the store buffers and are executed at the memory sequentially in the program order. Loads can bypass stores, but a load access always looks up the local store buffer for the latest store to the same memory location. If the store exists, the load returns the value of the store. Otherwise, the load reads the value from the memory.

3.1 The S3.mp Implementation Model of TSO

Many possible implementations result in legal and indistinguishable behavior from the essential TSO model. The design as embraced in S3.mp is to provide the programmers a single global memory image via coherent cache systems [10], even though the memory modules may be physically distributed, as shown in Fig. 2b.

In principle, a coherent cache system allows multiple copies of the same memory location to exist in the system, but they are always consistent by having processors broadcast the values of updates or invalidations [19]. To avoid sending updates or invalidations on every store, some state is usually associated with each cache block. For instance, the simplest write-invalidate protocol has three states: invalid (data is not present in the cache), shared (multiple copies may exist and are coherent), and dirty (the copy is unique). A processor can always read a shared or dirty copy, but it can only write to a dirty copy. When a processor writes to an invalid or a Shared copy, it must first procure an exclusive copy.

Because all misses and all requests for dirty copies to the same cache lines are serialized either by a shared bus [6], [9] or by the directory [3], it is not difficult to understand why the memory system with a write-invalidate protocol behaves the same as the primary TSO model of Fig. 2a when the memory accesses are *atomic* [10]. Under atomicity, it is clear that all stores to all memory locations are *temporally* ordered and no two processors can observe different orders for two stores to the same memory location. Thus, we can easily identify the *latest* store performed at the memory, as defined by Censier and Feautrier [3]:

Definition 1 (Data Coherence). *A memory system is coherent if the value returned on a LOAD is always the value given by the latest STORE with the same memory location.*

However, temporal ordering is impossible in practice because the memory accesses are actually not atomic and the invalidations take time to propagate. As a result, the concept of data coherence and the notion of *latest* store, as in the definition of data coherence, must be interpreted in a different way, i.e., not in its *literal* or *temporal* sense.

Consider the example of Fig. 3. Initially, processors p_1 and p_2 share a copy of memory location *x*. p_0 executes the store as soon as it receives the data block from the memory, while invalidations may be still in transit to p_1 and p_2. As a result, data inconsistency can occur during the shadowed period of Fig. 3. p_0 is allowed to

• *F. Pong is with Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304. E-mail: fpong@hpl.hp.com.*
• *M. Browne and G. Aybay are with Yago Systems, Inc., 795 Vaqueros Ave., Sunnyvale, CA 94086. E-mail: gunes@yagosys.com.*
• *A. Nowatzyk is with Digital Equipment Corporation, Western Research Lab, 250 University Avenue, Palo Alto, CA 94301. E-mail: agn@acm.org.*
• *M. Dubois is with the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90082-2562. E-mail: dubois@paris.usc.edu.*

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 105896.

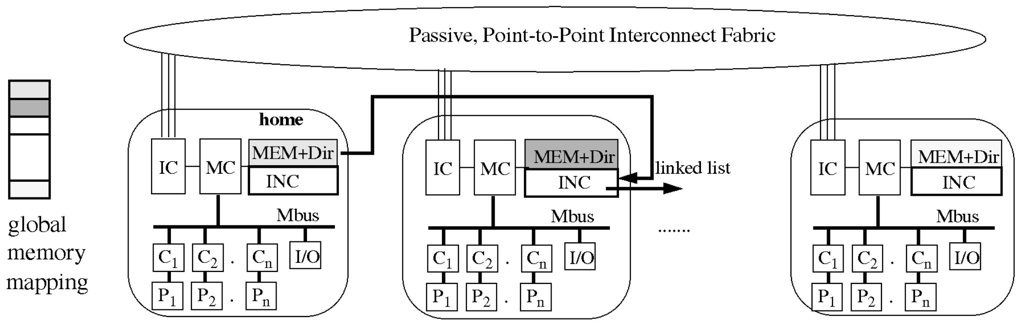

Fig. 1. Overview of the S3.mp system and global memory mapping.

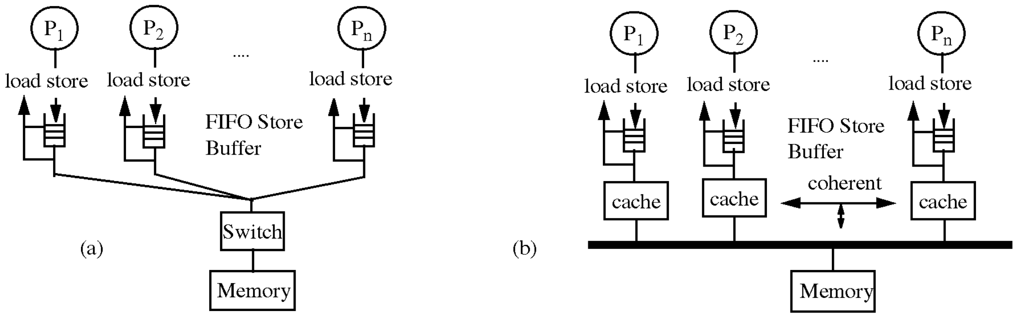

Fig. 2. The primary TSO model and an implementation based on a Cache Coherent Memory System.

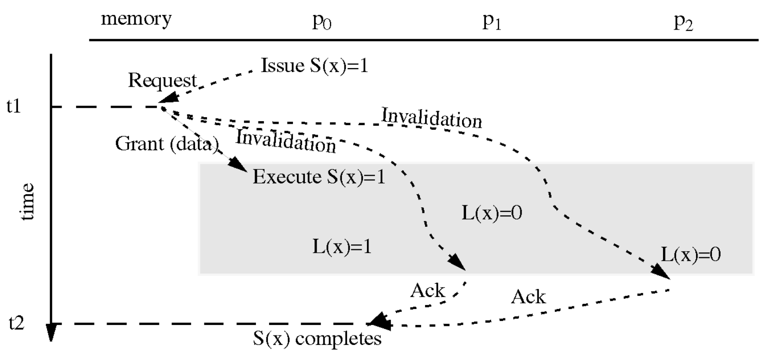

Fig. 3. Temporary data inconsistency according to data coherence.

read the new value, while the old value is still accessible by p_1 and p_2. This violates the strict, temporal definition of data coherence.

However, the strict, temporal interpretation of data coherence is illusive. The reason is that programmers cannot control the speed of processors p_1 and p_2. If p_1 and p_2 had run a little bit faster, the two loads of p_1 and p_2 would have been done earlier and the temporal interpretation of strict data coherence would have been respected. Therefore, Scheurich defined the concept of *general coherence* [17]:

DEFINITION 2 (General Coherence). *A system is generally coherent if a total store order is enforced on all stores to the same address and if any processor can only observe these stores in the total order.*

In a generally coherent system, processors cannot observe values of the same data in different orders, but, at any time, processors can observe different values produced at different times. If, at any time, all processors would stop issuing memory accesses, all copies would eventually reach the same value. Provided there is forward progress, a strict data coherent system is indistinguishable from a generally coherent system.

The S3.mp cache system supports general coherence. Write-invalidations are processed according to the scenario of Fig. 3, in which the memory controller does not accept any other request to word *x* during [t1, t2].

4 VALIDATION OF THE S3.MP CACHE COHERENCE PROTOCOL

4.1 Model and Correctness Properties

Fig. 4 shows the abstract S3.mp model in which all modules are modeled as finite state machines. Several abstractions are made to save complexity, but without compromising the properties to verify. For checking data consistency, we only model one cache block of one memory location. Replacement of the block is modeled by assuming that it can happen at any time [13], [14]. The home node of the block is distinguished from other homogeneous remote nodes. Each node consists of only one processor and the first-level caches are not modeled to focus the verification on the coherence among INCs (the Mbus snooping protocol supporting the first-level coherent caches and the mechanisms for maintaining the inclusion property are well understood).

The topology of the network is not fixed. Processors communi-

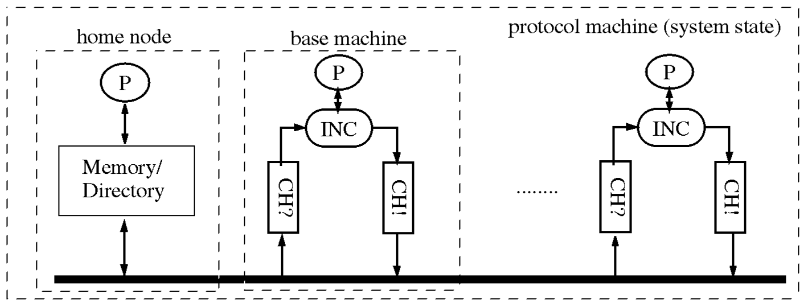

Fig. 4. Verification model for the S3.mp protocol.

cate via message receiving channels (CH?) and message sending channels (CH!). The state of a message channel is defined by the set of messages floating in the channel. These channels are non-FIFO buffers to model the unordered network. Also, we assume that messages are never lost and duplicated.

We verify two safety properties:

1) A state machine should never receive a message unspecified in its current state, and
2) The memory system must correctly support general coherence.

Violations of the first property are called *unexpected message reception* errors and are easy to detect. To verify the second property, the verification model keeps track of the *symbolic* status of all data copies, which can be either *fresh* or *obsolete* [14]. In short, when a processor commits a store access, it defines the most recent data copy and all other data copies are tagged as obsolete. Data inconsistency is detected when a processor can read a obsolete data copy.

4.2 Verification Methods

We have applied two tools, the Murφ [5] and the SSM [16], to verify the S3.mp protocol. The Murφ uses a straightforward *state enumeration* method, which is the most commonly used method for the verification of communication protocols. This method explores the entire state space of a protocol model; however, this is not feasible when the state space is very large.

The SSM system is based on a symbolic state model which exploits the *symmetry* and *homogeneity* of cache protocols to reduce the size of the state space. For example, all base machines in Fig. 4 are functionally identical. A single, canonical system state is sufficient to represent the set of states obtained by all permutations of the states of identical elements. Moreover, a unique property of cache protocols is that the exact number of data copies in a shared state is irrelevant to protocol correctness. On the other hand, it is critical to keep track of a cached copy in state dirty or modified. Thus, the SSM maps global states to more abstract states by grouping caches in the same states. For example, an abstract SSM state (Inv^+, RO*) represents all the global states in which “one or multiple caches are in the Invalid state, and zero, one, or multiple caches are in the Read-Only state.” Based on this compact representation, there is no need to list explicitly all global states as in a state enumeration method. Recently, a similar approach to SSM has been developed for Murφ [7]. For details of Murφ, SSM, and other techniques, one can refer to [15].

Although the SSM has the general advantage of verifying cache protocols independently of the model size, we need to adapt the method for S3.mp because it is difficult to abstract linked lists [14]. Specifically, processors which share the block are explicitly tracked in a linked list, while other processors are abstractly lumped together. Similar to the concept of [7], we start the verification runs by limiting one cached copy at any time. We then incrementally allow one more processor to share the data block in every successive run until the verification results stabilize [7].

4.3 Verification Results and Quality

Due to maintenance of linked lists by a distributed algorithm and the non-FIFO behavior of the interconnect network, the complex S3.mp protocol is a challenge for design verification tools. The protocol has a rough count of about 30 stable/transient cache states and 20 memory states. (These states represent branches in the microprogram implementation of the protocol.) The number of states is far more than what is typically needed for protocols using a central directory [16].

Table 1 summarizes the verification results by using Murφ and SSM on a Sun4/690 system with 500Mb memory. Learning from our experience, Murφ is efficient with small models (up to three nodes, memory bounded). It is a useful tool for catching most errors in the early stage of a design. However, an error due to ruptured linked lists was not detected by Murφ, since this error involved four or more nodes to be activated [14]. This error was later found by the SSM tool, which demonstrates its strength in verifying protocols independent of model size.

TABLE 1
VERIFICATION RESULTS BY USING MURφ AND SSM.

	Murφ	**SSM**
Max Model Size (# nodes)	3	∞
Unspecified Message Receptions	Found	Found
Data Inconsistency (Stale Write-back)	Found	Found
Data Inconsistency (Ruptured Linked List)	Not Found	Found

5 VALIDATION OF THE IMPLEMENTATION

A correct cache protocol at the level of finite state machines does not preclude implementation errors, but it provides invaluable guidance toward implementation. At present, testing and simulations are still the most efficient methods to detect implementation errors because formal verification methods, such as state enumeration, cannot handle the overwhelming complexity of all implementation details. In this section, we first overview the S3.mp cache coherence engine. The simulation environment and results are then described.

5.1 The S3.mp Cache Coherence Engine

The S3.mp cache coherence engines, RMH and RAS, which are both microprogrammed and multithreaded [12], are integrated on a customized memory controller (Fig. 5). When a memory access is received from the Mbus, the address is checked to determine whether it is mapped locally or cached in the INC. If the memory address is mapped remotely and the data is not in the INC, an appropriate request is sent to the home node of the block. Meanwhile, the access is suspended and monitored by the RMH. The RMH retrieves data from the home node, services invalidations and data forwarding requests, and maintains the INC. On the other hand, the RAS is responsible for handling requests from

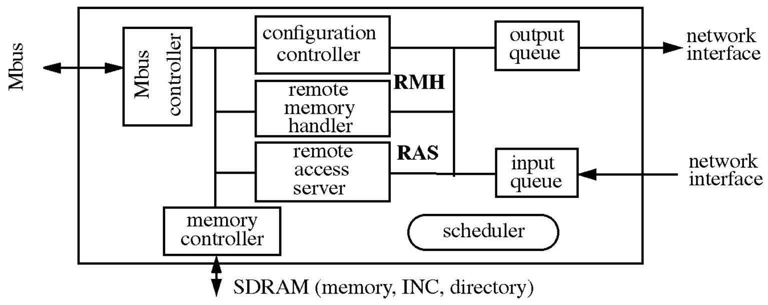

Fig. 5. Structure of the memory controller.

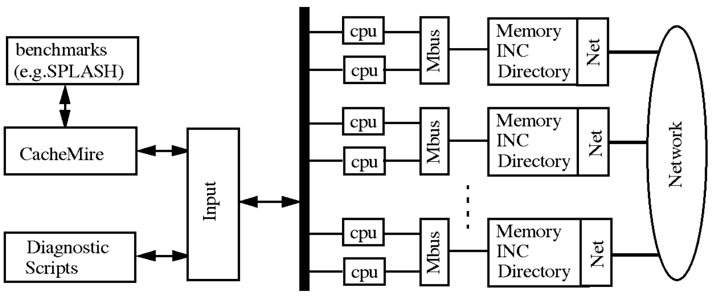

Fig. 6. The S3.mp simulation environment.

TABLE 2
APPLICATIONS OF SIMULATION STRATEGIES

	Random Simulations	Directed Scripts	Applications-Driven
Functional Error	Applied, Errors found	Applied, Errors found	Applied
Performance Error	Applied	Applied	Applied, Errors found
Behavior Error		Applied, Errors found	

remote nodes to locally mapped blocks. It services remote data requests, maintains the directory, and generates invalidation and data forwarding messages.

Another critical design feature is that messages are given priorities and are delivered according to their priority to prevent deadlock. Due to flexible topology and the message transport protocol, the order of message delivery network is not guaranteed. To detect performance bottlenecks, transactions are aborted after a preset time-out counter expires.

5.2 Test and Simulation

5.2.1 The Simulator

We have developed a cycle-accurate simulation system, which models the activities of the processors, the Mbuses, and the memory controllers cycle by cycle (Fig. 6). Inputs to the simulator are high-level scripts or memory reference streams provided by the CacheMire, a program-driven functional simulator of SPARC processors [2]. The simulator provides several features which, in our opinion, are important for multiprocessor design verification tools:

1) It provides a high-level input language with useful constructs such as loop control and if-else conditionals,
2) It allows quick changes of system configurations such as parameterizing the number of processors on a bus,
3) It facilitates quick simulations by swapping functionally equivalent modules in C language for the corresponding, but slow RTL (register transfer level) modules,
4) It lets explicit specification of network delays on coherence messages in order to exercise various parts of the design such as microcode segments of cache protocol engines, and
5) It has accurate bookkeeping of coverages and allows selective report of diagnostic traces which can be displayed in an interactive visual debugger.

5.2.2 Strategies and Results

In accordance to the classification in [21], we check the design for three types of errors:

1) The system should never enter wrong states and produce incorrect results (functional errors),
2) The system should not deadlock and processors should not be starved due to the priority scheme (performance errors which are detected when a time-out counter expires), and
3) The system operation should never result in violations of the TSO memory model (behavior errors).

We use three simulation strategies:

1) random-driven,
2) directed script-driven, and
3) application-driven simulations.

The parallel applications are from the SPLASH suite [18]. Applications of the three schemes to check different categories of errors are shown in Table 2.

Generally speaking, random simulation is the most popular validation strategy at the chip level (RTL). For example, the SPUR system [21], the SGI Challenge [6], and the recent Pentium Pro multiprocessor system [9] are validated by random simulations.

TABLE 3
CHARACTERISTICS OF DESIGNS AND THEIR IMPLICATIONS ON QUALITY OF RANDOM SIMULATIONS

SPUR, SGI Challenge, and Pentium Pro SMP	S3.mp
Shared-bus snooping protocols-based SMPs; corner cases caused by racing requests to the same cache lines are greatly reduced and simplified	A distributed directory-based DSM based on prioritized and unordered interconnects, which requires precise and flexible control of network delays to interleave messages in some orders
Direct-mapped caches (the SPUR and the SGI Challenge), which makes easier to generate accesses to exercise replacements of cache blocks	3-ways set-associative INCs, which requires four memory accesses to cache lines mapped into the same set in order to trigger replacements
Easy to check the results of random tests because of the support for strict data coherence (definition 1), data inconsistency is detected when different cached copies have inconsistent content	Difficult to check the results of random tests because the system can reach states in which processors read different values of the same location (general coherence, definition 2)

Ideally, the simulation strategies are geared toward two requirements:

1) All state transitions are exercised, and
2) The system is in a correct state after a series of test sequences.

Unfortunately, these two requirements are difficult to meet. In random simulations, processors issue arbitrary memory accesses, and very long simulation runs are needed in order to exercise all state transitions. Moreover, it is difficult to determine whether the outcome is correct if no appropriate control is applied. A general approach is to write a value to a memory location, read the location, and compare the returned value with the previously written value [6], [21]. In this approach, no new stores to the same memory location should be executed before the location is read and checked. This requirement is, unfortunately, difficult to achieve in random simulations without explicit blocking mechanisms for access generation [21].

For reasons listed in Table 3, random tests are unlikely to achieve the same validation quality for the S3.mp as for other designs. Although many functional errors were found by random simulations during the early validation phase, due to the randomness, the scheme failed to find some functional errors which were then discovered by directed scripts.

In principle, directed scripts are written to practice all branches of the microprograms for the cache coherence engines. This is done by explicitly generating coherence requests and delaying the transmission of coherence messages among nodes. Fig. 7 is a simple example with two nodes. Node 1 is the home node which initializes the target cache line with value 0xfffffff0. The remote node 2 then has a read miss, which have node 1 provide a data copy and an acknowledgment CR_ACK. We delay the CR_ACK, followed by a write-invalidation by node 1. Thus, we create a race between the CR_ACK and the invalidation from node 1 to node 2.

```
include "../diags/config.def";
include "../diags/tlb.def";
// ... more header omitted

if (node == 1){ // home node
        STORE(0060ff:140,4,0xfffffff0);// initialize some data to target line
        x = 1; // wake up the other node (a global variable not in the S3.mp address space)
        DELAY_PACKET(001,002,QUAD,CR_ACK,50); // delay 50 cycles
        LOAD(0060ff:140,4,0xfffffff0);
        STORE(0060ff:140,4,0xdeadbeef); // update the value
        x = 2;
        LOAD(0060ff:140,4,0xdeadbeef);
}
if (node ==2) { // a remote node
        wait x == 1;// wait until data is initialized
        LOAD(cc00ff:140,4,0xfffffff0);// get the line through remote access
        wait x == 2;
        LOAD(cc00ff:140,4,0xdeadbeef); // new value must be 0xdeadbeef
};
```

Fig. 7. A simple example diagnosis program.

One of the most notable errors which were not uncovered by random simulations is caused by write-back requests. Again, it is difficult to control random simulations to generate enough cache replacements to observe a particular access pattern for the three-ways set-associative INCs of S3.mp. Another interesting error exclusively discovered by directed scripts was a violation of the TSO memory model. This error was due to a mistake in the locking mechanism used by the memory controller to prevent a node from exporting values of stores which are not yet globally performed. The scripts were elaborately designed to test whether processors observe the same order of stores to different memory locations.

Unfortunately, directed scripts must be developed manually, which is inefficient and tedious. Therefore, the testing scripts are normally short programs which are designed to explicitly check particular parts of the system design.

To complement the random simulations and directed scripts, we have also tested the system under memory access streams obtained from the SPLASH benchmark [18]. The hope is that benchmark programs normally demonstrate a wide range of access patterns and, thus, the simulation of actual benchmarks is likely to discover errors which are sensitive to access patterns difficult to make up in random simulations and directed scripts. In our experiments, only one performance error was discovered by this scheme. The source of this error derives from the different priority levels assigned to messages. Messages of lower priority are normally blocked by requests of higher priority. During initialization of large arrays, requests of lower priority may be accidentally starved when a node is swamped by continuous flows of requests of higher priority. This error was not found by random simulations and directed scripts because the size of the data set is normally small in these two schemes.

6 CONCLUSION

We have described the various schemes applied to validate the memory system design of the S3.mp cache-coherent shared-memory system at different levels of abstraction. Based on a top-down approach, we clearly define the cache coherence property in the context of TSO memory model supported by the S3.mp. We have then shown that development and validation of cache protocols can greatly benefit from state-based verification methods. For the detailed implementation, we have demonstrated that random simulation is simple and efficient for catching errors in early phases, but it is difficult to generate tests to cover all possible cases. This is particularly true for the S3.mp system, which allows messages to be delayed in the network for a long time. To assist random simulations, we further test the system by the applications of specifically written scripts and of benchmarks. Our results show that the two schemes complement random simulations by detecting additional errors.

From our experience, we expect a tighter integration of formal methods and simulations in the future. One important feature of formal verification methods is to visit all possible states and explore all possible transitions. Therefore, it should be feasible to generate test scripts during the exploration of the state space in a formal method. This approach will significantly reduce the effort needed to develop suitably directed scripts. Another issue is the synthesis of a protocol implementation, which is the microprogram in the case of S3.mp. The approach taken in Teapot [4] is to derive the protocol and the model for verification from a common specification. This technique has great potential because it is difficult to design a correct cache protocol. With the aid of synthesis tools, it is possible to develop protocols which are suitably optimized for applications with different memory access behaviors. However, it is not clear whether the technique can be generalized to architectures and memory models.

REFERENCES

[1] S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer*, vol. 29, no. 12, pp. 66-76, Dec. 1996.

[2] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenstrom, "The CacheMire Test Bench—A Flexible and Effective Approach for Simulation of Multiprocessors," *Proc. 26th Ann. Simulation Symp.*, 1993.

[3] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, vol. 27, no. 12, pp. 1,112-1,118, Dec. 1978.

[4] S. Chandra, B. Richards, and J.R. Larus, "Teapot: Language Support for Writing Memory Coherence Protocols," *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, May 1996.

[5] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang, "Protocol Verification as a Hardware Design Aid," *Proc. Int'l Conf. Computer Design: VLSI in Computers and Processors*, pp. 522-525, Oct. 1992.

[6] M. Galles and E. Williams, "Performance Optimizations and Verification Methodology of the SGI Challenge Multiprocessor," *Proc. Hawaii Int'l Conf. System Sciences*, vol. 1, pp. 134-143, Jan 1994.

[7] C.N. Ip and D.L. Dill, "Verifying Systems with Replicated Components in Murφ," *Proc. Int'l Conf. Computer-Aided Verification*, 1996.

[8] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th Int'l Symp. Computer Architecture*, pp. 148-159, June 1990.

[9] D.T. Marr, S. Natarajan, S. Thakkar, and R. Zucker, "Multiprocessor Validation of the Pentium Pro," *Computer*, vol. 29, no. 11, pp. 47-53, Nov. 1996.

[10] T. Nipkow, "Formal Verification of Data Type Refinement—Theory and Practice," *Lecture Notes in Computer Science*, vol. 430, pp. 561-591. Springer-Verlag, May / June 1989.

[11] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin, "The S3.mp Scalable Shared Memory Multiprocessor," *Proc. Int'l Conf. Parallel Processing*, 1995.

[12] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin, "Exploiting Parallelism in Cache Coherency Protocol Engines," *Proc. First Int'l EURO-PAR Conf.*, pp. 269-286, Aug. 1995.

[13] F. Pong and M. Dubois, "A New Approach for the Verification of Cache Coherence Protocols," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 8, pp. 773-787, Aug. 1995.

[14] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois, "Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study," *Proc. First Int'l EURO-PAR Conf.*, pp. 287-300, Aug. 1995.

[15] F. Pong and M. Dubois, "A Survey of Techniques for Verifying Cache Coherence Protocols," *ACM Computing Surveys*, vol. 29, no. 1, pp. 82-126, Mar. 1997.

[16] F. Pong, "Symbolic State Model; A New Approach for the Verification of Cache Coherence Protocols," PhD dissertation, Dept. of Electrical Eng.-Systems, Univ. of Southern California, Aug. 1995.

[17] C. Scheurich, "Access Ordering and Coherence in Shared Memory Multiprocessors," PhD thesis, Univ. of Southern California, 1989.

[18] J.P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, vol. 20, no. 1, pp. 5-44, Mar. 1992.

[19] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer*, vol. 23, no. 6, pp. 12-24, June 1990.

[20] Sparc International, *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1994.

[21] D.D. Wood, G.A. Gibson, and R.H. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation," *IEEE Design and Test of Computers*, pp. 13-25, Aug. 1990.