Verification of an Industrial CC-NUMA Server

Rajarshi Mukherjee
Fujitsu Laboratories of America
595 Lawrence Expressway
Sunnyvale, CA 94085
USA

Yozo Nakayama Toshiya Mima
Fujitsu Ltd.
1-1 Kamikodanaka 4-chome, Nakahara-ku
Kawasaki-shi, Kanagawa-ken 211-8588
Japan

Abstract

Directed test program-based verification or formal verification methods are usually quite ineffective on large cache-coherent, non-uniform memory access (CC-NUMA) multiprocessors because of the size and complexity of the design and the complexity of the cache-coherence protocol. A controllable biased/constrained random stimuli generator coupled with an error detection mechanism using scoreboards and feedback with coverage analysis tools is a promising alternative methodology. We applied this methodology to verify a shared memory and message passing multiprocessor system consisting of 32 and 64 bit processor-based symmetric multiprocessing (SMP) servers connected by a proprietary cache coherent router-based interconnect fabric. This paper describes the problems faced, solutions implemented, and design decisions taken to design the scoreboard and discusses the errors found by this methodology.

1 Introduction

Traditional verification methodology based on identifying functions and other aspects of the design to be tested and writing directed test programs to exercise them is being overwhelmed by the rapid growth in size and complexity of today's designs. Use of a controllable biased/constrained random stimulus-based simulation coupled with scoreboards to analyze and detect error conditions and coverage analysis tools to provide feedback on the quality of the vectors generated has been shown to achieve coverage goals more quickly than directed test program-based verification [1]. In this paper we apply this methodology to verify an industrial server system. We concentrate on issues in designing the scoreboard which is a critical and complex entity in the methodology and in large part determines its success or failure. The server consists of a set of Intel McKinley/Foster-based symmetric multiprocessing (SMP) Processor Nodes and IO nodes connected by a network of proprietary Cache Coherent Routers. The Cache Coherent Router and its supporting chips constitute the ASICs under test. The verification environment supports both directed and random simulation. Simulation vectors for directed simulation are typically written to check a specific function or condition of the system. Therefore, it is usually easy to check if the system passed the simulation or failed it. However for random simulation the steps of execution are not known before-hand. In the absence of a *golden* model which can be simulated alongside the design, *self-checking* can be used to detect design errors. Unit-level self-checking by inserting assertions in RTL often cannot detect system-level errors. This motivates the need for a *scoreboard* that can globally *watch* the simulation, *analyze* the information it gathers, make *intelligent deductions* about the correctness of the system under test, and *output pertinent information* about an error once it is detected.

The rest of the paper is organized as follows. We discuss related literature in Section 2. In Section 3 we describe the overall architecture of the enterprise server, with an emphasis on the architecture of the Cache Coherent Router. In Section 4 we describe the salient features of the cache coherence protocol of the system. In Section 5 we describe the verification testbench. Detailed description of the scoreboard is given in Section 6. In Section 7 we discuss the different errors that were detected by the scoreboard in the design and the verification environment. We present our conclusions in Section 8.

2 Existing Literature on Verification of CC-NUMA Multiprocessors

Several verification efforts on CC-NUMA multiprocessors have been reported in literature. Application of formal verification to the HAL S1 System cache coherence protocol has been reported in [2]. In this work an abstract model of the directory-based cache coherence protocol was created using Murphi [3, 4] description. The abstract system consisted of two types of abstract nodes. One type contained

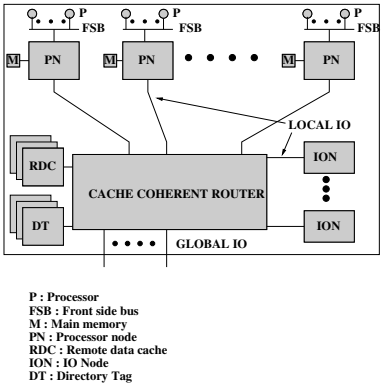

Figure 1: *Basic Configuration of the Server (Basic Server Unit)*

just a memory and a directory, and in the other type just a processor cache was modeled. This abstract design had a much smaller state space. The Murphi verification tool was applied to carry out the verification. This methodology, applied very early in the project, can uncover protocol level errors. However, since it works on an abstract version of the design it cannot uncover any RTL errors. In [5] the authors present their methodology to automatically generate executable test suites for the cache coherence protocol of BULL's CC-NUMA architecture. This paper deals with *black box conformance testing* where the behavior of the implementation can be observed only through its interactions with its environment. Testing consists of stimulating the design and observing its reactions at the interfaces. In [6] a technique to target error-causing interactions among different components in a design has been presented. This enables automatic generation of test vectors which exercise all transitions of control logic during simulation. The authors have applied this technique to validate an embedded dual issue processor in the node controller of the Stanford FLASH Multiprocessor [7] which supports both cache coherent shared memory and high performance message passing.

3 Architecture of Server

The server system in this work is a cache-coherent non-uniform memory access (CC-NUMA) multiprocessor with distributed shared memory and message passing and is targetted towards large parallel applications as well as a large number of single-threaded applications running in parallel. The basic configuration of the server consists of **K** processors as shown in Figure 1. The system can scale upto a **P** $\times$ **K**[1] processor configuration.

3.1 Basic Server Unit

Each Basic Server Unit (Figure 1) consists of the following units: **(1)** A set of Cache Coherent Routers (for the sake of simplified explanation in this paper, we have assumed only 1 Cache Coherent Router in each Basic Server Unit) **(2)** A set of SMP Intel McKinley/Foster-based Processor Nodes. Each Processor Node is connected to the Cache Coherent Router through Local IO Ports. A bus-based snooping protocol keeps the processors coherent in each Processor Node. **(3)**A set of IO Nodes, each hosting IO devices, and each connected to the Cache Coherent Router through Local IO Ports. **(4)** A set of Remote Data Cache units. **(5)** A set of Directory Tag units. The Cache Coherent Router has a set of Global IO Ports. These ports are used to communicate with Cache Coherent Routers in other Basic Server Units through a proprietary fast interconnect fabric. A brief description of the functionality of the Cache Coherent Router is given below.

3.1.1 Cache Coherent Router

The Cache Coherent Router analyzes each packet that comes in and carries out various actions to maintain cache coherence throughout the system. The principal functions carried out by the Cache Coherent Router are as follows. **(1)** It acts as the interface to the Directory Tag units which maintain ownership and caching state information of cachelines. Accesses to the same cacheline are properly sequenced to maintain coherence. **(2)** Correct routing of packets is carried out with a memory map of the system. **(3)** The Remote Data Cache bank maintains recently accessed cachelines that are stored in the shared caching state. This reduces the latency of future accesses to these cachelines. The Cache Coherent Router updates and maintains cache coherence of the Remote Data Cache and also properly sequences read and write requests to identical cachelines. **(4)** It handles several system management functions which include accessing and setting system registers and configuring the system.

The Cache Coherent Router has a bank of Local IO Ports and Global IO Ports to receive and foward incoming and outgoing packets. It also has a switching network for routing incoming and outgoing packets to appropriate protocol layer units or Local IO Ports, or Global IO Ports.

The Cache Coherent Router communicates with a bank of *Processor Nodes* and *IO Nodes*. The Processor Nodes act as the interface between the Cache Coherent Router and the processors. This unit is connected to the processor

[1]The exact values of K and P are proprietary information.

front-side bus, and the *Local IO Ports* of the Cache-coherent Router. It receives messages from the front-side bus, creates appropriate packets and sends them to the correct Local IO Port. It also receives packets from the Local IO Ports and decodes these packets and sends messages and data to the front-side bus. The IO Node acts as the interface between the cache-coherent router and the IO devices.

Data flow in the system takes place by means of transactions, each having a unique ID and consisting of a set of packets. The basic unit of data flow is a 128 bit long flit. Each physical link is shared by several virtual channels.

4 Cache Coherence Protocol

The directory-based cache coherence protocol of the system is designed to run on a distributed shared memory multiprocessor and provides a coherent shared image of memory to each Processor Node. Main memory is physically distributed across the Processor Nodes, is globally coherent and is addressed through a single flat address space. A portion of this flat address space is mapped onto each Processor Node. Reads to local lines are serviced faster than reads to lines which are not local. Each Basic Server Unit has a directory which maintains information about the nodes in the system that are currently caching any local cacheline. It also maintains the caching states of these lines.

A *transaction* originates in a Processor Node or an IO Node and consists of a series of messages. The first message in a transaction is a memory access request, for example a *read*, or a *write*, or *invalidate*. The messages in a transaction are of two types: a) Request b) Response. In order to describe all possible legal transactions in the cache coherence protocol, the Processor Nodes and the IO Nodes in the system are broadly classified into three categories: a) Source Node - the transaction (memory request) originates at this node. b) Home Node - the address of the cacheline accessed in a given transaction resides in a memory at this node. c) Sharer Node - this node is a sharer or owner of the cacheline being accessed by a given transaction.

An example of a transaction initiated by a request to read an uncached memory block is shown in Figure 2. The transaction is initiated by a request *RL* (Read Line) from Processor Node-A to read an uncached line. The request is routed to the Cache Coherent Router-A which sends the request via the Global Interconnection Network to the Cache Coherent Router-B which is connected to the home node of the cacheline. The Cache Coherent Router-B generates two packets simultaneously: a) a request packet *RLM* (Read Line from Memory) and sends it to Processor Node-B which is connected to the memory which contains the requested cacheline, and b) a response packet *CAS* (Cache As Shared), which is sent to Processor Node-A, indicating

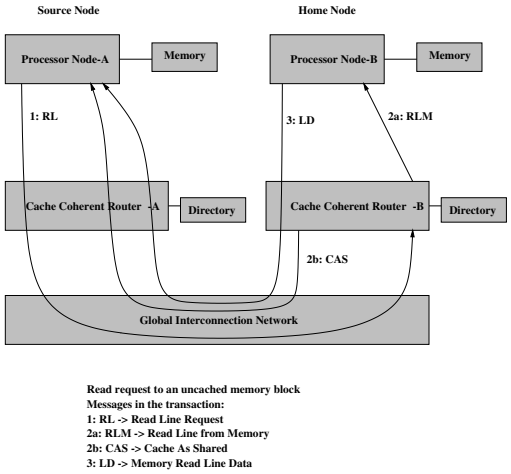

Figure 2: *Example of a Coherent Transaction*

that the cacheline must be cached in the shared state. At the same time an entry is created in the directory connected to Cache Coherent Router-B with the information of the sharing node and the caching state of the cacheline. The memory responds with the cacheline data. After receiving the data the Processor Node-B creates the data packet, which is a response packet of type *LD* (Line Data). This packet contains one header flit and several data flits and is routed through Cache Coherent Router-B, the Global Interconnection Network and Cache Coherent Router-A to Processor Node-A. Processor Node-A receives the data and sends it to the requesting processor. This ends the transaction. The Cache Coherence Protocol describes all possible legal transactions in the system.

5 Verification Testbench

The verification testbench for one Basic Server Unit is shown in Figure 3. This testbench consists of RTLs of Cache Coherent Router, Remote Data Cache, and Directory Tag. In addition, it contains instances of functional models of the Processor Nodes and IO Nodes, and instances of functional models of memory. In addition, there is a scoreboard and some amount of control logic used to control the testbench configuration and functionality. The functional models of Processor Nodes and IO Nodes can be programmed to create deterministic or random request packets which are sent to the Cache Coherent Router. Figure 4 shows how programming is done to create a particular packet instance. This instance can then be executed which causes the packet to be injected into the RTL. In this example a packet instance for a read type request is created for a cacheline with address “0xabcd”, for which the Home

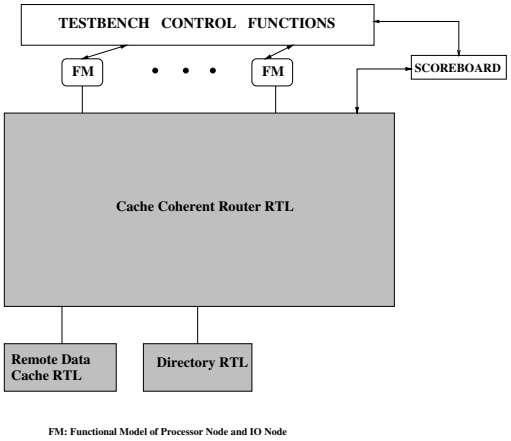

Figure 3: *Verification Testbench*

```
Packet instance = {
                    .home_node == N1;
                    .src_node == N2;
                    .opcode == read_line;
                    .cacheline_address == 0xabcd;
                };
```

Figure 4: *Packet Generation Through Functional Model*

Node is N1. The request originates from node N2 which is the Source Node. The instance can contain other information fields as well.

6 Scoreboard Design

In order to design the scoreboard the following issues had to be understood and addressed:

1. **Level of operation** - We had to decide which level the scoreboard would operate at. Broadly speaking, there are three layers of operation in the system - the protocol layer, the link layer and the physical layer. After detailed consideration, it was decided that the protocol layer would be most appropriate since it would allow the scoreboard to monitor the design at a higher level and also detect errors at the same level. This would reduce the complexity of the scoreboard and make it more robust. At the same time since errors would be caught and reported at the protocol layer, debugging would become easier.

2. **Scope of operation** - We had to decide which units in the system the scoreboard would have access to. Since the Processor Nodes, IO Nodes, and memory were third party chips, it was decided that these would not be monitored. Since the level of operation was chosen to be at the protocol layer, it was decided to focus on protocol level errors. Since the protocol could be understood by watching the signals at the Local IO Ports, it was decided to monitor these ports. Thus, all the signals shown in Figure 2 would be visible to the scoreboard.

3. **Checking goals** - The following self-checking goals were set for coherent transactions:

   - Verify correctness of cacheline data
   - Verify correctness of cacheline state
   - Verify that each transaction has all protocol level messages in correct order
   - Verify that no transaction has spurious messages

   For non-coherent transactions the scoreboard verified the mutual orderings of non-coherent writes to memory.

4. **Complexity goal** - A full-scale implementation of the cache coherence protocol would work well as a scoreboard. But, that would require significant effort almost similar to that of writing the RTL. Also due to the complexity, the scoreboard itself would be susceptible to errors. Therefore, abstraction was applied manually to the cache coherence protocol to minimize the number of message types to be monitored. (Note that this abstraction did not prune away any details of the design unlike abstractions typically used with formal verification methods.) This resulted in a simple algorithm which was used to update internal data structures of the scoreboard and detect errors.

5. **Efficiency goal** - The scoreboard operated as follows. Whenever it observed a new flit on any Local IO Port, it stopped the simulation, grabbed the flit and decoded its contents. These contents were then analyzed to detect errors and internal data structures of the scoreboard were updated with the new information. Once this was finished, control was returned to the simulator. Since this was an "invasive" process which delayed simulation, it was important for the scoreboard to be very fast. Special care was taken to make the data structures very compact and light weight.

The overall organization of the scoreboard is shown in Figure 5. There are three basic units in the scoreboard: a) *Transaction Storage*, b) *Data Storage*, and c) *Central Analysis Engine*. The Transaction Storage stores information about each new transaction observed on a Local IO Port. When the first message in a new transaction is observed on a Local IO Port, a new *Transaction Storage Entry* is created in the Transaction Storage. Each transaction in the system has a unique transaction ID. This ID is used to uniquely identify

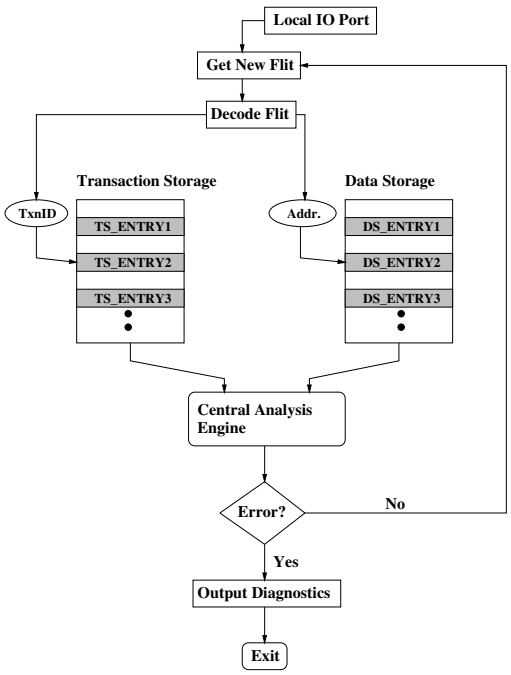

Figure 5: *Organization of Scoreboard*

the corresponding Transaction Storage Entry. A Transaction Storage Entry is deallocated when the transaction finishes. There is a Data Storage Entry in the Data Storage for each unique cacheline which is accessed during simulation. The Data Storage stores the correct data and state of each cacheline. Each *Data Storage Entry* is associated with two flags - a) a *V* flag which is TRUE when the data stored in the Data Storage Entry is *valid*, and b) an *S* flag which is TRUE if the main memory entry for this cacheline has invalid data, i.e. is *stale*. Figure 6 shows the state transition diagram of each Data Storage Entry. Each Data Storage Entry functions as a repository for the correct predicted data and caching state of the corresponding cacheline. The Central Analysis Engine identifies each transaction by observing its messages and independently computes (based on an abstracted cache coherence protocol-based algorithm) the new data and new state of the cacheline involved. Errors in the data and state of a cacheline are detected by comparing the data and state observed in the system with the data and state computed by the scoreboard. Since the scoreboard observes and identifies each transaction, it can detect any incomplete transaction, i.e. a transaction for which all messages have not been observed, and any illegal transaction, i.e. a transaction which has a spurious or incorrect message. Once an error is detected, the scoreboard stops further simulation and gives detailed diagnostic information to help the designer pinpoint the problem. Detailed diagnostic information includes the address of the cacheline involved in the error, the

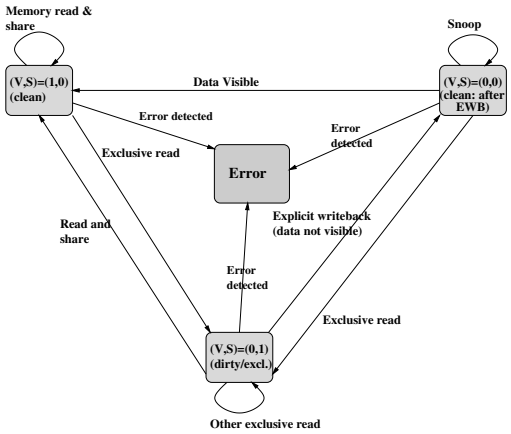

Figure 6: *State Transition Diagram of Each Data Storage Entry*

correct and erroneous data, and a trace of computation that led to the error.

7 Errors Detected by Scoreboard

The scoreboard was deployed during system-level verification of the server and detected more than 30 errors in the RTL and the verification environment. The details of the errors detected are presented below.

RTL Errors

The following types of errors in the RTL were detected using the scoreboard:

1. **Address decoding error** - Several errors in the RTL were found as a result of which the data in a cacheline seen at the RTL level was not identical to the data in the cacheline stored in the Data Storage in the scoreboard. These errors were traced to various types of decoding problems, and problems with timing of the write signals of memory.

2. **Incomplete transaction** - Several errors were found by detecting incomplete transactions. Once a transaction in the system is over, the corresponding transaction ID is released and can be reused for a subsequent transaction. Once a new transaction is observed in the system, the scoreboard allocates a new Transaction Entry in the Transaction Storage. Before allocating this entry it checks to see if any entry with the same transaction ID still exists in the Transaction Storage. If there is such an entry, that means that an earlier transaction did not complete legally.

3. **Duplicate/spurious messages** - The scoreboard keeps track of legal messages in a transaction. Therefore, it can detect an illegal message that violates the cache coherence protocol. Several errors were detected by this mechanism. Most of these errors were traced back to incorrect implementation of broadcast logic.

4. **Coherence error** - A coherence error which occurred as a result of a tricky timing issue in the Remote Data Cache was detected. The reason for the error can be explained as follows. The Remote Data Cache first receives an *invalidation* request. As a result, it responds with a confirmation of invalidation and starts to commit the invalidation. The central directory receives the confirmation and updates the information about this cacheline to "invalid". However, before it can finish the invalidation the Remote Data Cache receives a read request for the same cacheline and responds with the data. This causes the state of the cacheline to become illegal. This error was detected by the scoreboard error detection logic.

Verification Environment Error

In addition, the scoreboard detected a critical error in the functional model of Processor Node. This error caused the Processor Node to return data bytes in an incorrect *burst order*. This was detected as a cacheline data miscompare between the data observed during simulation and the data stored in the Data Storage. This error can be explained as follows. When a processor issues a request for a certain byte with an address X, the data is returned as a memory block within which the requested byte is contained. The block size for this design was 128 bytes. Now suppose the byte requested by the processor is the last byte in the returned block. In that case if the bytes are returned to the processor in a serial order with the first byte first, the processor would have to remain idle for 127 clock cycles (assuming one byte is returned in each clock cycle). Therefore to minimize this delay, depending on the position of the requested byte in the memory block being returned, the order in which the bytes are returned is different. This depends on the exact specification of each processor. By detecting a problem with the burst order implementation in the functional model of the Processor Node, the scoreboard essentially discovered an error in conformance with processor specification.

8 Conclusion

Cache coherent non-uniform memory access (CC-NUMA) multiprocessor architectures are becoming popular for high-performance enterprise server designs. These designs pose a significant challenge to traditional verification methodologies due to their size and complexity. In this paper we apply biased/constrained random stimuli-based verification along with scoreboards and coverage tools to verify an industrial server system. We describe the various challenges faced in designing an efficient and powerful scoreboard and give details of its goals and architecture. Finally we describe the different errors that were discovered by this methodology in the RTL and testbench.

References

[1] P. James, and C. Macionski, "Shotgun E", In Proceedings of the Verisity Users' Group Meeting, March 7, 2001.

[2] A. J. Hu, M. Fujita, C. Wilson, "Formal Verification of the HAL S1 System Cache Coherence Protocol", In Proceedings of the IEEE International Conference on Computer Design, 1997.

[3] D. L. Dill, "The Murϕ verification system", R. Alur and T. A. Henzinger, editors, In Proceedings of Computer-Aided Verification: Eighth International Conference, Springer-Verlag, July 1996. Lecture Notes in Computer Science Number 1102.

[4] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid", In Proceedings of the IEEE International Conference on Computer Design, 1992.

[5] H. Kahlouche, C. Viho, and M. Zendri, "An industrial experiment in automatic generation of executable test suites for a cache coherency protocol", In Proceedings of the International Workshop on Testing of Communicating Systems, 1998.

[6] R. C. Ho, C. H. Yang, M. A. Horowitz and D. L. Dill, "Architecture Validation for Processors," In Proceedings of the International Symposium on Computer Architecture, 1995.

[7] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH Multiprocessor," In Proceedings of the International Symposium of Computer Architecture, 1994.