Verifying a Multiprocessor Cache Controller Using Random Case Generation

David A. Wood, Garth A. Gibson, Randy H. Katz

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

The newest generation of cache controller chips provide *coherency* to support multiprocessor systems, i.e., the controllers coordinate access to the cache memories to guarantee a single global view of memory. The cache coherency protocols they implement complicate the controller design, making design verification difficult. In the design of the cache controller for SPUR, a shared memory multiprocessor designed and built at U.C. Berkeley, we developed a random tester to generate and verify the complex interactions between multiple processors in the functional simulation. Replacing the CPU model, the tester generates memory references by randomly selecting from a script of *actions* and *checks*. The checks verify correct completion of their corresponding actions. The tester was easy to develop, and detected over half of the functional bugs uncovered during simulation. We used an assembly language version of the random tester to verify the prototype hardware. A multiprocessor system is operational; it runs the Sprite operating system and is being used for experiments in parallel programming.

1. Introduction

The advent of powerful microprocessors has spurred recent development of multiprocessor architectures. A promising class of architectures are bus-based multiprocessors that use cache memories to reduce contention for their shared busses [Bell85]. These systems generally use hardware cache coherency protocols [Arch86] to maintain a consistent image of memory across all the processors (see Box on page 3). Implementing cache coherency in hardware simplifies the operating system, compiler, and applications software because it provides the illusion of a single-level, completely shared memory.

Cache coherency is fairly simple to implement using a *write-through* update policy, i.e., every processor write updates main memory. However, to achieve acceptable performance with more than a few processors requires a *write-back* update policy, i.e., processor writes update the cache, but only update main memory when replacing a modified cache block. Coherency protocols for write-back caches are notoriously difficult to design and debug. Some companies have had significant product delays because of bugs in their cache coherency implementations. We believe more extensive simulation could have significantly shortened their time to market, because simulations are much easier to debug than hardware.

As cache controllers, and entire caches, are integrated into single chip designs, the importance of simulation increases. The designers must spend more time verifying the design to decrease the time (and cost) of multiple fabrication cycles. The increased visibility of a simulation, coupled with a powerful debugging environment, more than compensates for the slower execution speed of the simulation.

However, a simulation requires input, in the form of test vectors or some higher-level format. But the complexity of these controller designs make it difficult to develop a comprehensive set of design verification tests. The number of cache states, blocks in each cache, and independent processors compound to create a large state space. In addition, coherency protocols are very

Multiprocessors, Caches, and Coherency Protocols

Multiprocessors promise improved system performance by simultaneously executing multiple instruction streams. However, programming multiprocessors has proven difficult. A shared memory model, where all processors have equal access to memory, eases the problem by allowing programmers to allocate memory without regard to access time.

A bus organization is the simplest interconnection topology for a shared memory multiprocessor. However, if all processor memory references must use the shared bus, this single resource will be saturated by a few processors. Cache memories can reduce each processor's load on the bus, by servicing most memory references locally.

Caches work in two ways: by keeping around the most recently accessed memory words because they are likely to be referenced again (*temporal locality*), and by fetching neighboring words because they are likely to be referenced soon (*spatial locality*) [Smit82]. Caches usually divide memory into fixed size chunks, called *blocks* or *lines*. Based on the address of a block, the cache places it into a particular *set*. The size of the set is known as the *associativity* of the cache: a two-way set-associative cache has a set size of two; a direct mapped cache has a set size of one. Caches maintain an *address tag* and *state* information with each block.

The *update policy* is another important parameter of cache design. The simplest caches use *write-through*, where all processor writes update memory. Unfortunately, since 10-20% of all processor references are writes, using write-through caches limits the number of processors that can effectively share a bus. Instead, caches usually use *write-back*, where processor writes only update the cache, and main memory is updated only when space must be freed for another block.

Managing write-back caches in uniprocessor systems is fairly simple, but it gets much more complex in multiprocessors. The problem arises because the same block may reside in multiple caches; when one processor changes a block in its cache, how are the other processors prevented from accessing stale data in their caches? Allowing processors to access stale data violates the single-level shared memory assumption, causing programs to execute incorrectly.

Cache coherency protocols prescribe a set of rules that prevent stale data. Modern shared bus systems usually use distributed protocols, where the state information is distributed among all the caches. In the Berkeley Ownership protocol, each block may be in one of four states in each cache: *Invalid, UnOwned, OwnPrivate, and OwnShared.* Ownership implies the right to update a block without first accessing the bus. Only one cache can own a particular block at a time, but many caches can have shared copies. In addition, each block has a *dirty* bit, to indicate whether it has been modified. The cache controllers monitor backplane traffic to determine when a a state change is necessary. If a cache "owns" a block, and another processor wants a copy, the cache must supply it, since the version in memory may be out-of-date.

sensitive to specific sequencing activity; offsetting a bus operation by a single cycle can result in completely different behavior. The myriad of sequence specific events makes it easy to miss important test cases.

An obvious alternative to specifying test patterns by hand is to automatically generate them using a computer. Unfortunately, while we understand how to generate patterns for *fault*

detection, at least for combinational logic [Kirk88], the same is not true for *design verification*. In fault detection, we verify that a given piece of logic implements a particular boolean function, in the presence of possible faults (e.g., stuck-at-0). In design verification, we verify that a particular boolean function implements a higher-level specification, in our case written in English. In addition, we want to verify that the higher-level specification is consistent and correct.

For the design of the SPUR cache controller chip, part of a large multiprocessor project at U.C. Berkeley, we created a random test generator to exercise the functional simulation of the SPUR memory system. Random test generation has been used in processor design verification to check pipeline interlocks and arithmetic algorithms [Maur88, Shal87]. Our tester, described later in the paper, uses a pseudo-random number generator to probabilistically generate all multiprocessor interaction cases. This tester required only two person-months to develop, while we optimisticly estimated one-person year to develop a comprehensive test suite by hand.

In the next section, we present an overview of the SPUR architecture, and describe enough implementation details to illustrate the complexity of the design. In Section 3, we describe the functional simulation and the random tester. Then we discuss our experience with the tester, and characterize the bugs uncovered during simulation. Finally, we summarize with our experience with the fabricated chips, and suggest areas for future work.

2. The SPUR Architecture

SPUR (an acronym for Symbolic Processing Using RISCs) is a multiprocessor workstation developed at U.C. Berkeley [Hill86]. It is a research prototype intended as a test platform for parallel LISP applications, and to evaluate different architectural alternatives to support these applications. The Sprite operating system [Oust88] provides a network environment that runs on both SPUR and Sun Microsystems workstations. A Common LISP system runs on the SPUR hardware [Zorn87] , with a package of routines that implement shared-memory processes, with mailboxes and MultiLisp-like futures.

A SPUR workstation contains up to 12 processor boards and 96 megabytes of memory. The processors communicate across a modified NuBus [Gibs88, IEEE86] to access memory and peripherals. Each processor board contains 3 custom VLSI chips and a 128 kilobyte cache memory. The RISC-style CPU [Patt85] uses tags to efficiently support the LISP programming language. The floating point coprocessor (FPU) chip implements IEEE 754 extended-precision floating point arithmetic. The third chip, the cache controller (CC), manages the cache and communicates with main memory and peripherals. Because this chip controls the multiprocessor aspects of the system, we will focus on it for the remainder of this paper.

The cache controller provides a (mostly) transparent view of memory, handling all cache hits and misses. It supports memory management functions using the *In-Cache Address Translation* mechanism [Wood86]. The chip also supports system functions, such as memory-mapped interrupts, interval timers, non-cacheable pages, and unmapped (physical) accesses.

The controller maintains cache coherency with a distributed protocol known as *Berkeley Ownership* [Katz85]. Because the protocol is distributed, each cache controller must monitor all backplane transactions. When another processor requests a block *Owned* by the local cache (i.e., is, or recently was, writable) , the cache controller responds by sending the data directly to the requesting processor. Other requests may require the cache controller to invalidate an *UnOwned* (i.e., read-only) copy of the block.

Requests from other processors may arrive at any time. To accommodate this asynchronous behavior, the cache controller consists of two communicating halves: the processor cache controller (PCC) and the snooping bus controller (SBC). The two sides run on separate, asynchronous clocks to allow the processor to run at a different speed than the 100ns backplane bus cycle. Each side has its own copy of the cache tags, so they can run independently until either a cache miss, or a snooping event requires the SBC to communicate with the PCC. Because they share a single copy of the cache data, normally allocated to the PCC, the SBC must arbitrate with the

PCC when it requires access.

Substantial internal state makes both sides of the cache controller fairly complex. The PCC's sequencer has over 80 states, and over 450 transitions (arcs) between the states; roughly half the complexity is for basic operation and cache coherency, the rest results from virtual memory support and other system support features. In addition, a 4 entry push-down stack allows the current state to be saved and later resumed (a hardware subroutine mechanism). Over 400 stack configurations are possible during uniprocessor operation, and nearly 2000 during multiprocessor operation (5 possible external requests). The SBC consists of 7 separate controllers, with an average of 9 states and 40 transitions each. Since these 7 controllers are semi-independent, the SBC has hundreds of possible state configurations. Just testing the uniprocessor cases, when the SBC is a slave to the PCC, is a non-trivial task. However, when you consider the multiprocessor cases, with the two sides running independently of each other, there are thousands of interaction cases.

The multitude of interaction conditions make verifying the design very difficult. Merely generating many of the conditions is hard, because they rely on the precise sequencing of the implementation. Writing design verification tests and test vectors is tedious and error-prone; because this task is so unpleasant, it is rare to find a thorough job. To address this problem, the random tester uses computer time, rather than human time, to obtain good simulation coverage.

3. Functional Simulation and the Random Tester

We implemented the SPUR cache controller [Wood87] in full-custom CMOS, fabricated through the MOSIS implementation service. Because of the lengthy fabrication cycle, and because debugging hardware is harder than debugging a simulation (especially custom integrated circuits), we wanted the chips to function correctly on first silicon. To achieve this goal, we developed a detailed functional simulation using the N.2 system from ENDOT [Rose83]. Local enhancements to this software provided a limited silicon compilation capability, allowing us to

generate much of the control logic automatically [Kong87]. Using vectors taken from the functional simulation, we used switch-level simulation to verify that the extracted layout matched the functional description. Later, we used these same vectors to test the fabricated chips.

Because of the complexity of the chip and the difficulty in writing test vectors, we felt that simulating the cache controller in isolation would not be enough to verify the memory system design. Instead, we developed a system level simulation, including models for the CPU, processor board (cache and bus interface), and main memory. By instantiating multiple copies of the processor board model we could simulate a 3 processor system, as illustrated in Figure 1.

Of course, a simulation requires input, and the output must be checked for correctness. Unlike most random test strategies, the random tester performs both functions. The key observation is that every multiprocessor test case is merely the interaction of (at least) two uniprocessor operations. And while generating all the multiprocessor cases deterministically is extremely difficult, generating all the uniprocessor operations is comparatively simple. By randomly generating the uniprocessor operations on the different processors, carefully choosing the addresses so that cache blocks will be shared, the tester can probabilistically generate all the multiprocessor interaction cases. In other words, if the tester runs this random generation procedure forever, it will generate all the multiprocessor interaction cases with probability 1.

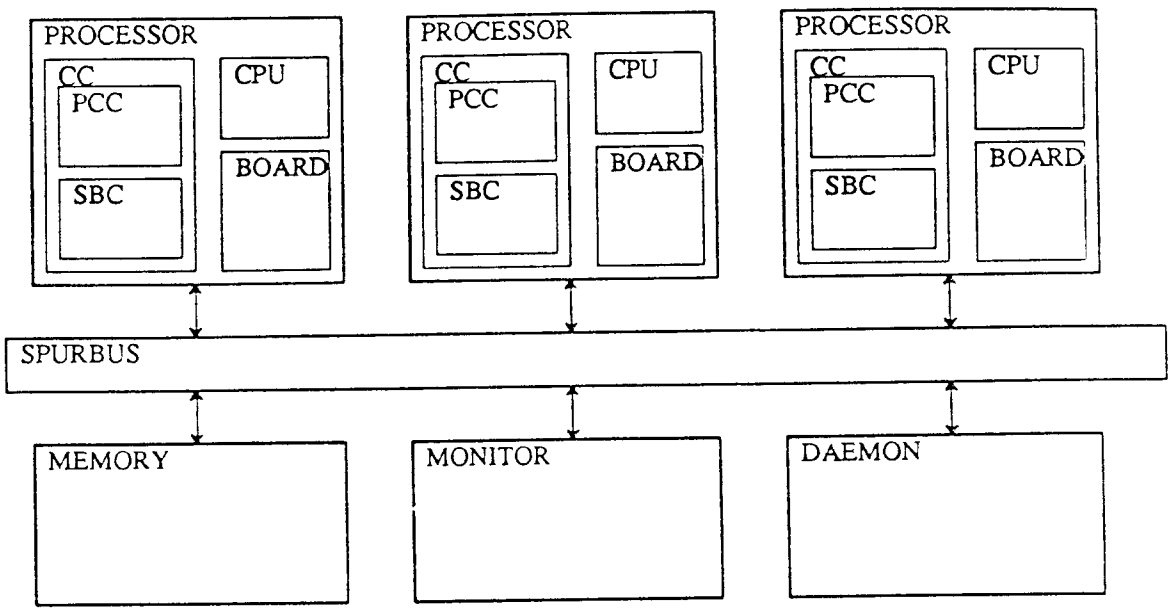

Figure 1: Simulation of 3 Processor System

To illustrate this concept, consider the following simple example. Suppose the "test" is to verify that each square on a checker board is covered with a piece. A probabilistic approach is to roll two 8-sided dice, using one to generate the x-coordinate and one the y-coordinate. On each iteration, roll the dice and check the square with the coordinates (x,y). It is easy to see that if repeated forever, then this method will check every square with probability 1. Of more interest, it requires only 556 iterations to have a 99% probability that all squares have been tested[1]; this is less than a factor of 10 worse than optimal. While no one would solve this trivial problem probabilistically, this example illustrates why random testing works effectively on difficult problems.

---

[1] $F(x,t) = P\{$ exactly x squares have been checked by iteration $t\}$

$$F(x,t) = \frac{x}{64} F(x,t-1) + \left[1 - \frac{x-1}{64}\right] F(x-1,t-1)$$

In the random tester, a "stub" module replaces the CPU, accurately modeling the interface, but operating completely differently on the inside. The stub CPU generates memory references by randomly selecting from a predetermined script, rather than executing assembly language instructions. These scripts are broken into *action/check* pairs, where the action causes some state change and the check verifies that the change happened correctly. For example, an action might be to write a particular value to a particular address. The corresponding check verifies that the update occurred correctly, signaling an error if not. Because of the random sequencing, an arbitrary amount of time and a random permutation of other actions and checks may elapse between an action and its check. Figure 2 illustrates the action/check scripts. The keywords ACTION, CHECK, and END delimit a single action/check pair. The interspersed lines contain cache interface commands, that map directly onto the main CPU/CC interface signals. The first field specifies the cache operation, e.g., Read32, Read64, Write32, TestSet, etc. The next two fields specify the address and data values, respectively. On write operations, the CPU supplies the specified data; on read operations, it compares the specified data to the data returned on the read, signaling an error if they don't match. Reserved data values instruct the CPU to expect specific

```
ACTION
        Write32         0x00000660    0x05050505    USER
CHECK
        Read32          0x00000660    0x05050505    KERNEL
        Write32         0x00000660    0x90909090    KERNEL
END
ACTION
        TestSet         0x0000A800    0x0           USER
CHECK
        Read32          0x0000A800    0x1           USER
        Write32         0x0000A800    0x0           USER
END
```

Figure 2: Example of Action/Check Script

exception conditions to occur (e.g., page fault). The final field specifies kernel (privileged) vs. user (non-privileged) mode. For example, the test and set operation in the second action/check pair affects address 0x0000A800. It expects the current value to be zero, then sets it to be one (the set is implied by the cache operation). The check, executed some time later, verifies that the set occurred correctly, then clears the word so that the next execution of the action will find the value zero.

A simple translator generates the appropriate simulator commands to initialize the stub CPU at run-time. Figure 3 illustrates the stub CPU's internal data structures. Using a pseudo-random number generator, the CPU selects an action/check pair and checks the *pending* bit. This bit indicates whether the check is pending, and needs to be executed before the action. This enforces a strict alternation within a single pair, which is necessary to prevent false errors from being detected. Based on the pending bit, the CPU begins sequential execution of the corresponding cache operations. It monitors the interface signals from the cache controller, and advances to the next operation only when the current one completes (i.e., the cache controller asserts either DataValid or an exception).

Each processor node executes a similar but distinct script. In general, we assign each action/check pair to a unique word (or more) of memory, preventing interference from another processor (or action/check pair on the same processor). The scripts for alternative processors are often assigned to different words in the same cache block, insuring that the block will move from one cache to another. By having different processors issue different sequences of read and write operations, we exercise all the cases of the coherency protocol. Because writing a good set of scripts requires an intimate knowledge of the high-level behavior, it should be done by a system designer.

Because the action/check pairs are self-checking, any error that results in incorrect data will be detected. However, because the time between the occurrence of an error and its detection via a

Action		Check		Pending
Loc	Length	Loc	Length	
0	1	1	2	1
3	1	4	2	0
•	•	•	•	•
•	•	•	•	•
•	•	•	•	•

	CacheOp	Address	Data	Mode
0	Write32	0x00000660	0x05050505	U
1	Read32	0x00000660	0x05050505	K
2	Write32	0x00000660	0x90909090	K
3	TestSet	0x0000A800	0x00000000	U
4	Read32	0x0000A800	0x00000001	U
5	Write32	0x0000A800	0x00000000	U
	•	•	•	•
	•	•	•	•
	•	•	•	•

Figure 3: Stub CPU Data Structures

check may be arbitrarily long, debugging can sometimes be difficult. In addition, certain types of bus protocol errors (e.g., bus arbitration violations) may not result in corrupted data, and therefore will not be detected. To address these problems, we implemented a *Monitor* model to watch the backplane and detect violations of the bus and cache coherency protocols.

For the Monitor to track coherency, the protocol needed a slight 'de-optimization'. In the original protocol, when a processor flushed a cache block (as the operating system does during certain virtual memory operations), the controller only wrote modified blocks to memory. If the block was clean, the controller changed the state to Invalid without a bus operation. Since there was no bus operation, the Monitor could not detect the state transition.

To fix this problem, we changed the protocol so an Owned block would be written back to

memory, regardless of whether it was modified [2]. This change decreases the performance of the system slightly, but greatly increases its testability. The trade-off was easily justified; as discussed in the next section, we found the Monitor invaluable for detecting bugs in the functional simulation.

SPUR's backplane bus, or SpurBus [Gibs88], is an extension of the Texas Instrument NuBus. Additional signals provide communication between processors to support cache coherency. The I/O devices are standard NuBus peripherals, and therefore do not take part in the coherency protocol. To include their behavior in the functional simulation, we developed a *Daemon* model that randomly generates I/O requests using the NuBus protocol. It follows the same action/check paradigm and detects corrupted requests. This Daemon uncovered several bugs in the design.

4. Simulation Experience

Before designing the random tester, we spent several person-months writing assembly language design verification tests and diagnostics. Despite the time commitment, the set of tests was grossly incomplete, covering less than half the of the uniprocessor cases and none of the multiprocessor cases. Because of the poor coverage, these diagnostics uncovered few design errors. In contrast, the initial version of the random tester required roughly one week to develop from scratch. It immediately uncovered numerous errors, including several significant design problems. The total design effort for the random tester, including the Monitor and Daemon, was about two person-months. The Monitor required most of this time, about 6 person-weeks, because it entailed a detailed examination of the *actual* implementation behavior. Simply writing the Monitor provided significant insight into the design, and lead to several high-level changes.

[2] UnOwned blocks are never written back because they are guaranteed to be read-only.

The random tester is different from many tests because it will run until it uncovers a problem, or forever if it can't find any. Initially, when the design was still buggy, the tester averaged about 20 minutes on a Sun-3/160 workstation before discovering an error (about 300 simulation cycles). Before submitting the chip for fabrication, it ran continuously for two weeks on multiple machines (using different random seeds), or roughly 3.5 million cycles.

To help debug errors found during long runs, we saved the last 1000-2000 cycles of each simulation using a tracing facility of the N.2 simulator. This trace generally provided enough data to diagnose the problem without further simulation. However, 15-20% of the errors required additional tracing or interactive debugging. Because pseudo-random number generators are deterministic, given the original seed, we could always reproduce the error.

A good test suite should *cover* all the important cases of the system under test. Unfortunately, while *fault coverage* is well understood [Kirk88], at least for combinational logic, *design verification coverage* is not. In fault coverage, the goal is to select chips that are free from fabrication errors; usually by testing all circuit nodes for stuck-at-0 and stuck-at-1 faults. In this case, the coverage is merely the fraction of nodes tested. Conversely, in design verification the goal is to verify that the circuit or functional specification implements the semantics defined by a higher-level specification. Because the possible state space is large and poorly defined, no definitive way exists to evaluate the coverage of these tests.

To estimate the coverage of the random tester, we used the N.2 coverage facility plus an intimate knowledge of the behavioral specification. The N.2 tool provided statement level coverage, recording the number of times each line of ISP' (N.2's C-like hardware description language) was executed. By examining long simulation runs (50,000 cycles, or 2 days) we determined operations excluded by the tester. We periodically added additional action/check pairs, and the I/O Daemon discussed above, to improve the coverage. After these improvements, we estimated that the random tester covered over 99% of the normal cases, and 75% of the exception

cases. Most of the important exception cases, such as reset, were tested by hand.

How long should the random tester run? Because the order of actions and checks is random, coverage of a particular case is probabilistic; the longer the simulation runs the greater the probability that the tester generates all the cases. For simple problems, such as the checkerboard example discussed earlier, we can compute precise confidence levels for the coverage. But for the random tester, simply enumerating the state space is too difficult. Taking a pragmatic approach, we ran the simulation as long as possible. During the peak simulation period, about 5-6 months, independent simulations ran on 14 Sun-3 workstations. Runs were automatically checked once a day, failed runs saved for analysis, and a new simulation started. Since the simulator executes about 1000 cycles per hour, we estimate that we simulated between 50 and 100 million cycles. Although the elapsed time of the simulation was quite long, the human time required to manage the simulations and debug the problems was comparatively small. This allowed us to complete the chip layout and most of the timing and switch-level simulation in parallel with functional simulation.

We kept detailed records on the bugs uncovered during simulation. We classified bugs by type: *Functional* bugs produced incorrect results, *Performance* bugs produced correct results

Table 1: Break-down of Functional Simulation Bugs

Type	Functional	Performance	N.2 Artifact	Test	Total	(%)
Random Tester	25	2	2	9	38	(55%)
Inspection	13	2	2	0	17	(25%)
Other	9	1	4	0	14	(20%)
Total (%)	47 (68%)	5 (7%)	8 (12%)	9 (13%)	69 (100%)	(100%)

slowly, *N.2* bugs were simulation artifacts, and *Test* bugs were errors in the test structure (e.g., the Monitor and Daemon). We also kept track of how the bug was detected: the random tester (Random), visual inspection and design reviews (Inspection), and other tests and diagnostics (Other). Table 1 summarizes the bug reports. The random tester uncovered over half of the functional bugs found in the design. More importantly, it detected a number of complex sequencing and interaction problems that we believe would not otherwise have been found until after fabrication. Although the Monitor was difficult to debug, accounting for most of the Test bugs and three-quarters of the tester design time, we feel it was invaluable. The Monitor detected 28% of the functional bugs found by random testing. A common symptom of a coherency problem was that a processor did not respond to requests from other processors when it should have. Without the Monitor, this event might not have been caught for many cycles. With it, these kinds of bugs were caught immediately, making them much easier to debug. Random testing does not preclude design reviews: nearly 30% of the functional bugs were found by careful inspection and design walk-throughs. Because many of these errors were found in exception cases, the random tester probably would not have detected them. Conversely, the Other tests found only 20% of the functional bugs, and the random tester would have found most, if not all, of them.

Figure 4 charts the number of bugs found per week; squares, crosses, triangles, and diamonds represent Functional, Performance, N.2, and Test bugs, respectively. The histogram illustrates the effects of increased coverage. Until the last few months of simulation, every extension to the action/check scripts uncovered a new burst of bugs; this can be seen in the "spikes" of activity. The spikes also correspond to increased levels of activity; the large gaps in the histogram occur when the designers were involved in other activities, and random testing stalled while waiting for bug fixes.

Except for the large spike at week 25, owing to a week long design review, this chart suggests a generally declining rate of bug detection. Figure 5 also shows this effect, plotting the cumulative number of bugs over time, displaying the sum of the data in Figure 4. The solid line

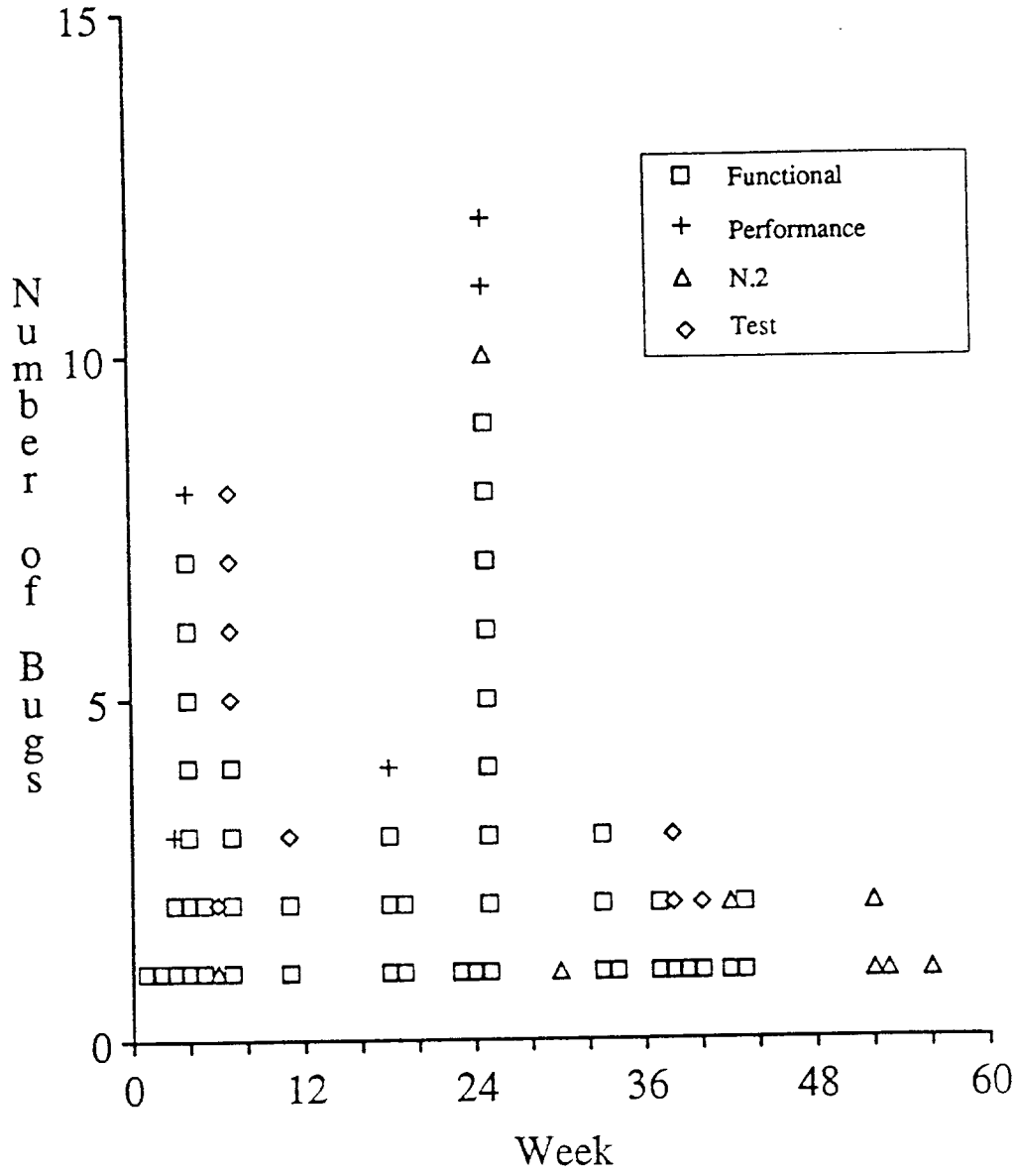

Figure 4 : Bugs Found Per Week

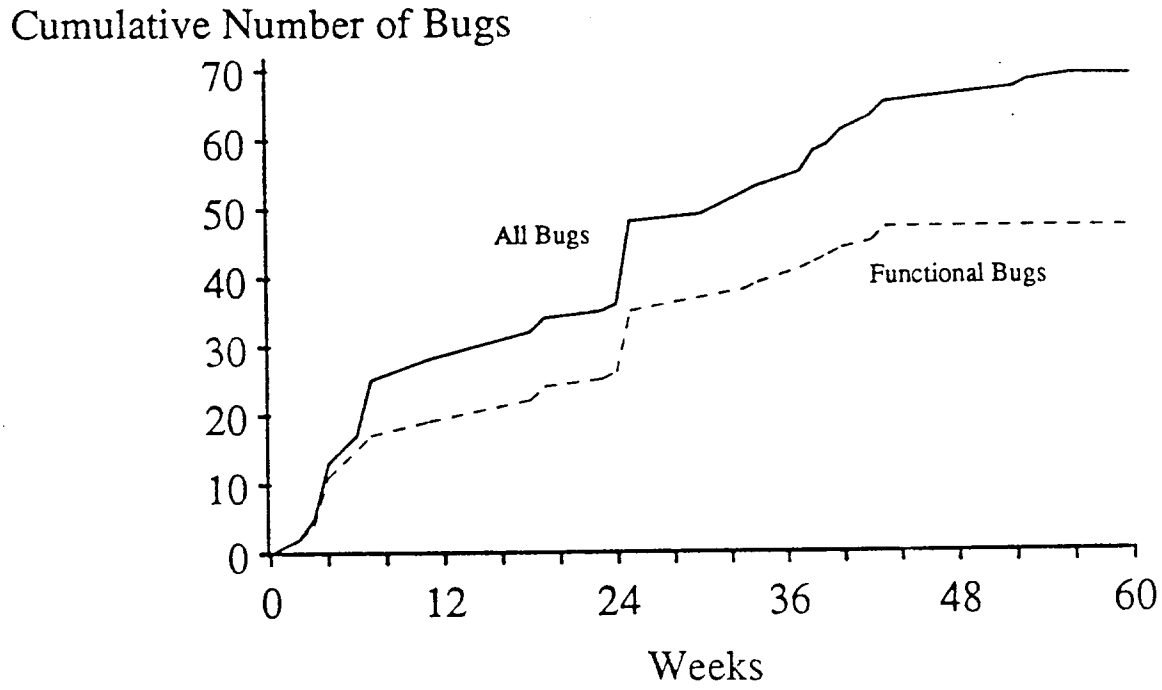

Figure 5: Cumulative Bug History

plots the all bugs, while the dashed line plots only the functional bugs. By the time we submitted the chip, in week 60, the rate of bugs being found dropped to zero. Despite stepped-up efforts to increase the coverage, we found no functional bugs during the last 4 months of simulation. As a result, we were confident that the chip would work correctly on first silicon.

5. Status and Conclusions

We submitted the cache controller for fabrication in November 1987, and received silicon in February 1988. We completed unit testing in April 1988, using test vectors extracted from the functional simulation. In May 1988, we completed uniprocessor hardware testing, with the CPU and cache controller running diagnostics together in a processor board. Multiprocessor testing, delayed by competing demands for resources and personnel, was completed in December 1988 after less than one month of effort. The operating system group began to port Sprite to SPUR

hardware in July 1988, and had a reliable uniprocessor system by September. As of January 1989, the multiprocessor version of Sprite runs reliably on a three processor Spur system.

During both uniprocessor and multiprocessor hardware testing, we used an assembly language version of the random tester. This version was based on the same principles, and the same action/check scripts, as the simulation version, but was written in assembly language so it could run on a real SPUR CPU chip rather than our simulation stub. This tester was very useful for finding functional errors on the processor board; however, it was much less effective at uncovering electrical problems caused by cross-talk and improper termination. Because the tester uses predetermined address and data values, hardwired into the action/check scripts, it rarely exposed these types of electrical noise problems. Random generation of address and data patterns, as we used in other diagnostics written specifically to isolate these problems, would improve the utility of the random tester for checking hardware.

A few minor design errors were uncovered during testing. One electrical problem, a floating well, resulted in a dynamic node, which could be refreshed periodically under operating system control. A timing problem prevented the uniprocessor system from running at design speed, fortunately it was possible to correct this error using a single PAL on the board. Finally, we found two problems in the CPU/Cache Controller interface. Although neither problem was serious since both were easily fixed on the board, they point to two important limitation in the methodology. First, the N.2 system does not properly model tri-state signals[3], allowing the cache controller to continue using a signal no longer driven by the CPU. While this signal retained its value indefinitely in the simulation, the signal decayed to logic 0 in the real hardware. Second, because a *stub* CPU drove the cache controller in the simulation, we did not verify the interface with the *real* CPU. If we had used an assembly language version of the tester, running on the real CPU model, we would have found this last problem. In addition, we would have uncovered

[3] ENDOT has addressed this problem since we wrote the simulation.

several problems in the CPU itself, which were not detected by their hand-coded diagnostics.

To summarize, cache coherency simplifies multiprocessor programming, but complicates the hardware design. These cache controller implementations require extensive simulation to verify the design, particularly for custom chips with their long fabrication cycles and poor visibility. But generating tests to cover all the multiprocessor test cases is extremely difficult. We have developed a random tester, that probabilistically generates all multiprocessor interaction cases and checks that they function correctly. This tester, requiring only two-person months to develop, uncovered 25 out of 47 functional bugs found during simulation. And although the elapsed simulation time was nearly 14 months, the simulations required little designer interaction, allowing chip design, switch-level simulation, and other activities to proceed in parallel. We found no serious bugs in the fabricated chips, and a three processor prototype system is now operational. We believe this is a strong case supporting random test generation, and that random testing is a powerful technique that can be applied to other problems as well.

Future work is needed to evaluate the benefits and limitations of random testing. The two most important questions are: "how long should it run?" and "has it covered all the important cases?" We think our coverage analysis was effective but ad hoc. A possible, more formal approach is *mutation analysis* [DeMi78], where one intentionally injects errors into the simulation, then evaluates how long the tester takes to find them. Application of mutation analysis to software engineering has shown some promising results. Some researchers are already trying to apply it to random testing [Maur88].

6. Acknowledgements

We would like to thank Susan Eggers, Jane Doughty, and Mark Hill for reviewing prior drafts of this paper. Walter Beach, Dimiti Lioupis, Susan Eggers and Bill Cox developed some of the other diagnostic programs, and helped convince us we needed a better approach. Deog-Kyoon Jeong, Susan Eggers and Walter Beach contributed to the design and implementation of

the cache controller. We would also like to thank all members of the SPUR group who tolerated 6 months of constant simulation on their workstations. This work was supported by SPUR/DARPA contract No. N00039-85-C-0269, the California MICRO program (together with Texas Instruments, National Semiconductor, Xerox, Honeywell, and Philips/Signetics), the IBM Predoctoral Fellowship program, and the Canadian NSERC Fellowship program.

7. References

[Arch86] Archibald, J. and J. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, Vol. 4, No. 4 , November 1986, pp. 273-298.

[Bell85] Bell, C. G., "Multis: a new class of multiprocessor computers", *Science*, Vol. 228 , April 26, 1985, pp. 462-467.

[Cens78] Censier, L. M. and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, Vol. 27, No. 12 , December 1978, pp. 1112-1118.

[DeMi78] DeMillo, R. A., R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", *IEEE Computer*, pp. 34-43, April 1978.

[Gibs88] Gibson, G. A., "SpurBus Specification", U.C. Berkeley, Technical Report No. UCB/Computer Science Dpt. 88/480, December 1988.

[Hill86] Hill, M. D., S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, "Design Decisions in SPUR", *Computer*, Vol. 19, No. 11 , November 1986, pp. 8-22.

[IEEE86] "NuBus -- A Simple 32-Bit Backplane Bus", P1196 Specification, Draft 2.0, IEEE Standards Board, December 1986.

[Katz85] Katz, R. H., S. J. Eggers, D. A. Wood, C. L. Perkins and R. G. Sheldon, "Implementing a Cache Consistency Protocol", *Proc. 12th International Symposium on Computer Architecture*, Boston, Mass. , pp. 276-283, June 1985.

[Kirk88] Kirkland, T. and M. R. Mercer, "Algorithms for Automatic Test Pattern Generation", *IEEE Design and Test of Computers*, Vol. 5, No. 3 , June 1988, pp. 43-55.

[Kong87] Kong, S., D. Wood, G. Gibson, R. Katz and D. Patterson, "Design Methodology of a VLSI Multiprocessor Workstation", *VLSI Systems Design*, Vol. VIII, No. 2 , February 1987, pp. 44-54.

[Maur88] Maurer, P. M., "Design Verification of the WE32106 Math Accelerator Unit", *IEEE Design and Test of Computers*, Vol. 5, No. 3 , June 1988, pp. 11-21.

[Oust88] Ousterhout, J. K., A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch, "The Sprite Network Operating System", *IEEE Computer*, Vol. 21, No. 2 , February 1988, pp. 23-36.

[Patt85] Patterson, D. A., "Reduced Instruction Set Computers", *Communications of the ACM*, Vol. 28, No. 1 , January, 1985, pp. 8-21.

[Rose83] Rose, C. W., G. M. Ordy and F. I. Parke, "N.mpc: A Retrospective", *Proceedings of the 20th Design Automation Conference*, Miami Beach, Florida , pp. 497-505, June, 1983.

[Shal87] Shalem, S. and I. Carmon, "Testing the Design of the NS32532 Microprocessor", *Proceedings IEEE International Conference on Computer Design*, pp. 181-184, October 1987.

[Smit82] Smith, A. J., "Cache Memories", *Computing Surveys*, Vol. 14, No. 3 , Sept. 1982, pp. 473-530.

[Wood86] Wood, D. A., S. J. Eggers, G. A. Gibson, M. D. Hill, J. M. Pendelton, S. A. Ritchie, G. S. Taylor, R. H. Katz and D. A. Patterson, "An In-Cache Address Translation Mechanism", *Proc. Thirteenth International Symposium on Computer Architecture*, Tokyo, Japan , pp. 358-365, June 1986.

[Wood87] Wood, D. A., S. Eggers and G. Gibson, "SPUR Memory System Architecture", Technical Report UCB/Computer Science Dpt. 87/394, University of California, Berkeley , December 1987.

[Zorn87] Zorn, B., P. Hilfinger, K. Ho and J. Larus, "SPUR Lisp: Design and Implementation", Technical Report UCB/Computer Science Dpt. 87/373, U.C. Berkeley, September 1987.