Pre-silicon Verification of the Alpha 21364 Microprocessor Error Handling System

Richard Lee and Benjamin Tsien
Compaq Computer Corporation
181 Lytton Avenue, Palo Alto, CA 94301
{richardl, tsien}@pa.dec.com

ABSTRACT

This paper presents the strategy used to verify the error logic in the Alpha 21364 microprocessor. Traditional pre-silicon strategies of focused testing or unit-level random testing yield limited results in finding complex bugs in the error handling logic of a microprocessor. This paper introduces a technique to simulate error conditions and their recovery in a global environment using random test stimulus closely approximating traffic found in a real system. A significant number of bugs were found using this technique. A majority of these bugs could not be uncovered using a simple random environment, or were counter-intuitive to focused test design.

1. INTRODUCTION

Reliability is an important feature in a microprocessor used for mission critical systems [1]. As microprocessor feature sizes decrease, the probability of logic errors due to alpha and non-terrestrial particle strikes increases. Experiments show that structures such as memory and register cells are the most susceptible to these transient errors and in high performance microprocessors, these are the structures that are most prominent. To ensure correct operation, error logic is designed into the micro-architecture to detect these transient failures and to correct the errors if possible. If a detected error cannot be corrected, the error logic must be designed to provide the correct information and status to enable graceful degradation of the system. The error handling system of a microprocessor, by its very nature, is only called upon on relatively rare occasions i.e. when an error occurs. An undetected error or poor error containment can lead to disastrous consequences if the error logic fails to operate correctly [2]. The correct verification of the microprocessor error handling logic is both important and complex.

Transient errors occur asynchronously to the normal operation of the microprocessor. The correct verification of the error handling logic must comprehend the cross product of the microprocessor machine states and its potential error conditions. Additionally, error handling requires complex interactions between hardware and software at a global level; assumptions made at the interface level are prone to misinterpretation.

Traditional verification methods include focused and random testing [3-5]. Focused testing is effective for validating well-documented and understood areas. Error handling verification, however, requires the ability to generate complex interactions that may not be intuitive. Thus, focused tests are ineffective here because of the size and complexity of the state space. Random testing is better suited when a test space is large. However, if complex interactions between units cannot be identified, then simple random testing cannot be used to target interesting states. The limitations of the traditional methods described above indicate that these techniques cannot be relied upon as the sole means of finding complex bugs in error handling logic.

This paper introduces a global random error environment to meet the challenge of error handling verification. To achieve complex interactions between units, the approach allows error injection and recovery as well as re-injection of uncorrectable errors. Correctable errors can be continuously injected anywhere in the system, at any time. The environment also allows injected errors into a system running with realistic self-checking stimulus. When executed in conjunction with correctable and uncorrectable errors, this approach provides maximum test coverage. The results show that many of the bugs found using this approach could not have been found or would have taken much more effort to find using focused or simple random testing.

2. BACKGROUND

The Alpha 21364 microprocessor [6] is the fourth generation of the Alpha microprocessor family. The system-on-a-chip design of the Alpha 21364 microprocessor includes two levels of cache memory, a memory controller, an interprocessor router, and the Alpha 21264 core [7]. The Alpha 21364 design lends itself to the design of glueless, scalable, shared memory multiprocessor systems. A complete set of error handling features is incorporated into the Alpha 21364 micro-architecture including extensive parity and ECC protection on interfaces and memory structures. Error conditions are recorded to a comprehensive set of configuration status registers (CSR's) which are used by error handlers to react accordingly.

3. CHALLENGES AND SOLUTIONS

There are two major challenges to comprehensive global error testing: exploding states and inter-unit collaboration.

3.1 Exploding States

An error condition occurs asynchronously within a system. For example, an alpha particle can strike a memory cell and flip its value, or signals within an external cable can be affected by noise. Errors such as these are correctable on the fly , hence their effect on the rest of the system can be confined. A more serious

error is one that can be detected but whose original state is lost. Here, the error condition must be contained to minimize its effect on the rest of the system. The containment mechanism adds many more states to the design because the handling of such errors requires accounting for the cross products of all the states in the system. Verification of the error logic is a non-trivial task because events occur relative to one another both logically and temporally. This cross product of all the state machines in the "normal" flow of the design, coupled with the additional states introduced by the detection of errors, results in an explosion of the state space under error conditions.

This large number of states cannot be easily identified nor exhaustively tested through focused tests. What is needed is a comprehensive random environment to supplement focused testing. This random environment must support the injection of random error types at random times. Furthermore, because of the large number of cases, this support must be done with minimal overhead.

3.2 Inter-unit Collaboration

An error detected by a logic block or unit may need to be communicated to other units. The consumers of corrupted data need information about these errors so they can either correct them or attempt to minimize their effects. Bad data needs to be isolated from good data so that the error can be contained and the impact on the rest of the system minimized. This technique requires complex interactions among different hardware and software components. A specific error verified to be handled correctly within a unit does not mean that an error will be correctly handled across all the affected units within a system. An error needs to be verified to be correctly handled as it progresses from unit to unit and from processor to processor, leaving a "trail" in the form of error status registers and interrupts.

4. REQUIREMENTS

To meet the challenges identified above, a global random error-testing environment was implemented. The requirements necessary for this environment are as follows:

- Correctable errors – Correctable errors are errors detected and then corrected by the hardware and generally do not involve intervention outside of the unit. In the error test environment, these are allowed to happen repeatedly in the background, anywhere in the system.
- Uncorrectable errors – An uncorrectable error is a detected error that cannot be corrected by hardware. The challenge is to control the side effect of the error and allow the system to recover. Recovery from an uncorrectable error is the best way to make sure that an erroneous side effect did not happen. With a full understanding of the error mechanisms of each uncorrectable error, the system can be brought to an error-free state. This level of recovery requires a strategy for error injection time and locations, error containment, and post error clean up and recovery.
- Error rate – Simulating on a pre-silicon model is slow. The test environment should have minimal overhead in error injection and recovery. Additionally, because the system being tested is a multiprocessor, the environment should allow multiple errors to happen simultaneously as long as they do not interfere with each other. This constraint is needed to correctly verify the isolation of these errors.
- Self-checking – All test cases should be self-checking to require minimal human interaction. This requirement means self-checking on the error status as well as the data involved with an error, based on the analysis of error effects.
- Realistic stimulus – The stimulus needs to approximate the traffic on a real system while an error is being injected.
- Engineering effort – The effort needed to verify error handling should be minimal. Thus, any solution must be simple as well as effective.

5. IMPLEMENTATION

A global random error-testing environment was created from the requirements listed above. This environment consists of adding error-handling capacity to an existing environment.

5.1 RAMP

For the purpose of doing multiprocessor (MP) verification of the Alpha 21364, we had earlier developed an environment to allow running various sharing algorithms, similar to [8]. This environment, the Random MP (RAMP) Exerciser, allows these MP algorithms to run under a mini-OS environment in a distributed full-chip model. The RAMP OS is made up of a set of privileged architecture library (PAL) [9] calls along with a boot up reset sequence. The PAL calls range from context switching to translation buffer (TB) miss handlers. The MP algorithms, also referred as RAMP *tasks*, can be written in standard C or assembly programming languages and are independent of the simulation configuration.

The novelty of RAMP is the interaction between the OS and the tasks. Prior to simulation, scripts can be used to randomize the number of tasks and their placement within the simulation topology. This task information, consisting of an instantiation count and a unique id for each instantiation, is compiled into the OS and communicated to the tasks during start up. Similarly, a shared memory region is also defined and communicated to the tasks. For example, during the startup of a false-sharing task, we simply index the id into the shared memory region.

We refer to the tasks participating in one sharing event as a *group* of tasks. Under RAMP, we are allowed multiple groups of tasks in the same simulation. For example, we can participate in false sharing on a set of processors while doing a fetch-increment lock test on another. We can even have multiple groups of false sharing on different shared memory regions. To make multiple grouping possible, the RAMP OS performs a lightweight virtual to physical address translation based on mathematical functions. This mapping allows tasks to be written in a fixed virtual address without concerns that they overlap with another task, as in a real OS. Likewise, the configuring of RAMP groups can be done by a randomizer script and compiled in with the OS. With an OS and a library of tasks, the RAMP environment can generate an enormous amount of different test stimuli under varying configurations.

We decided to leverage the existing RAMP environment. First of all, running under an OS simplifies the task construction, as the OS already handles many of basic functions. Since RAMP already has a sizable library of tasks, grouping of these tasks allows varying background traffic in addition to the task, creating

a more interesting environment to inject errors. Self-checking on existing RAMP tasks also helps to identify situations where errors have not been contained. Finally, RAMP is already a global test environment with PAL calls and exception handlers. Most error handling should simply be expanding these handlers for error conditions.

5.2 Extensions to the RAMP Environment

The work to extend RAMP into an error verification environment was undertaken on three fronts: error task, OS support, and error injector demons, to best realize the list of requirements.

5.2.1 Error Task

Interesting errors involve a group of processors sharing a piece of data that is corrupted somewhere in the course of execution. This error may then propagate through the system affecting other processors. Processors sharing the data may each enter interrupt or machine check handlers when they reference the affected data. The most natural solution is to create error task instances, that when run in a group under RAMP, can generate sharing on the error data, check the effects caused by these errors, and take the appropriate actions to clean up afterwards. Placing error handling in a task allows us to leverage the most out of the RAMP environment. As with any other task in the RAMP environment, we can mix and match different tasks, thereby creating interesting background traffic for resources that are being shared. However, in order to mix error tasks with other tasks or even other instances of the same error task, we must make sure we can isolate the errors to the group of tasks to which they belong. This means an error group must inject its errors to known addresses so nothing belonging to other tasks will be affected. Since RAMP does address translation, a fixed virtual address within the error test will be mapped to different physical addresses in different groups. Also, there is a mechanism between error groups and injectors to prevent multiple groups from injecting errors to the same resource in the system before each has had a chance to handle them. Doing so corrupts error registers and makes it hard to identify the owners of each error. Extensive self-checking on the error data guarantees errors are handled and cleaned up properly. Self-checking on non-error tasks guarantees errors do not incorrectly propagate. Thus, placing error handling in a user task allows us to achieve multiple uncorrectable errors in a system running under realistic MP traffic while doing self-checking.

5.2.2 OS Extensions

Under the RAMP environment, we added interrupt and machine check handlers. In a real system, these handlers are expected to evaluate the error status reported by error registers and decide to correct them, kill the process affected by the error, or in the worst case, gracefully shutdown the system. In our verification-centric approach, the OS simply records the relevant errors into a queue as they occur in the handlers so the error tasks may use that information for checking. The OS also performs self-checking on other errors that should never occur due to the design of our error environment. One such example is an unexpected second error occurring before the first one is handled. Finally, the OS clears the error registers so additional error events can trickle in and be recorded. This design allows us to test under multiple error groups, because the OS does not need any knowledge regarding which errors belong to which group. The tasks themselves claim the errors that belong to them. The OS, of course, needs to verify that all the errors in a system were claimed at the end of a simulation run.

The OS also provides a set of PAL calls to clean up errors. Upon interpretation of the error information, the tasks decide on the appropriate actions to repair the errors. It is important not to indiscriminately repair every possible thing that can go wrong. If we only repair the errors reported by the error registers, we can easily detect if something unexpected has also happened. In addition, fixing only what is necessary is least expensive (cycle-wise) in terms of simulation.

5.2.3 Error Demons

Demons [5] are programs that act as I/O devices within the RTL. As virtual I/O devices, they can be accessed from the tasks and they can also access memory during a simulation. Because they are compiled into the RTL, they have access to all of the RTL signals. Error injection is a perfect role for the demons. Demons are controlled by code running on the Alpha 21364 core through I/O reads and writes to a set of predefined control registers. I/O operations provide the ideal means for code running on one model to communicate to demons on any model in a distributed MP simulation environment. Communication is done through the built-in coherency protocol and requires no additional communication channel between different models. In addition, I/O operations are non-cacheable and have their own routing channels. This approach has a minimal effect on the chip and is very unlikely to be affected by error conditions caused by the test stimulus.

We divided the task of error injection to be handled by multiple demons. There is an error demon for each architectural unit in the design. The division is in place since some errors may not need to involve the whole chip. In reality, the interesting errors involve units from separate processors. Therefore, from a single-processor point of view, it is perfectly normal for different units to be involved in handling different errors at the same time without overwriting status registers. For example, an uncorrectable data error may occur in the cache due to a request from the local Alpha 21364 core, while another group of tasks injects an error at the memory unit that is processing a remote request. These errors should be recorded in their respective status registers without interfering each other. Therefore, having multiple, separately programmable demons simplifies the task of tracking injection states within a unit and allows these demons to simultaneously inject unrelated errors at their discretion.

To allow a more random error stimulus, we allow multiple groups of error tasks to configure a demon. Each task can register with a demon, which returns an address offset that the task uses to control the demon separately from other tasks. Each task can separately request error injections and change weights between the types of uncorrectable errors independently from others. Of course, a demon would only inject an uncorrectable error for one task at a time, until their errors are handled, so it does not pollute the error status registers. Also, error checking is made more effective by allowing multiple users of a demon. From the standpoint of a group of tasks, an error not recovered correctly will be blatantly obvious to another group.

In the overall scheme, we choose to implement as much intelligence into the demons as we can. The demons randomly choose between different types of uncorrectable errors while the

tasks can provide minimal guidance in terms of weights. In addition, demons also contain behavioral models predicting the effects of the errors upon the status registers. That information is used by the tasks to compare against what is read from the status registers by the OS during handler routines. In this sense, the analysis of the error conditions is done by the demons, which run magnitudes faster than the same code on the model. We also reduce the engineering effort because we are better able to debug our code running in the behavioral model in isolation. Engineering effort considerations were also a part of our decision to implement most of the runtime error checking and handling in tasks, in addition to the reasons in the above sections. Code written in C provides much better debuggability than assembly.

5.3 Process

5.3.1 Task Initialization

When a task is started by the OS, it receives an id within its group. One task will be the master task, controlling the demons and checking the errors, while others participate in generating interesting traffic on the error address. A shared memory location is reserved for error injection. The demon requires two addresses, one to inject the errors and another to report them. It is initialized by an OS PAL call, which translates the addresses from virtual in the task to physical in the demon. In addition, a virtual mapping of the demon control address is returned to the caller task. Although member tasks in a group are not started simultaneously by the OS, the OS has an option to guarantee that the tasks will be running simultaneously at some time. In this case, we perform a code barrier to make sure all tasks are synchronized at the same point in the code before we start any error testing.

5.3.2 Task Execution

At the start of each loop of execution, we randomly pick a demon and assert readiness by writing to its control address. While awaiting injection, we perform self-checking on the error address. Following is a simplified version of the code:

```
main() {
…
i = 0;
while(!error_injection_poll) {
  j = byte_false_share();
  if (i != j) { //self-checked
    evict_cache_block(error_injection_poll);
    memory_barrier();
    if (!error_injection_poll)
      break;
    else
      exit(1); // self-check error
    i = i + 1 & 0xff; //generate new check value
  }
}
memory_barrier();
evict_cache_block(error_address);
code_barrier(my_group_id);
…

int byte_false_share() { //random FS variations
  if (random < XXX) { //simple load-store
    val = error_address[my_group_id];
    error_address[my_group_id] = val + 1 & 0xff;
  } else if (random < YYY) { //load-evict-store
    val = error_address[my_group_id];
    evict_cache_block(error_address);
    error_address[my_group_id] = val + 1 & 0xff;
  } else // other variations…
…
  return val;
}
```

In the main code, all tasks perform false sharing while awaiting the demon to inform error injection through a memory write to the polling address. When that occurs, all members of a group stop sharing on the data to perform recovery. However, because memory latency is variable in an Alpha 21364 system, an error may have already been observed by a task resulting in bad data that fails self-checking. Under that situation, the polling address data is evicted and re-requested after doing a memory barrier. A memory barrier guarantees any outstanding requests to the error address have been resolved. Upon receiving a response for the second poll we can determine whether the self-check had been legitimate or we had just gotten corrupted data.

The running task is not aware of the error recording done by the OS behind the scenes. The Alpha 21364 core may jump back and forth between interrupt handlers, machine check handlers, and error task code before all the errors are recorded. To guarantee all the errors are recorded on one processor, a memory barrier is performed. The tasks then perform a code barrier to synchronize all the members, guaranteeing all errors on all member processors are recorded before proceeding to handle the error.

The master task walks through the error queues on the affected processors depending on the type of error. Errors in the queues are claimed if they match the address and checked against the demon's expectation. The demons also supply information on what actions need to be taken to recover from the errors, such as the necessity to reinitialize the directory state for a memory block. These actions are performed by the master task to allow another injection. Also during recovery, the master task notifies the demon that all errors have been collected, so the demon can inject errors for another ready group of tasks.

5.4 Examples of Errors

Error injection to different areas of the chip has diverse consequences on the types of error conditions and the number and composition of processors observing errors. This section describes some of these complex sequences that are achievable and verifiable though global random error testing.

5.4.1 Cache Tag Uncorrectable Error

The Alpha 21364 has a large L2 set-associative cache. A tag error is detected when a lookup to an index results in a way with a tag error. If the Alpha 21364 core makes a request to that index, we would receive a machine check. However, if the error is detected during a remote probe, the remote requestor would also receive a machine check. In addition, as the coherency of the block is compromised by the corruption of the tag, the directory is eventually made aware of the error, which causes a memory controller error that triggers an interrupt.

The demon would report the type of transaction that encountered the error along with its error status. The task is intelligent enough to check the status recorded by the CSRs against the demon. In addition, the effects of additional errors are taken into consideration, and the error queues of all members in the group, along with the memory unit of the processor containing the shared memory, are examined for error occurrences belonging to this address.

5.4.2 Router Uncorrectable Error

An uncorrectable error in the data section of a packet causes a router error in the node where it is detected and triggers an

interrupt. The packet is converted to a known pattern and transmitted to the destination, where it causes a machine check. The error pattern itself does not cause further router errors. It is important to be able to inject errors in any node along the path a message packet takes between source and destination. The Alpha 21364 uses adaptive routing so pre-arranging injection at a node is impossible because the path of a packet is unpredictable. However, a useful attribute of the Alpha 21364 router protocol is that packets always take the shortest path from source to destination. Random injection of router errors is achieved by randomizing the number of hops before an error is injected. Also, to track the packet before injection, a spare bit in the header is used to indicate a special packet. In addition, the data is replaced with the injection hop count and the polling address of the error group. The router demon at each hop decrements the counter and when it reaches zero, we inject the error. During injection, the data portion of the packet is randomized with a combination of correctable and uncorrectable errors as well as non-error ticks. The injector router also communicates the error information along with its CPU id to the error group. The master task doing the recovery would check the injector processor along with the requestor processor for errors. It also tolerates any other group members that have received bad responses from forwards. Any extraneous errors elsewhere that are not claimed will eventually self-check by another group or be caught by checking done at the end of the simulation.

5.5 Difficulties and Workarounds

Sometimes, handling certain errors requires all the processors in the system to be in a recovery mode. For example, in some cases, the hardware requires that memory traffic be quieted to a single node. Synchronizing processors for recovery is time consuming and may not even be necessary if there is no sharing at that node. We devised a method, through simulation tricks, to isolate such node while allowing other unrelated nodes to continue operating. Studying the Alpha 21364 coherency protocol in detail, one can infer that certain messages create other messages. For example, requests can result in forwards sent by the directory controller. These forwards then result in acknowledgements back to the directory node. Acquiring a block exclusive by a requestor can also result in victims being sent back to the directory node. To isolate a node, any new requests destined to the node are first buffered up. A demon control register is polled until the directory controller is empty of all outstanding requests. Then, any new victims become buffered until the demon determines that all existing victims are drained. During this time, the router's inter-processor traffic handling is not affected. With the local processor also sleeping, it is easy to clean up the memory location and reintegrate the node.

6. RESULTS

We believe we had good success with the global random error approach. The types of bugs we found with this test method are much more complex than we had previously encountered using either focused testing or unit level random testing.

6.1 Analysis

Figure 1 shows the number of error-related bug counts attributed to each verification method used to verify the error logic on the Alpha 21364 at the time we started to use global random error testing. Prior to and during the early introduction of global testing, the bug rate was high and traditional methods of focused and random testing yielded significant number of bugs. However, as time progressed towards the end of the project, global testing comprised a larger share of the sum.

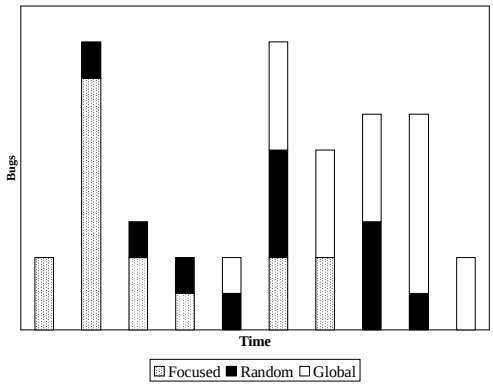

Figure 1: Error bug breakdown according to test stimulus

While focused and random testing tapered off due to their limitations, global testing continued to find more bugs. This was in part due to the thoroughness of the method. Many of the bugs were simply too complex for focused or random tests to find.

Figure 2 classifies the bugs found using the global random error testing whether they could also have been found through other means. We classify a bug as "global only" if:

1) A focused test around the area of the bug is too complex or unintuitive to be designed by a verification engineer with a reasonable effort.
2) A random exerciser does not cover the cross product of the bug due to unexpected complex interactions.

From our analysis, we believe the majority of the bugs found by the global environment could not reasonably have been found by any other test stimulus. Furthermore, in doing similar analysis, we found global testing should also be effective in finding many of the bugs found using other techniques.

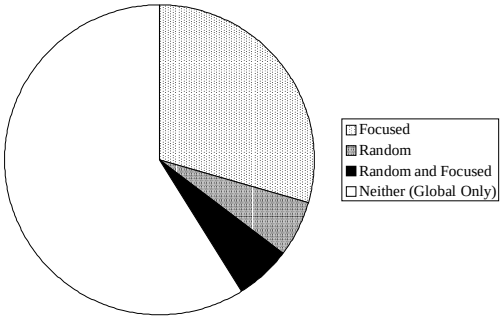

Figure 2: Classification of bugs found in global testing

Table 1 compares the effectiveness of each environment in finding complex bugs. We define complex bugs to be bugs that either involve multiple units on different processors or require timing sensitive cross products. Also, we are more stringent in classifying bugs involving correctable errors as complex, because they do not usually require recovery and can be injected continuously on any random simulation models.

Focused	4%
Random	13%
Global	59%

Table 1: Percent "complex" bugs in each environment

This table shows that only 4% of the bugs found by focused testing were considered complex and indicates that focused testing generally catches bugs in error cases that are usually well conceived and documented. Random testing may find some complex cross products that escape focused testing. The majority of the bugs found using global testing are considered complex.

6.2 Examples of Error Bugs

The analysis of error bugs shows the bugs found using global random error testing are more complex than those found through other methods. The following are examples of such bugs.

6.2.1 Multiple Victims in a Global System

In this case, a method to recover from an error in one unit causes an input pattern to violate the protocol assumptions made by another unit. Normally, only one victim to a block can exist in the system at any given time. To victimize implies that one processor had exclusive ownership of the block. It is logical to make the assumption that the directory controller should only expect to see one victim when it is handling a transaction. This was a valid assumption until we started testing the error logic. During the recovery of a cache tag error, the accepted handler sequence would turn off error protection and perform a sequence of loads to clear out the tags of all ways at an index, thereby replacing the bad tag. However, should a tag error corrupt the status of a way to exclusive, we would unintentionally send a victim out to the directory controller of the block. If we are in the process of handling a transaction to the same block at the directory controller and a legitimate victim has merged to the transaction, a second (bad) victim merge to the same transaction results in a resource leak with the buffer holding the data for the first victim. This case is important because if the node causing the incorrect victim were in a different partition from the node containing the protocol logic, an error in one partition may affect a different partition, resulting in error containment issues. It is interesting to note that we rethought and chose a different strategy for tag error recovery as we fixed the bug.

6.2.2 Low Probability Event Involving Directory Controller

When the owner of a block detects uncorrectable ECC errors in the data it wants to victimize, it signals that information when sending the victim packet. The directory controller that handles the victim would tag the directory state to a special state indicating the block is incoherent. In this bug, there was a request from a different node arriving first at the directory controller at around the same time. A request is always snooped at the local processor. In this case, our snoop just missed the specially tagged victim and replied an "in-flight" status to the directory controller. Under normal conditions, we always read the directory state first, drain the victim, and then respond to the requestor. However, the background traffic in the directory controller exaggerates the delay in the directory read so the victim write to the memory occurs earlier. All of the above conditions: error victim, snoop miss, or early victim write, are rare events and when combined together, produced a bug that resulted in no response being sent to the requestor. The pre-silicon environment allowed us to debug and fix this bug within hours.

7. CONCLUSIONS

With the increasing market requirements for reliable, high-performance systems, high reliability has become an extremely important feature of large-scale SMP systems. Testing the entire system's ability to handle major and minor errors deterministically is a complex task because it involves an exploding number of states and interactions among multiple units and multiple processors. The ability to test complex global errors pre-silicon saves major efforts post-silicon.

Our global error testing environment based on RAMP uses random error injection to cover as many states as possible while the combination of OS, user tasks, and I/O demons create complex inter-processor traffic patterns during which errors are injected and handled. Many of the bugs found using this new environment are complex bugs that could not have been found with unit-level random testing or focused testing. We believe we satisfied our goal of creating realistic error stimulus in a global MP environment to allow us to test error conditions in quick succession, using minimal engineering effort. Finally, we believe that we have raised the confidence in the integrity of the Alpha 21364 microprocessor through testing error conditions in a global environment.

8. ACKNOWLEDGEMENTS

We wish to thank Scott Kreider and John Fu, for their encouragement and support; Jason Mancini for his work in the memory error injector code; Jonathan Nall for the implementation to buffer inter-processor traffic during recovery of a node. Appreciation is also extended to members of the design team for their feedback.

9. REFERENCES

[1] "Data Integrity Concepts, Features, and Technology," Compaq White Paper, 1999.

[2] E. Jenn et al, "Fault Injection into VHDL Models: The MEFISTO Tools," in Proceedings of the 24th International Symposium on Fault Tolerant Computing, pp. 66-75, 1994.

[3] M.Kantrowitz and L.M. Noack, Functional Verification of a Multi-issue, Pipelined, Superscalar Alpha-Processor – the Alpha 21164 CPU Chip. Digital Technical Journal, 7(1):136-144, August 1995.

[4] M. Bass, T.W. Blanchard, D.D. Josephson, D. Weir, and D.L. Halperin, Design Methodologies for the PA 7100LC Microprocessor, Hewlett-Packard Journal, 46(2):23-25, April 1995.

[5] S. Taylor et al, "Functional Verification of a Multi-issue, Out-of-Order, Superscalar Alpha Processor – The DEC Alpha 21264 Microprocessor," in 35th Design Automation Conference Proceedings, 1998.

[6] P. Bannon, "Alpha 21364: A Scalable Single-chip SMP," in Proceedings from Microprocessor Forum October 1998, http://www.alphapowered.com/alpha_tech_presents.html.

[7] Linley Gwennap, "Digital 21264 Sets New Standard," Microprocessor Report, pp. 11-16, 1996.

[8] A. Hosseini, D. Mavroidis, and P. Konas, "Code generation and analysis for the functional verification of microprocessors," in 33rd Design Automation Conference Proceedings, 1996.

[9] Richard L. Sites, *Alpha Architecture Reference Manual*, Digital Press, Burlington, MA, 1992.