A Simulation-Based Method for the Verification of Shared Memory in Multiprocessor Systems

Scott Taylor[1], Carl Ramey[2], Craig Barner[3], David Asher[3]

Compaq Computer Corporation

Abstract

As processor architectural complexity increases, greater effort must be focused on functional verification of the chip as a component of the system. Multiprocessor verification presents a particular challenge in terms of both difficulty and importance. While formal methods have made significant progress in the validation of coherence protocols, these methods are not always practical to apply to the structural implementation of a complex microprocessor. This paper describes a simulation-based approach to modeling and checking the shared-memory properties of the Alpha architecture by using a directed acyclic graph to represent memory-access orderings. The resulting tool is integrated with a simulation model of an Alpha implementation, allowing the user to verify aspects of the implementation with respect to the overall architectural specification. Both an implementation-independent and an implementation-specific version of the tool are discussed.

Keywords

Microprocessor, Functional Verification, Shared Memory, Checking

1. Introduction

Multiprocessor (MP) systems are becoming very popular in high-performance technical computing, where massively parallel applications can make use of many processors in a single system box. For example, the EV7 processor can support up to 128 processors in a single system configuration [1].

1.1 The Alpha Shared Memory Model

In a shared-memory multiprocessor system, many programs on many processors may be contending for access to a single memory location. When multiple programs access the same memory location, the value of a location read by a CPU may not match the last value written by that CPU. The specification of the Alpha architecture contains specific rules for cache coherency, data sharing, atomic update mechanisms, memory ordering considerations, visibility rules for data as seen by other processors, read/write ordering, etc. [2]

This specification allows programmers to write shared-memory programs for any implementation of the architecture, while keeping processors free to implement efficient memory systems [2]. The architecture is not required to maintain strict program order of all memory accesses. On an Alpha processor, the following sequence is perfectly legal (**Example 1-1**):

Assume memory is initially 0 for all locations, and that address X != address Y.

Example 1-1:

CPU-0

ST 2 à X (store, value=2 to address X)

ST 2 à Y (store, value=2 to address Y)

CPU-1

LD Y ß 2 (load from address Y, value received was 2)

LD X ß 0 (load from address X, value received was initial 0)

Alpha allows the two stores in the above example to be reordered in the memory system. If the programmer wants the ordering to be maintained, they must explicitly order the references via a "memory barrier" instruction. A memory barrier guarantees that all subsequent loads or stores will not access memory until after all previous loads and stores have accessed memory, as observed by other processors.

One of the most interesting reorderings allowed by Alpha is the ST-LD program order of accesses on a single CPU. The Alpha specification allows the following memory accesses to be reordered (**Example 1-2**):

Example 1-2:

CPU-0

ST 2 à X

LD X ß 2

[1] Now employed by Intel Corporation

[2] Now employed by Stargen, Inc.

[3] Now employed by Cavium Networks

Even though the LD in this example can be ordered[†] before the ST, it must "see" the value written by the ST. (Consider the programming implications were this not true.) By allowing these accesses to be reordered, a processor can tolerate LDs consuming data from write buffers before that write has been made available to other processors. The property that causes the LD to "see" the ST and allows programmers to write meaningful programs is called "visibility".

The Alpha architectural reference manual lists a formal set of properties and rules for accessing shared-memory regions that provides both constraints for hardware and expectations for software.

1.2 Implications of Multiprocessor Architectures

Shared-memory multiprocessing presents a difficult problem to the verification engineer. One of the realities of functional verification is that formal verification methods are not yet capable of handling a full design. Thus, we must still rely upon simulation for the bulk of our verification. Often, the memory-ordering rules for an MP architecture can lead to complex interactions, which can further lead to post-silicon bugs in an MP system [3].

For instance, consider a case in which multiple processors are reading and writing a single memory location (such as an operating system semaphore). It is not difficult to picture a bug in which multiple processors think they own the block, or where an atomic update fails and store-data from a processor is lost. (Events in **Example 1-3** are time-ordered [A], [B], etc.)

Example 1-3:

CPU-0

[A] LD X ß 0	(Atomic load operation locks address X)
[C] ST X à 1	(Increment and atomic store, which passes)

CPU-1

[B] LD X ß 0	(Atomic load operation locks address X)
[D] ST X à 1	(Increment and atomic store. Store SHOULD fail due to [C] in the absence of a bug)

1.3 Current Approaches to Multiprocessor Verification

Traditionally, companies have verified MP-capable CPUs by instantiating a single instance of the CPU and using a complex test bench to mimic the behavior of all other processors in the system [4,5,6,7]. However, the complexity of large MP configurations make such a test bench difficult to implement and increases the risk that valid sequences of events in an actual system are not covered by the test bench. And while formal methods have made significant progress in the validation of coherence protocols [11-13], these methods are not always practical for the *implementation* of a protocol within a complex microprocessor.

This led the EV7 team to try an additional approach—performing MP verification by instantiating multiple instances of the RTL (Register Transfer Level) code and designing a new simulation checker to detect memory coherency/consistency violations. This gives the benefit of realistic inter-processor communication, but at the cost of simulation size and performance.

It also became clear that the complex cross products of memory accesses in a ccNUMA architecture such as EV7, DASH [8], and SGI's ORIGIN2000 [9] would require better coherency checking techniques to cover correctness, deadlock/starvation avoidance, races, and other special events in the memory system.

Many previous Alpha simulation environments checked results using a generalized reference model for each CPU in the system. This model was typically connected to a simple array-like memory, similar to **Figure 1**. The reference model executes each instruction and updates all architectural state (such as register values, program counter, and memory).

The reference model's memory behaves as a single-ported array to which the reference model can read and write data. If a location is written with a value, that value is returned on the next read. But this configuration is ineffective in a system that contains multiple processors writing the same memory location. The value read on a CPU is not necessarily the last value written from that CPU.

A simple solution to this problem is to treat all shared memory in a simulation as unpredictable. The reference model would then "trust" all values that were loaded from regions of memory in which other processors are allowed to write. All non-memory instructions would then have the benefit of complete runtime checking because both the reference model and the simulator would be executing each instruction using the same register contents.

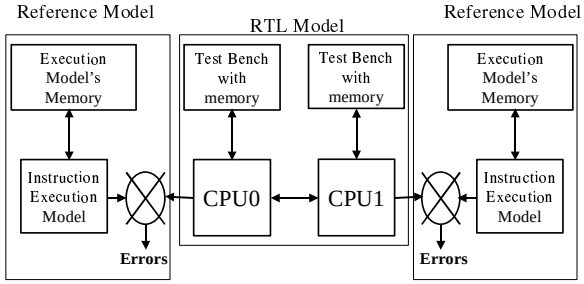

Figure 1

[†] Here, "ordered" refers to the order in which other processors in the system "see" these memory accesses.

The "unpredictable memory" solution clearly leaves opportunities for bugs to go undetected. For example, if no other processors happened to write the location then the value read *should be* the same as the value written. In order to check that the data read from a shared-memory location is legal within the shared-memory specification, the reference model must use some type of system-wide shared-memory model.

Ideally, the simulation environment could contain a single memory with which all reference models would communicate. But, when reference models do not *exactly* match the timing of the simulated RTL, memory ordering at the reference level will not match the ordering at the RTL level. In these cases, it is not possible to simply connect all reference models to a single array-like memory: Even if CPU-0 completes a store in its reference model before CPU1 completes a store in its reference model, the memory order of those stores in the RTL cannot be predicted, and the final value of that memory location is not simply the last store completed by the reference model.

The reference memory problem can be broken into three distinct categories: memory data storage, memory data checking, and protocol checking. The latter was determined to be a critical verification bottleneck, so a new tool was developed to address this issue. As you will see in the next section, the first two issues can be trivially solved within the framework of the protocol checker.

2. The Memory Order Model

2.1 Concept

The Memory Order Model (MOM) allows the results of shared-memory simulations to be checked against the Alpha shared-memory specification. Rather than shared memory being treated as unpredictable, MOM can dynamically decide whether the data returned for a load is legal.

MOM provides results checking not just for data integrity, but also for ordering violations. Consider **Example 2-1**:

Example 2-1:

CPU-0	CPU-1
ST 2 à X	ST 3 à X
LD X ß 3	LD X ß ??

Given that the LD on CPU-0 returned the result from the ST on CPU-1, we know that the ST on CPU-1 must have occurred after the ST on CPU-0 (regardless of the order of reference model completion). Since time must move forward, the LD on CPU-1 must receive a 3 as well. In other words, the LD on CPU-0 created an ordering between the two STs. This ordering determined that "3" is the only legal value for the LD on CPU-1.

A MOM-based simulation environment replaces all the reference models' memories with a centralized shared-memory model (see **Figure 2**). This centralized memory verifies a memory access's validity against the Alpha shared-memory specification.

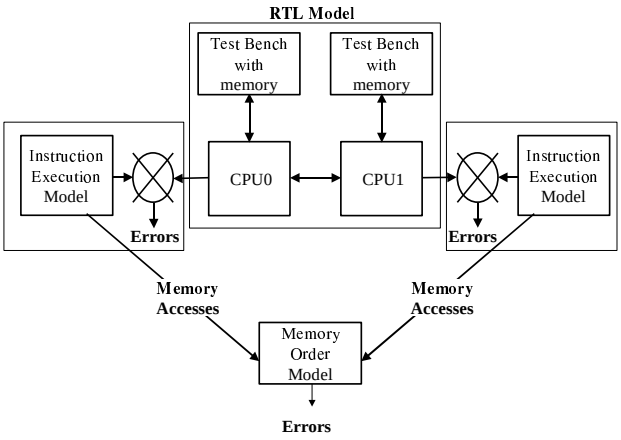

Figure 2

Traditional reference model based simulations use a reference memory system only to provide data. If that data doesn't match the data in the simulated processor, an error is reported. In a shared-memory environment, there is often more than one "correct" answer, which is dependent on the execution-time order of events in the simulation. It is not possible for the shared-memory model to provide a single memory value to the reference model. Instead, the reference model must rely on (and trust) the value generated by the simulated CPU. This is entirely acceptable because MOM will look at the simulation memory data, compare it to the list of possible values, and check that the received value is valid. The act of observing the simulated data causes the state space of the memory model to collapse into the observed state (similar to the behavior of a quantum particle which collapses to a single state when observed under Heisenburg's Uncertainty Principle!). Thus, both memory storage and memory data checking come "for free" when checking the protocol. The reference memory simply becomes a memory *checker*. The following section highlights some of the more common violations that the checker looks for.

2.2 Checking Shared Memory Properties

The memory order model is capable of detecting violations of any of the rules and properties defined in the Alpha shared-memory specification [2]. Three of the primary properties checked by MOM are the visibility property, the storage property, and the acyclic ordering (causality) property. The code that implements these property checks is small—about 2000 lines of C++ code.

2.2.1 Visibility Property

A piece of data is "visible" to a CPU when that CPU can legally consume it. The visibility rule states that a LD may

receive data from either the latest external[‡] ST observed by the LDs CPU, or it may receive data from the latest internal ST on the LDs CPU (as ordered by the program order) whether or not it is yet visible to other CPUs. Violations of the visibility rule often occur when a load receives data from a write buffer entry that is not the latest store in the program. In the Alpha architecture, this condition is supposed to be detected by the memory system and result in a LD-ST order trap. This hardware trap detects that a load would receive stale data, and schedules the load to re-try later (after the original ST instruction has completed). See **Example 2-2**:

Example 2-2:

CPU-0

ST 2 à X

ST 3 à X – could issue to the memory controller after the LD in an out-of-order machine.

LD X ß data=2 (load should have been trapped and wait to re-issue after the second ST)

In the above example, both stores are visible on CPU-0 as soon as they are reported to the memory order model. But the second store is the *most* visible because it is ordered *after* the first store. Note that the LD in this case may actually be ordered before the stores because there is no strict ordering requirement for LDs following STs in program order.

2.2.2 Storage Property

Violations of the storage property occur when the data read from a location does not reflect the most "visible" data written. This can occur if there is a bug in the logic responsible for latching, byte aligning, or ECC-correcting the data. See **Example 2-3**:

Example 2-3:

CPU-0

ST 2 à X

---Data is not properly latched, so stale data remains--

LD X ß data != 2

One can see that there could be cases where the stale data ==2 and we would miss the bug. This brings up the issue of data uniqueness, which is covered later.

2.2.3 Acyclic Ordering (Causality) Property

Since memory access orderings are required to be acyclic (time must move forward), this property may also be checked. The following case (**Example 2-4**) would lead to a cycle violation.

[‡] Here, "external" refers to memory accesses from other CPUs or I/O devices; "internal" refers to memory accesses originating from the same CPU.

Example 2-4:

CPU-0	CPU-1
[A] LD X ß 3	[C] LD X ß 2
[B] ST 2 à X	[D] ST 3 à X

Since [C] received its data from [B], [C] is after [B]. By the same reasoning, [A] is after [D]. We also have an ordering between [A] and [B] due to the program order, as well as between [C] and [D].

This leads to the cycle: [B] => [C] => [D] => [A] => [B]

Violations of the acyclic property most often result from violations of visibility properties. They can also occur when the cache coherency protocol in a processor has broken down and allowed multiple processors to gain write permission to a block concurrently. The concurrent writes of multiple processors can cause CPUs to observe memory accesses in opposite orders (thus creating a cycle in the ordering graph).

2.3 Representation of Memory Events in MOM

MOM represents shared-memory orderings in a directed acyclic graph (DAG). The nodes are memory accesses such as loads, stores, instruction-fetches, and memory barriers. The edges represent orderings between accesses. **Figure 3** is a representation of **Example 2-1**. If, for example, a store is determined to be before a load, then there will be a directed edge from the store node to the load node. The "Root" node contains the initial value of all memory locations referenced by the DAG.

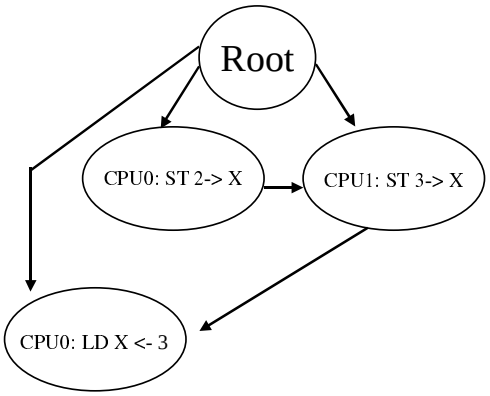

Figure 3

As memory accesses are added to the DAG, the rules specified in the Alpha shared-memory specification are applied. These rules may lead to additional graph edges or changes in visibility. An edge that would introduce a cycle in the graph indicates a violation of the Alpha shared-memory specification (**Figure 4** illustrates **Example 2-4**):

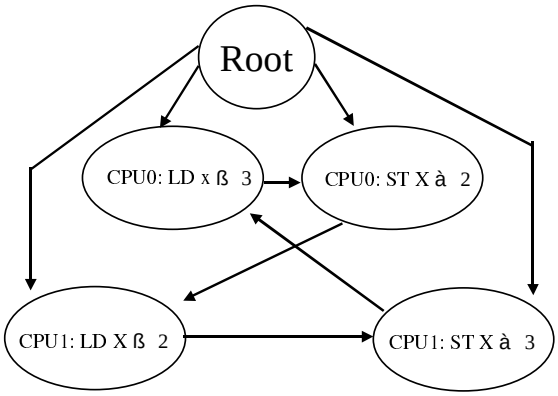

Figure 4

Other shared-memory properties, such as visibility, storage, and causality are also checked within the DAG.

2.4 Algorithmic Considerations

Early on in the development and testing of the shared-memory model, it became clear that computational efficiency is critical. Initial implementations of MOM achieved a test-bench overhead of less than 15% for simple cases. However, test cases containing hundreds of LDs and STs caused simulation time to increase by orders of magnitude. Several techniques were applied to reduce the run-time cost of the memory order model.

2.4.1 Algorithmic Optimization

Many of the rules that are applied when accesses are added to the DAG involve searches. To prevent the order model from becoming a computational bottleneck in the simulation, algorithms had to be implemented with O(Nlog(N)) worst-case complexity. This was accomplished using common DAG data structures/algorithms [10] to make graph searching more efficient. Other search optimizations involved traditional techniques such as height-based termination and node marking to deal with reconvergent fanout.

Applying these optimizations at the expense of memory usage allowed the model to process thousands of nodes before its computational requirements began dominating the simulation.

2.4.2 DAG Size Reduction

Additional optimizations were required to allow efficient simulations of hundreds of thousands of loads and stores. Reducing the size of the DAG is the other major component of optimization. The method for optimizing the DAG takes advantage of the knowledge of what rules can be applied to memory accesses. Accesses that are no longer needed are collapsed from the DAG and their state is moved to the root node. Referring again to **Figure 3**, if no CPU has seen the data stored by CPU0, and anyone sees the data stored by CPU1 (as with CPU0's load), then the "ST 2" node is no longer visible within the system and can be collapsed into the root node. The root node's "initial value" for that address would be set to "2" as a result.

This collapsing optimization allows simulations to dispose of unimportant DAG nodes. Whenever graph updates have occurred, MOM sweeps through the candidate nodes and collapses appropriate nodes. Using the collapse method, simulations of hundreds of thousands of accesses can easily be achieved without the memory order model becoming a significant computational or memory drain relative to the RTL simulator.

Another way to reduce the number of nodes in the graph involves trading off some implementation independence. By using specific knowledge about an implementation's access properties, the DAG size can be further reduced by avoiding the creation of unnecessary nodes. Any such assumptions about an implementation must then be checked either by the memory order model or by an additional assertion built into the simulator.

2.5 Tracking Data

Each implementation of the Alpha architecture must build specific hooks to communicate with MOM. One such hook must report the program order of memory accesses on a given CPU. This data is necessary to allow the DAG entries for that CPU to be inserted with the proper ordering. This hook provides MOM with the ability to do data checking in MP shared-memory simulations. But another more complex simulation hook is needed to provide MOM with the ability to check for compliance with the shared-memory specification. Namely, MOM must be able to determine which store produced the data received by each load. Doing so requires that we associate a unique identifier for each datum associated with a memory location.

2.5.1 Tracking Based on Data Uniqueness

For unique data, this task is trivial; the received data can be used to determine the store that created it. However, if two different CPUs perform stores of the same value to the same location and a load subsequently consumes that value, MOM cannot rely on the received data to distinguish between the two stores. This may seem like an unlikely event, but consider a semaphore contention test where the semaphore is always 0 or 1. Also, the Alpha architecture supports an instruction that zeroes out an entire 64-byte cache block. So, consider a program instruction sequence in which the cache block is zeroed out, partially written, then zeroed out again. If a load were to receive zero, it cannot be determined if the load received the first version or the third version of the memory location simply from the data received. While this program sounds useless, such sequences must be verified to be in compliance with the Alpha shared-memory specification.

This issue can be solved in many (but not all) cases by careful analysis of the non-unique store data. For instance, if there are two potential sources for the data, and one is definitely illegal,

we can assume the load is associated with the legal one. If this leads to a conflict or cycle later on, then there is a bug. If there are multiple store scenarios (i.e., multiple combinations of stores that could have sourced all the observed bytes of the load), then MOM will apply its "best guess" of the correct sequence (potentially allowing a bug to slip through!). This has been measured to occur less than 1% of the time in random simulations.

2.5.2 Tracking Based on Data Versioning

That 1% could be caught if MOM avoided the unique data constraint by associating a unique identifier, or version, with each memory access that occurs in the system. This identifier allows MOM to reference a specific memory access when reporting errors or updating state, and allows the tracking of non-unique data as well.

To provide this capability in the RTL-to-MOM interface, a version-tracking module was developed. It was the responsibility of this module to shadow all data movements through the memory subsystem and inherently across multiple processors with version movements. If a block of data is moved anywhere, its version must also be moved.

During the verification of EV7, a "version buffer" (VB) was created for the register file, the store queue, the L1 instruction cache, the L1 data cache, the L2 instruction/data cache, the router, main memory, the victim buffers, and even the I/O ASIC. These version buffers contain the versions of memory locations residing in the corresponding structure within the simulated RTL. The version tracker watches the RTL for movements between these data-storage structures and moves the corresponding version identifier between the corresponding version buffers. As an example, consider a load that hits in a remote processor's L2 cache. The simulated RTL will first move the data to a victim buffer. The version tracker will consequently move the data's version from the L2 VB to the victim buffer VB. Next, the RTL will move the data to the router, and the version tracker will move the data's version from the victim buffer VB to the router VB. Eventually the data will reach the router of the processor from which the load originated. The version tracker will then move the version from the remote processor's router VB to the requesting processor's router VB. Lastly, the data fills into the L1 data cache and the register file. This causes the version tracker to move the data's version from the router VB to the L1 data cache VB and also to the register file's VB.

Using the version-tracking module, the interface can inform MOM of not only the data that was received, but also the version that was received, with a resulting independence from data uniqueness constraints. However, this checking came at great cost. The implementation of the version buffers was extremely complex, time-consuming, and highly dependent on the implementation of a particular processor. Fortunately, this effort overlapped with the creation of other cache coherency checking software and reduced the net cost of the version buffer development.

2.6 Implementation-Dependent Checking

If the model only reported errors on orderings that violate the Alpha shared-memory specification, it would not detect bugs that violated an implementation's cache coherence protocol (unless that breakdown eventually led to the violation of an architectural shared-memory property). Implementations also commonly maintain coherence by creating memory systems that maintain more strict orderings than the shared-memory specification requires. One such example: The Alpha memory model allows more than one processor to have write permission to a block concurrently. Having multiple unique dirty versions of the data in the system is difficult to support, so the feature has never been implemented in an Alpha processor (instead, we assume that only one dirty copy of the data may exist in the system for a given address).

MOM is well positioned to report violations of implementation- specific assumptions about orderings. This allows robust checking against both the implementation-specific and the generic Alpha memory specifications. This type of implementation-specific check typically catches cases where more than one processor *does* get write permission to a block concurrently. For this bug to become an ordering violation, a series of events would need to occur such that two CPUs observe different orderings of the writes to the cache line. Putting the implementation-dependent checks into the order model allows the bug to be detected immediately rather than waiting for the necessary observability conditions to be met.

There are many other areas in which an implementation may implement rules that are stricter than the Alpha architectural specification requires. Examples include atomic data update instructions, IO processing, and cache management instructions.

2.7 Debugging MOM failures

Debugging memory order failures can be a daunting task. Violations of ordering rules can manifest themselves long after the actual bug has occurred in the simulation. The user may only see the effects of an illegal cache transaction after the block has passed through several other processors.

It is also common for a bug to require dozens of memory accesses to create the visibility and ordering components necessary for failure. A user can easily be overwhelmed when looking through text-based representations of the memory accesses and resultant orderings.

To simplify debugging of failure cases as well as the order model itself, we use a visual representation of the DAG. The viewer displays a typical graph representation with memory accesses shown in boxes containing pertinent information about the access and with arrows between the boxes representing the directed edges (memory orderings), similar to **Figure 3**. This representation allows the user to quickly see the relationships between memory accesses leading up to the failure. The visualization is linked to the simulator such that it

is updated concurrently with the internal data structures. The visualization tool also minimizes nodes from uninteresting parts of the graph.

The memory order model's visualization tool along with text based debugging tools and RTL model debugging tools have allowed users to efficiently determine the source of memory order specification violations.

3. Results

MOM has been used on two successive generations of the Alpha architecture, one using the data uniqueness tracking method, and the other using the version buffer tracking method. As stated earlier, only 1% of the total loads are thrown away in the data-based method. The data-based method is nearly implementation-independent, because only a simple simulation hook is needed to report retired memory accesses. Thus, a new project can get 99% coverage of memory contents very early in a project. As the project stabilizes, version buffers can be added to catch the last 1% of interesting data-dependent memory operations.

3.1 Bugs

Theoretical and experimental results indicate that we could detect memory-order-related bugs 35% earlier in a test case when activating MOM. This, however, is not a good measure of effectiveness because MOM incurs a certain percentage of simulation overhead for both passing and failing cases.

The effectiveness of MOM can be better measured in its ability to find unique bugs that are undetectable by other means such as assertion checkers, reference models, or (formal) verification of the high-level coherence protocol. **Table 1** summarizes MOM-detected bugs as found on EV7/EV8:

% Bugs detected simultaneously by MOM and other mechanisms:	38%
% Bugs detected first by MOM, but which could be found by other mechanisms later in the simulation or later in the project:	61%
% Bugs that are detected by MOM but which could not be found via any other existing checking mechanism:	1%

Table 1

The first category of bugs generally occurs in memory accesses that are not to shared memory locations. Bugs of this nature are usually simple single-stream memory ordering issues on the same processor where the loaded data doesn't match the most recent store.

The second category of bugs can have several sources. If MOM detects an error during insertion of a node into the DAG, that error might not be visible to the RTL unless the memory location is subsequently accessed. This may or may not happen during the current simulation test case. Bugs can also be hidden from checkers if subsequent store data overwrites the bad data before any processor observes it. In either case, it may take many simulation cases (which stimulate the bug) before the necessary observability conditions are met. In the pathological case, the conditions for observing the bug *in simulation* may never be achieved.

The third category of bugs, while small, is the most important. It usually occurs for loads from shared memory locations. Here, a standard memory reference model would not be able to predict the exact ordering of loads and stores, and consumers of this particular memory location would be marked as "unpredictable" and would not be checked. MOM is the only mechanism capable of determining the correct answer in these cases. Without MOM, it is likely that such bugs would have reached the post-silicon testing stage.

3.2 Simulation Overhead and Scalability

One of the primary concerns when developing MOM was to achieve low simulation overhead (in both computation and memory) and excellent scalability from a single simulated processor to hundreds of processors. Contributing factors include:

- The ratio of memory instructions to non-memory instructions in the simulation (affects number of DAG nodes)
- Total number of memory accesses in a test case (affects DAG width)
- Number of accesses to a particular address (affects DAG height)
- Read/Write patterns of the stimulus (affects complexity of the DAG node ordering, reconvergent fanin/out, etc)
- Uniqueness of data (Affects search time in DAG. When not using version buffers, DAG must be searched to find which stores may have generated the data observed by a load)

Figure 5 illustrates these factors:

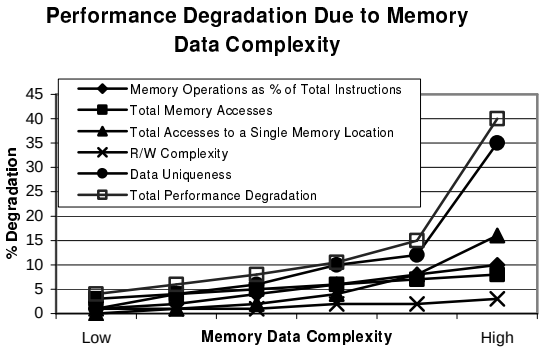

Figure 5

Note that, when not using version buffers, performance for complex cases is dominated by the non-uniqueness of the data. This is because the DAG search/data-resolution overhead increases dramatically when multiple sources for a data value exist. These non-unique data cases were minimized by the intelligent generation of data values by the pseudo-random test scripts… most MOM simulations incurred approximately 10% overhead in the EV8 simulation environment. MOM was typically disabled during simulations that tested non-memory subsections, which also reduced the total cost of running the tool.

3.3 Future Work

While the memory order model has found many bugs, it still has numerous areas that can be enhanced. There is always room for improvement in efficiency, error checking, and debugging capabilities. Work is underway to reduce or simplify the memory order model's dependence on implementation-specific features. Each new implementation currently requires new version modules and new rules.

Another significant task is to formally prove that the memory order model itself implements and fully checks the Alpha shared-memory model. Without formally verifying that the model accurately represents and checks an implementation's memory access ordering, we may be reporting errors on orderings that are in fact legal… or much worse, we may be allowing violations of the formal specification to slip through the checker. Until formal verification is performed on the order model, the engineering team is relying on the *informal* interpretation of the specification.

4. Conclusion

The memory order model has proved to be a valuable verification resource by helping to find difficult multiprocessor bugs. While flagrant violations of the Alpha shared-memory specification are not common in the functional development stage, it is common to find subtle problems with the cache coherence protocol that can lead to ordering and visibility violations. The primary benefit of the order model is that it allows the verification team to run shared-memory MP simulations with confidence that memory data is being checked. Other solutions to this problem, such as complex run-time assertions, formal verification, or tightly controlled directed stimulus, require very tightly restricted stimulus generation, impacting the time necessary to create cases and the quality of the cases, or provide sporadic checking.

5. Acknowledgements

Several individuals helped to generate and analyze the data contained in this paper. The authors would like to thank Joe Huber, Soohong Peter Kim, Thomas Labonte, Michael Quinn, and Greg Trendel for their assistance.

6. References

[1] Jain, A., et al., "A 1.2 GHz Alpha Microprocessor with 44.8 GB/sec of chip pin bandwidth", ISSCC 2001 Proceedings.

[2] The Alpha Architecture Committee, *Alpha Architecture Reference Manual*, Digital Press, 1998.

[3] Intel Corporation, "Pentium II Xeon Processor Specification Update", order number 243776-003, 1998.

[4] Mike Quinn, Scott Taylor, et al., "Functional Verification of a Multiple-issue, Out-of-Order, Superscalar Alpha Processor – the Alpha 21264 CPU Chip," *Proceedings of the 35th Design Automation Conference*, June 1998.

[5] D. Marr, et al, "Multiprocessor Validation of the Pentium Pro Microprocessor," Intel White Paper.

[6] J. Yen, et al., "Multiprocessing Design Verification Methodology for Motorola MPC74XX PowerPC Microprocessor", *Proceedings of the 37th Design Automation Conference*, June 2000.

[7] D. Marr, et al., "Multiprocessor Validation of the Pentium Pro", *IEEE magazine*, November 1996, pp. 47-53.

[8] Lenoski, D., "The Stanford DASH Multiprocessor", Ph.D. Diss., Computer Systems Laboratory, Stanford University, 1992.

[9] Laudon, J.P., and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server", *Proc. 24th Int'l Symposium on Computer Architecture*, 1997.

[10] Lewis, H.R., and Denenberg, L., *Data Structures & Their Algorithms*, HarperCollins Publishers, Inc., 1991.

[11] Pong, F., et al., "Verifying Distributed Directory-based Cache Coherence Protocols: S3.mp, a Case Study", *International Conference on Parallel Processing* (EuroPAR), 1995, pp 287-300.

[12] Nalumasu, R., et al., "The 'Test Model-checking' Approach to the Verification of Formal Memory Models of Multiprocessors", in A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, Springer-Verlag, 1998, pp. 464-476.

[13] Henzinger, T., et al., "Verifying Sequential Consistency for Multiprocessor Memory Protocols", *Proceedings of the 11th International Conference on Computer-aided Verification* (CAV 1999), Lecture Notes in Computer Science 1633, Springer-Verlag, 1999, pp. 301-315.