Stimulus Generation for Functional Verification of Memory Systems in Advanced Microprocessors

BhanuPratap Singh Chouhan, Vaibhav Anant Ashtikar, Basavaraj Talawar, Vani M.
National Institute of Technology Karnataka, Surathkal
Karnataka, India

Krishnakumar Ranganathan, Nagesh Vishnumurthy
Broadcom Corporation
United States

***Abstract* — Memory model verification is one of the most important and complex processes in functional verification of advanced microprocessors. Various components (such as system bus, buffers, caches, memory management unit, and so on) comprise the memory system, and each component demands verification of its functionality for all possible states. This paper proposes various scenarios onto which memory model state space can be covered to a large extent. The stimulus generation should revolve around these methods to impose verification of the memory model. Combinations of such methods will ensure coverage to a large number of corner cases, which are practically infeasible to be generated on a stand-alone basis. The proposed methods extend over Uni/Multi processor scenarios and ensure the integrity of stimulus at the architecture level.**

I. Introduction

Functional verification of the memory system is a vital module of the functional verification of a microprocessor. Functional verification is the task of verifying that the logic design functions as expected; in other words, it is functionally correct. The functional verification of the memory system basically comprises verification of the implemented memory model. And verification of functionality of hardware (such as system bus, read and write buffers, caches at different levels, physical memory, and memory access controller) against the memory operations issued verifies implementation of the memory model.

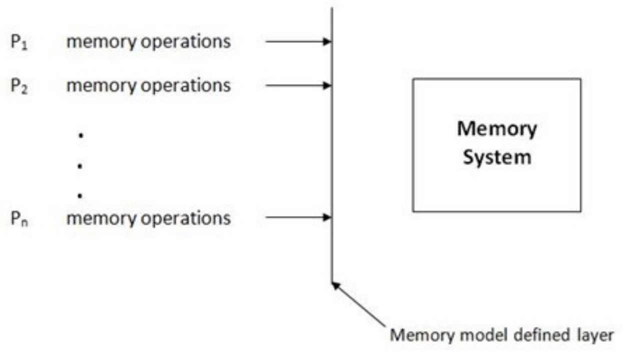


Fig. 1. Conceptual view of a memory model

A memory model is a contract between hardware designers of memory systems and programmers that describes how a memory system behaves in response to memory operations such as reads and writes [1]. Processes P1, P2, and Pn issue various memory operations to the memory system, as shown in Figure 1. Typically, these memory operations are reads and writes to various memory locations. A memory model defines the set of possible ways to execute these operations from various processes [1]. The memory model must ensure that the program will not read or write a wrong value from or to memory at any point of time in execution.

A memory model for a typical uniprocessor system is the classic Von Neumann memory model, which requires that all memory operations in a program complete in the order in which they appear in the program. Memory models for multiprocessors are usually much more involved because of complex interactions between memory operations on different processors. One of the first memory models proposed for multiprocessors is sequential consistency [2], which extends the uniprocessor memory model for multiprocessors in a natural and intuitive way. A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [3]. Sequential consistency is the strongest memory model proposed for multiprocessor memory systems. However, it restricts the use of many commonly-used optimizations in the design of memory systems [3].

Hence, weaker memory models [4] have been proposed as an alternative to sequential consistency to achieve better performance. Weaker memory models relax the constraints of sequential consistency in one or more ways, enabling various optimizations. Modern-day multiprocessor systems provide weaker memory models, providing a considerable enhancement in performance. The SPARC V9 architecture proposes one such weaker memory model with Total Store Order (TSO) [5]. Read operations are allowed to bypass a write operation to a different address. All write operations are enqueued in the write buffer in program order. These write operations are completed by updating the memory when the processor can access the memory. If a write operation for the same address as a read operation is enqueued in the queue, then the read operation completes by reading the value associated with the write operation in the queue. Otherwise, the read operation bypasses the write buffer and completes by reading the value from the memory. Note that all reads complete in program order and no write is allowed to bypass a read.

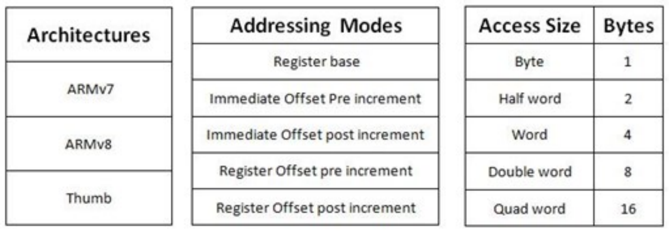
To verify weakly-ordered memory, the simplest way is to execute the sequence of memory operations from a stimulus on the memory model and see whether or not it is behaving as expected. However, doing this alone will not cover all the state space of the memory model. The key is to cover as many scenarios as possible. To achieve this, we can explore the following methods, which cover different aspects of memory model verification. All these verification methods are stimulus-based and can be used to generate stimulus. The generated stimulus will try to cover some or more scenarios, depending upon the methods used to generate it, and then it will be verified against the expected execution.

A. Contribution of This Paper

This paper aims to improve memory system verification with the help of different randomization techniques. These techniques ensure that test generation need not be too complex, and that it is not required to take all the aspects of the memory model space into consideration for the generation. These techniques are more useful to generate stimulus using verification test generator tools like Random Test generators [6] [7] [8]. This paper describes both random test generation as well as random environment techniques like page table randomization, random bus configuration, and so on.

II. RANDOMIZATION ON MEMORY OPERATIONS

A microprocessor can support one Instruction Set Architecture (ISA) or more. An ISA (AArch32, AArch64, Thumb, or other [9] [10]) defines the instructions or operations (in this paper, the terms instruction and operation mean the same) and their behavior. For example, "MOV R1, R2" is an instruction from AArch32, and it moves the data of R2 into R1 (where R1 and R2 are registers). A domain of instruction for a microprocessor is comprised of all the instructions and their variants (based on addressing modes and access size) from all the supported ISAs. Consider a subdomain of all the memory operations and their variants. A generated stimulus will contain random operations from the considered domain.



Architectures
ARMv7
ARMv8
Thumb

Addressing Modes
Register base
Immediate Offset Pre increment
Immediate Offset post increment
Register Offset pre increment
Register Offset post increment

Access Size	Bytes
Byte	1
Half word	2
Word	4
Double word	8
Quad word	16

Fig. 2. Possible architectures, addressing modes, and access sizes

This method covers verification of the ISA and some simple scenarios in memory verification like operations ordering and accesses of different sizes. Verification of memory operations behavior against memory model is also an important aspect of memory verification. This method can verify support for accesses of different sizes and different addressing modes. It can also verify the normal memory operation ordering and dependency. A large stimulus will cover the domain of instructions, while a small stimulus (which covers a sub domain) can be useful to target different aspects.

We can have control over selection of memory operations based on architecture, access size, and addressing modes for targeted verification of memory operation. The controlled selection of an operation can be seen as a weighted selection tree, as shown in Figure 4. The nodes of a tree, showing the selection criteria and the edges from parent node to child node, have the relative weights to select the particular child node. A weight W of any edge is the probability of being selected among other edges from any node. And the selected edge will give the next selection criteria. For example, Warch1 is the probability that Architecture 1 will be selected for the next level from all N available architectures. The sum of weights of all edges starting from one node is 1. On the leaves of the tree, a group of instructions will be collected, based on the criteria that come in the path from the root to that particular leaf. This group can then be used to pick one random operation.

For generation of each new instruction for the stimulus, one path will be selected. At each level, starting from the root, selection of a node in the next level (based on weight) is done to determine the path. The end of the path will give a group to select an instruction or operation. If all the weights on all of the edges of the tree are the same, then the probability of any instruction being selected is the same, as all paths are equally eligible. This will be a case of unbiased randomization over the entire domain of memory operations. Uneven weights for outgoing edges at any node make some next nodes more probable to be selected over others.

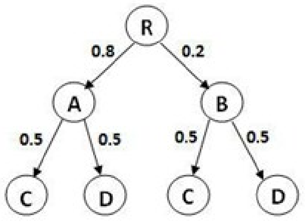


Fig. 3. Weighted tree example

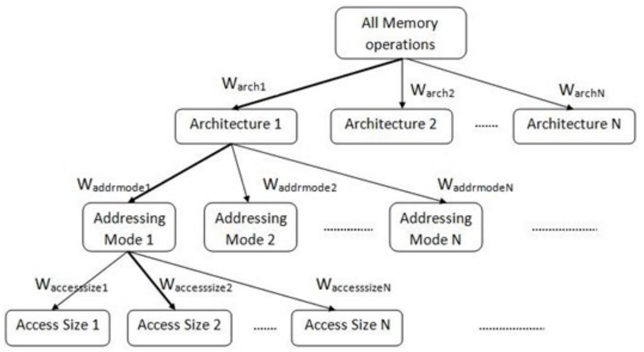


Fig. 4. Selection tree of memory operations

Consider Figure 3, in which the tree is biased towards node A at the first level. Thus node A is more likely to be selected than node B. However, once the node is chosen at the first level (A or B), all weights are equal at the second level; therefore C and D are equally likely to be selected, which is totally random, not biased.

In the tree in Figure 4, the path shown in bold arrows will result in a group of memory operation variants which are from architecture 1, having an addressing mode of 1, and which access the memory of access size 2. The memory addresses that are accessed in generated stimulus are not in control, but rather the accesses are random. This method alone is not of much significance, as it does not cover corner cases. This method is more to verify the behavior of memory operations.

If A is the number of architectures to be verified as having B addressing modes over C varieties of access sizes, then the complexity space of verification would be: (AxBxC)!

III. CONSTRAINED MEMORY REGION

The main memory size is large in modern microprocessors, and one should be able to target verification to specified regions in addition to full memory verification at a particular time. In this method, one or more memory regions are considered for the stimulus generation.

After selection of a memory operation, a memory region is selected randomly or based on relative weights from considered memory regions for that memory operation to access. Any accesses made by the operation must not cross the boundary of the selected memory region. A memory region is defined by starting address (min addr) and ending address (max addr), both of which are inclusive. Suppose the access size (Figure 2) for the operation is X bytes. The operation will access X continuous bytes, and all X bytes must fall in the memory region (min addr, max addr). The starting address of the access made by operation must not fall within the last X bytes of the memory region. The domain for the starting address of the operation would be (min addr, (max addr – X)), as shown in Figure 5. This method follows for all the subsequent operations in the stimulus. This specified region is the hard (must be followed) constrained region, hence the starting address for the operation must be selected from this region only.

Initially, the selected memory region is constrained based on the access size of the operation. Furthermore, the specified region can be constrained by some soft constraints, based on overlapped access and unaligned access.

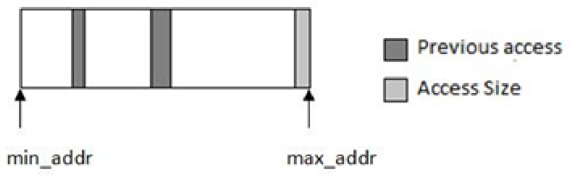


Fig. 5. Memory region

Overlapped Access: In a stimulus generation, operations are generated one by one. A memory region among all considered memory regions can be selected many times for memory operations in generation. At any instance of generation, some of the addresses in a memory region have already been used by generated operations. If the same memory region has been selected for current selected memory operation, then the starting address can be selected either from the previously accessed part or from the unused part of the memory region. The starting address selected from the previously accessed part will cause overlapped access. Consider a part of generated stimulus as given below. The LDR operation has an access size of 4 bytes (a word). It will access 0x4000, 0x4001, 0x4002, and 0x4003. The memory region that was selected for LDR operation has these addresses. In that region, these addresses have been used after LDR. Now, if the same memory region is selected for any selected instruction going forward, these four addresses will be seen as used, and the memory region might contain many other unused addresses. If the starting address for that instruction falls within these used addresses, this could cause an overlapped access. In the following example, LDRB (access size = 1 byte) is showing overlapped access.

```
MOV R1, #0x4000
LDR R8, R1
MOV R1, #0x4003
LDRB R9, R1
```

The generation can have weighted constraint on overlapped access. Based on weight, the generation will select a used or unused part of the memory region to get overlap or non-overlap access, respectively. Although it is not a hard constraint, there is no used address available for the first operation in a stimulus, therefore overlapped access is not possible and constraint cannot be followed.

Unaligned Access: an unaligned memory access occurs when a memory operation with an access size of X bytes tries to access data starting from an address that is not evenly divisible by X (addr % X != 0). For example, reading 4 bytes of data from address 0x10004 is fine, but reading 4 bytes of data from address 0x10005 would be an unaligned memory access.

Unaligned access is costly. For instance, one memory operation of X bytes will actually access two aligned X bytes to get unaligned X bytes, as shown in Figure 6. However, some architectures support unaligned access for some of the memory operation. For example, ARMv8 supports unaligned access for LDR instruction [10].

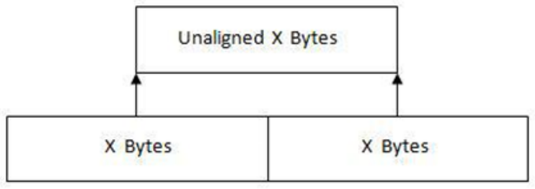


Fig. 6. Unaligned access

Alignment can also be one of the constraints on the starting address of the access by a memory operation to verify aligned access, unaligned access, and a mixture of the two (access selection is based on relative weights). It is also not a hard constraint; for example, if the memory operation selected supports aligned access only, then unaligned access is not possible for that instance in generation.

IV. DEFINING MEMORY REGIONS

A. *Stimulus Generation for a Uniprocessor*

Only one memory region is considered for the generation. All the operations generated will select access addresses from this region only. In this way, the verification can be targeted to a specific memory region.

Alternatively, more than one memory region can be considered. In this case, a memory region is first selected for a memory operation from all considered memory regions, and then that memory region will be constrained based on other parameters. One can verify the behavior of memory system if the accesses in the stimulus are from more regions (belonging to different parts of memory). For example, one can verify how the memory system is behaving if two accesses are either very distant (in different and distant regions) or in the same cache line (in the same region).

B. *Stimulus Generation for a Multiprocessor*

In the case of a multiprocessor system, one or more memory regions can be considered. However, these memory regions need to be divided further. Consider a system of M clusters having N processors each (Figure 7). For each processor, one or more divisions of a memory region are assigned such that each processor has its own access space. This division is necessary because physical memory is shared between processors and we are not considering true sharing scenarios.

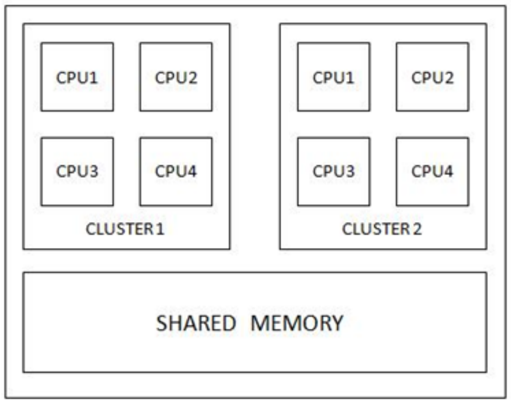


Fig. 7. Microprocessors system M = 2 and N = 4

False sharing occurs when processors in a shared-memory access to different addresses within the same coherence block (cache line or page) [11]. True sharing occurs when the same addresses are accessed by the processors. True sharing between the processors is to be verified using some sort of locking mechanism, such as semaphores or mutex locks. Excluding true sharing scenarios from the memory model for multiprocessor functional verification will reduce the state space of verification. True sharing in the memory model is not considered in any of the methods proposed.

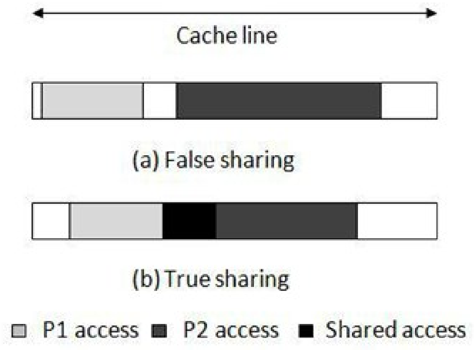


Fig. 8. False and true sharing

Each memory region can simply be divided into N*M equal divisions. Every processor will be assigned one of these N*M divisions in each memory region. Each processor is now analogous to a uniprocessor with memory regions assigned as divisions in memory regions.

For cache verification, cache line sharing (false sharing) between processors would be an interesting scenario. If the memory region is divided as mentioned above, only the cache line at the boundary of divisions will be shared between the processors (also if the boundary is dividing a cache line and does not lie at the end of the cache line). In Figure 9, cpu0 and cpu1 are sharing a cache line at the boundary of their divisions in the memory region. The greater size of the region will result in a larger size of division and lessen the chances of false sharing because, at the maximum, only one cache line is shared between the two divisions.

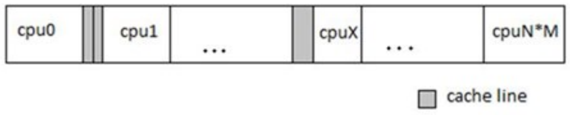


Fig. 9. Simple division of memory

If only one memory region is defined, the same cache line will always cause false sharing between the two processors which are sharing a boundary of division. And at maximum, N*M – 1 cache lines between processors (when every nth processor shares a cache line with the (n+1)th processor) and M – 1 cache lines between clusters (two clusters will share a cache line if the last processor of one cluster shares a cache line with the first cache line of the second cluster), which are less numbers if the region size is big. However it will be useful for targeted verification of a memory region. And sharing behavior on a particular cache line can be verified, if that cache line is at the boundary of some division and the size of the memory region is of few cache lines, to get the sharing more often in a stimulus.

In the case of more than one memory region, one processor will get more divisions (one in each memory region), so more cache lines are shared between processors. But the probability of false sharing will be the same if the sizes of memory regions are the same because the number of memory regions is also increased. However, more cache lines will be verified for false sharing as there will be one cache line shared between the two processors sharing a boundary in each memory region.

The memory region can be divided in different ways to achieve different goals. One way is to divide the memory region into equal parts (say P), and then each part is divided into N*M divisions. Total P*N*M divisions where each processor will be assigned P divisions (from P parts) are shown in Figure 10.

Another way to define a memory quantum is as a division size (say m, where memory region size is divisible by m). The quantum is assigned to processors in round robin fashion. The processors take each quantum one by one. After the last processor has taken its quantum, the next quantum will be taken by the first processor, and so on.

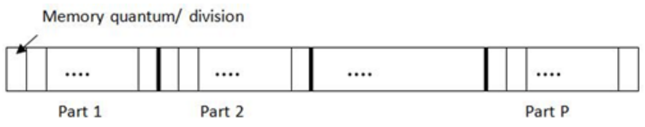


Fig. 10. Part-wise division of memory

In this way, a processor will get more divisions in one memory region and will access memory from different parts of memory. The probability of false sharing is greater here with respect to previous method. By changing the size of the parts, one can create different scenarios. For example, if the size of the part is equal to the size of the cache line, then all of the accesses in the stimulus are false shared.

V. SOME OTHER TEST GENERATION SCENARIOS

Cache Clean and Invalidate: Clean operation causes the contents of the cache line to be written back to memory (or the next level of the cache), but only if the cache line is dirty. That is, the cache line holds the latest copy of that memory. Invalidate simply marks a cache line as invalid, meaning the cache line will not be hit. Invalidate alone will invalidate the dirty line, too; which might result in loss of data. Therefore, an invalidate operation should always come after a clean operation.

In between memory operations in a stimulus, the generator can put random instruction to clean or clean and invalidate the specific or random cache lines. This will target the verification of eviction and refill in the memory system and verification of behavior of memory at different levels when explicit cleaning of the specific cache lines is carried out.

Stride Access Pattern: Simple stride access pattern is controlled accesses of the same size after every interval of a particular size. Complex stride patterns may have increasing/decreasing (in a particular pattern) interval sizes or increasing/decreasing (in a particular pattern) access sizes or a combination of the two. The stride access patterns are accessed more efficiently in memory system. The memory system detects the pattern and prefetches the next probable data for access to a lower level of memory (caches). The generator can put specified and controlled macros of stride accesses in between the stimulus randomly. It will target the verification of prefetcher, bus interface unit, and supported patterns.

Bus Configuration Randomization: Memory systems can have more than one bus for general and special memory accesses. The generator can either choose a specific bus or select the bus randomly from available buses for accesses over default buses to verify the functionality of the buses. This can be done by setting system configuration, so the generator should generate a stimulus in which the system configuration is also manipulated on the user's inputs or on a random basis.

Flushing Read (RAC) and Write Buffers Randomly: Any memory operation in a memory system reads or writes into the memory at a different level. These read and write transactions go through read and write buffers. Different memory models may have different rules over the buffers, like a read following a write transaction on same address might wait or can read from a write buffer if the write transaction is still not completed. These types of rules are defined for read and write buffers for all combinations of transactions.

To verify buffers and the memory model rules over them, the generator can explicitly flush or fill the buffers in between the stimulus in controlled or random fashion and then try a different combination of memory transaction on the same addresses which are in buffers or on different addresses. To flush a buffer, the generator can use system or configuration operation; and to fill it, it can use stride or continuous accesses until the buffer is full.

VI. PAGE TABLE RANDOMIZATION

Page tables can be generated for verification tests which will do the translation from virtual space to physical address space. The page table descriptors contain memory attribute information. Page table descriptors decide the memory verification area onto which verification testing will focus. Generally, the page table entries are created in the test generation and are static throughout the test execution duration.

Page table randomization is aimed towards having a verification test that ensures coverage of different memory model attributes every time the test is executed. To cover all the memory attributes of memory, the test generation need not address all the various aspects of the memory model. For example, the test is not aware that the load/store transactions are of the write back write allocate or write back not write allocate memory model. Hence the same test can be used to cover all the memory attributes with page tables that are randomized at run time.

Page table randomization involves randomization of the attributes in the page table descriptors, which defines the memory model properties for the page. For example, ARMv7/v8 page table descriptors contain various attributes like cacheable, shareable, global, Secure-Non Secure, and so on.

Page table randomization flow:

- Set up page table: Initially set the basic translation page table, which is static.
- Disable MMU: Disable the MMU to avoid any translation by any of the CPU cores until new page table entries are created.

Randomized page table:

- The page table entry is randomly selected (mostly from the address space where the test is going to perform most of the transactions).
- Depending on the user input, a page table descriptor is randomized to get a randomized memory model for the page.
- Flush the TLB: clean the Translation Look-aside Buffer entries to invalidate the previous page table translations.
- Enable MMU: Switch on the MMU for address translation.

A. Experiment on Page Table Randomization

This experiment was conducted to plot the scope of memory attributes covered for the verification purpose with the help of the page table randomization technique. One random test was executed 1000 times with page table randomization and plotted against 1000 directed tests generated with different page tables randomly generated.

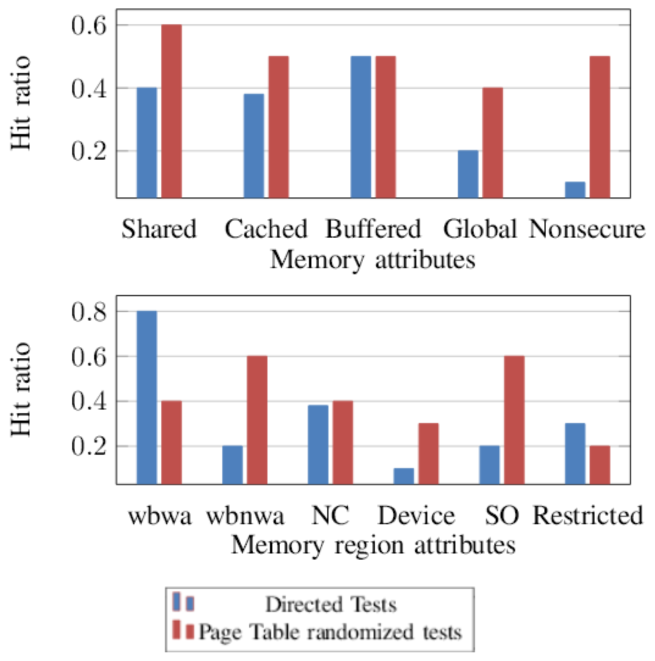


Fig. 11. Results of randomized test vs. directed tests

Hit ratio = Number of times the attribute is seen/total number of times the test is executed.

Figure 11 shows a single test with page table randomization to have more proper distribution over the memory attributes than 1000 directed tests. The mutually exclusive attributes like WriteBack-Write, allocate (wbwa), and WriteBack-Not Write allocate (wbwna) have more random distribution than directed tests. Similarly, all the attributes are covered, including Strongly-Ordered (SO), Non Cacheable (NC), and so on. The attributes are considered from the ARMv7/v8 architecture page table descriptors [9] [10].

The advantage of page table randomization is clearly seen as only one memory verification test is required, which on stimulus randomization produces effective results.

VII. CONCLUSION

Memory System verification has a huge state space to be covered and demands significant amount of stimulus generation. The methods described in this paper can be used to generate dense constrained random stimuli. Though the state space coverage of memory verification of these methods varies from processor to processor, the initial basic verification state space, including memory operations and uni/multiprocessor scenarios, will be well covered. Memory model verification ensures verification of the memory system of a microprocessor.

REFERENCES

[1] Ghughal, R. Test model-checking approach to verification of formal memory models. Master's thesis proposal. University of Utah, Salt Lake City, UT, USA, June 1998.

[2] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. C-28, 9 (Sept. 1979), 690 691.

[3] Goodman, J. R. Cache consistency and sequential consistency. Tech. Rep. 61, IEEE Scalable Coherence Interface Working Group, Mar. 1989.

[4] Adve, S. V., and Gharachorloo, K. Shared memory consistency models: A tutorial. IEEE Computer 29, 12 (Dec. 1996), 66 76.

[5] Weaver, D. L., and Germond, T. The SPARC Architecture Manual Version 9. P T R Prentice-Hall, Englewood Clis, NJ 07632, USA, 1994.

[6] L. Zhongshu, Y. Xiaolang, W. Jiebing and X. Zhihan. A Dynamic Random Instruction and Stimulus Generation for Functional Verification of Embedded Processor. Proceedings of the 5th International Conference on ASIC, October 2003.

[7] Foumier, L, Arbetman, Y and Levinger, M. Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator - Application to the x86 Microprocessors Family, Proceedings of the Design Automation and Test in Europe Conference and Exhibition (DATE99), March 1999.

[8] K. U. Bhaskar, M. Prasanth, G. Chandramouli, and V. Kamakoti. A universal random test generator for functional verification of microprocessors and system-on-chip in VLSID, 2005.

[9] ARM, ARM Architecture Reference Manual. ARMv7-A edition, ARM, 2012.

[10] ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile, ARM Limited, 2013.

[11] Scott, M. L., and Bolosky, W. J., (1993): False Sharing and its effect on shared memory performance. Technical Report MSR-TR-93-01, Microsoft Research, One Way Microsoft, Redmond, WA 98052, 1993.