2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Transactional Memory Subsystem Verification for an ARMv8 Server Class CPU

Ramdas Mozhikunnath, Parveez Ahamed, Brijesh Reddy, Applied Micro, Bangalore, India (rmozhikunnath@apm.com, pahamed@apm.com, breddy@apm.com)

Jayanto Minocha, Applied Micro, Sunnyvale, USA (jminocha@apm.com)

***Abstract*—This paper describes a new verification methodology used on a highly scalable multi-core memory subsystem using a Transactional Test bench. This memory subsystem design is part of an enterprise class ARMv8 server-on-chip that integrates large number of CPU cores along with Level2 (L2) and Level3 (L3) caches, multiple high-bandwidth memory controllers and an IO subsystem. A key design point of the test bench is the ability to map standard multi processor algorithms that stress memory coherency and consistency from the instruction set to a transactional domain. The test bench is developed in SystemVerilog using the OVM methodology.**

Keywords— multi processor verification; ARM server; sub system verification; memory coherency; transactional test bench

I. INTRODUCTION

This paper will introduce a fully featured enterprise class ARMv8 server CPU design that integrates large number of processor cores along with multiple Level 2 (L2) and Level 3 (L3) caches, multiple high bandwidth memory controllers and an IO (Input/Output) subsystem. The challenges of verifying this complex and scalable multi-core memory subsystem is described and details about a new verification methodology using a transactional sub-system test bench is introduced. Results are described in terms of how this methodology helped in rapid and continuous integration testing of memory subsystem, finding multi-processor coherency and consistency bugs, improving simulation efficiency and early performance benchmarking of the design.

A. ARMv8 Server Memory Subsystem:

Figure 1 shows a block diagram of the multi-core CPU design along with the memory sub system which is being verified using the approach described in this paper. The design is highly scalable in terms of number of cores (1 to "N") as shown in the figure. The L2 caches are shared for a pair of cores and are connected to the central switch fabric which takes care of the ordering of memory requests from multiple cores and also has the functionality of a snoop controller. Multiple L3 caches, memory bridges and an IO bridge also are connected to the central switch and are part of the system coherence domain. Multiple high bandwidth memory controllers sit behind the Memory Bridge and connect to standard DRAM memories. The IO Bridge interfaces between the SOC and the process cores. A custom communication protocol is implemented which describes the process by which memory is accessed and the means through which memory is kept coherent across all of these agents.

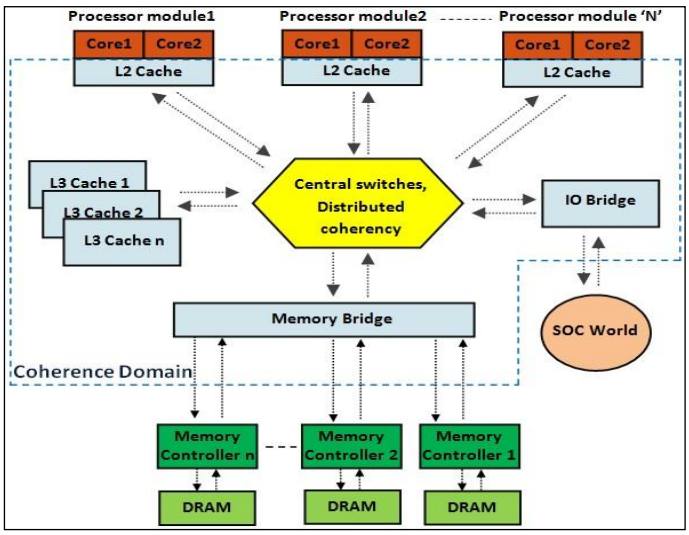

Figure1. Multi-core memory sub system design

B. Multi-core Memory Subsystem Verification challenges:

Multi-core processors are designed to enhance the execution throughput of instructions by executing them in parallel on multiple cores. This increased throughput however, can be utilized to its fullest only when memory dependency between instructions is minimal. This problem is further enhanced due to the presence of caching agents where, recently accessed or recently computed data tends to sit longer, therefore being invisible to the other cores. The coherency protocol ensures that all cores have a 'coherent' view of each memory location in each of the caching agents. In addition to maintaining memory coherency, different processor architectures also define a set of rules governing how the memory system processes memory operations from different processors, known as a memory consistency model. The ARM v8 architecture, that this design implements has a weakly ordered consistency model.

The verification of a coherence protocol and consistency model must address the challenge of state transitions of the entire system which increases exponentially with increasing number of cores and levels of caches. For instance, the fairly common MESI protocol can be described as a finite state machine having four states per caching agent. For a multi-core system using the MESI protocol with 'N' caching agents, the system level coherence protocol FSM can potentially contain 4^N states. This exponential rise in the complexity of the FSM is more challenging in server class processors, which, tend to have a higher number of caching agents as compared to other embedded processors. In order to stress the memory coherency protocol, instructions executed on each individual core need to use addresses from a shared pool which triggers the maximum state transitions across multiple agents. Furthermore, the verification scheme must be highly scalable and configurable to verify designs with a large number of cores and caching agents, without extensive changes to the test bench. This flexibility is important from the perspective of reusability, since every new generation of processors increase the number of cores and caching agents. In addition to stressing the coherence protocol, micro-architectural details must be stressed in conjunction with coherence protocol messages for greater coverage. This is important as it serves to uncover potential micro-architectural deadlocks or live locks arising from snoops which are harder to cover in unit level tests.

Traditionally a lot of multi-core verification happens using tests written in a high level language like C and translating into assembly instructions. These are then simulated on a full processor model including the cores. This is not a very efficient approach especially if the focus of the tests are to cover different aspects of coherency implementation in the memory subsystem rather than the core. This approach is also harder when there is a need to exercise coherent traffic between a CPU core and IO sub system in which either the whole IO subsystem needs

to be part of the design under simulation or need to implement some handshake between the C test and the BFM modeling IO subsystem. The programmatic stimulus running on a full multiprocessor model is also not the efficient approach to generate long harmonic patterns that stress cache state transitions, address collisions, resource starvation and several other micro architectural details concurrently. These are best verified at a sub system level which includes all components part of the coherence domain and which can be simulated faster and longer.

II. STIMULUS GENERATION

This section describes the different approaches towards stimulus generation used in this new methodology, the underlying concepts and the motivation for the same.

A. Mapping instruction set to transactional domain

Every instruction set or any software program has three basic constructs, namely, sequential execution constructs (assignments, computation etc), loops (for, repeat, while etc) and two-way decision constructs (e.g. if..else). One of the key design points of this new methodology is the ability to map these constructs into a sequence style implementation. Implementing response channel on the sequence-driver API enables the sequence to model a decision tree based on responses received. These concepts helped in implementing several multi-processor algorithms in a transactional domain.

Table I below shows mapping of three basic program constructs into a sequence style implementation. An if..else condition can be modeled by a sequence which issues a request to read a memory location and based on the read data received traverse in different paths for generation.

Table I. Sequence Styles for Program Constructs

Mapping Program Constructs to Sequences	
Program Construct	***Translated Sequence***

if (flag ==0) {
do_something();
} else {
do_something_else();
}
	start_read_get_response_seq(flag);
	if (response_data ==0) begin
	do_something();
	end else begin
	do_something_else();
	end

while (flag =0) {
do_something();
}
	read_get_response_seq(flag);
	while (response_data ==0) begin
	do_something();
	start_read_get_response_seq(flag);
	read_get_response_seq.start()
	end

wait (flag ==1)	do begin
	read_sequence.start();
	rand_delay();
	end while (response data != 1)

Table II below shows further examples on how these basic sequence styles can be used to model building blocks for generating multi-processor algorithms in a transactional domain. The first example shows how an atomic increment is implemented using ARMv8 load and store exclusive instructions. A memory location (addr) is loaded exclusively into a register “r1”; the value is incremented by 1 and written back to same address using a store exclusive instruction. The loop is repeated until the store exclusive instruction completes successfully. The second column shows an equivalent sequence style implementation in transaction domain. A sequence issues a read transaction to a memory address; the response fill data is incremented by 1 and passed on to a write exclusive transaction. The loop is then repeated until the response from the write transaction shows an exclusive pass.

The second row explains a similar translation for a standard producer-consumer program across two cores into transaction domain. The third example is for a memory barrier. A memory barrier instruction (DMB in ARMv8) executes to make sure all prior stores are completed and visible. This can be implemented as a request-response

transaction on the core to memory subsystem interface which drives the corresponding signals to flush all stores and monitor for a pending store signal to get de-asserted as response. Using this same approach several other multi-processor programs like message/baton passing algorithms, semaphore get/put implementations, false sharing patterns, spin-lock algorithms etc can be implemented using sequences. Most of these sequences were self checking as the request-response style of API enabled to check for a set of memory locations against an expected value that the sequence can predict. This approach enabled us to build a sequence library of MP (Multi-processor) specific stimulus patterns which can then be randomly exercised as combinations on different configurations of the memory sub system design.

Table III. Equivalent Sequence style for basic blocks of MP programs

<table>
<tr><th colspan="4">Sequence translation for building blocks of basic MP program</th></tr>
<tr><th colspan="2">Basic building blocks for MP programs</th><th colspan="2">Equivalent sequence style</th></tr>
<tr><td colspan="2">atomic increment:
loop: ldex r1, addr
r1 += 1
stex r1, addr
retry loop if stex_fail</td><td colspan="2">do begin
r1 = rd_get_resp_seq (addr)
r1 = r1+1;();
excl_pass = write_excl_seq (addr, r1);
end while (!excl_pass)</td></tr>
<tr><td>Producer:

While(flag!=0){};
Data =1;
flag =1;</td><td>Consumer:

While(flag!=1){};
print data;
flag =0;</td><td>Producer:

rd_wait_seq(flag,0)
write_seq(data,1)
write_seq(flag,1)</td><td>Consumer:

rd_wait_seq(flag,1)
rd_seq(data);
write_seq(flag,0);</td></tr>
<tr><td colspan="2">DMB (Memory barriers)</td><td colspan="2">barrier_seq():
-drive flush stores signal to subsystem
-wait till pending stores goes low</td></tr>
</table>

B. Complex System Level Scenarios

One other subset of stimulus generation consists of complex system level scenarios that focus on memory ordering and coherency across the multi-core memory subsystem while also stressing on the micro architecture and coherence protocol implementation. We defined a set of high level scenarios that are of interest from a sub-system level verification and implemented those as part of a higher layered virtual sequence. The virtual sequence would then map these scenarios into multiple sub-sequences that can be run in co-ordination across multiple lower layer sequencer/drivers across different agents. Table III below shows few examples of these higher layered scenarios and how those are translated to a lower layer implementation. An example scenario like "CPU_MEM_WR_BARRIER_FETCH" would translate to a sequence that streams writes on L1 Data cache interface and then starts a barrier sequence followed by a sequence on the L1 Instruction Cache interface that streams code fetches.

Table IIIII. Higher layer Transactions

Sample list of higher layer Transactions	
Higher layer transaction	***Translation to lower layer***
CPU_INST_FETCH CPU_PTE_READS CPU_MEM_RDS CPU_MEM_WRS	CPU doing instruction fetches One CPU doing Page Table Reads CPU doing stream of Memory Reads CPU doing stream of Memory writes
CPU_MEM_PRFMS	CPU doing stream of pre fetch operations
CPU_MEM_RD_MOD_WR	CPU doing Read-Modify-Write sequence
CPU_MEM_WR_BARRIER_WR	CPU doing stream of writes followed by barrier followed by another stream of writes

Sample list of higher layer Transactions	
Higher layer transaction	***Translation to lower layer***
CPU_MEM_WR_BARRIER_FETCH	CPU doing stream of writes followed by a barrier and followed by instruction fetches
CPU_BARRIER	Barrier on a CPU
IOB_MEM_RD IOB_MEM_WR	Memory reads and writes from IOB
CPU_TLB_INV	CPU doing stream of TLB invalidates

The implementation of running multiple lower layered sequences was done in a way to make sure that each of the interfaces had concurrent traffic for most of the simulation duration. In addition to running tests based on transaction count, the methodology also supported running sequences for multiple iterations across interfaces until all of them finish around same time. The sequences also had user defined delay controls to exercise design with long harmonic patterns which are important for identifying bugs related to resource starvation and arbitration related logic.

The address generation for different transactions was done with plenty of controls to make sure that sequences across multiple cores used shared addresses to create colliding patterns at right times to maximize coverage for underlying micro architecture. Few examples include maintaining recently used shared and local address pools across cores and also having a feedback from the checker to pick addresses colliding with in flight transactions.

C. Multi Stage generation and feedback from RTL

As mentioned earlier, in addition to stressing the coherence protocol, it is important to stress the micro-architectural aspects of the memory sub-system to isolate dependencies, deadlocks and live locks. Sequences that stress the memory coherence protocol and provide coverage of all state transitions do not necessarily stress the micro-architectural details of the design. As a result, potential flaws arising due to coherency requests are not isolated. For instance, system level deadlock and livelock issues that can arise due to limited system buffer sizes are not caught by sequences meant for stressing the coherence protocol. Similarly, preferential arbitration mechanisms can cause resource starvation at the system level leading to deadlocks and livelocks which are hard to isolate in unit level tests. Therefore, to cover micro-architectural features across subsystem components adaptive sequences are designed which are intended to target FIFOs, buffers, arbiters and any micro-architectural units that are potential dependencies in the memory subsystem. By proving virtual channel independence, we ensure the stressing of the micro-architecture along with the coherence protocol.

The random stimulus generator is used with dynamically altered weights based on a feed-back mechanism from the RTL. Figure 2 shows the structure of the weight adaptation process.

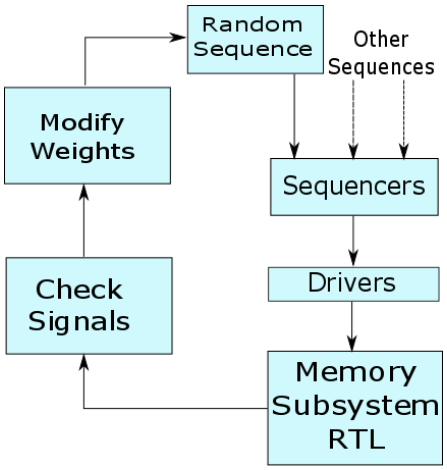

Figure 2. Sequences with adaptive weights

A sequence that generates random transactions is used in multiple modes with modified weights to generate certain targeted conditions that create resource constraints in the memory subsystem. An example implementation uses following stages –

1) Pre-Target Stage – In this initial stage, the random stimulus generator is directed to generate transactions that target certain RTL states that create bottlenecks in the design. Specifically this targets buffers, FIFOs and other resources which can stall forward progress of certain types of transactions like snoops, writes etc. The RTL signals are monitored to reach the necessary starvation scenarios.

2) Target Stage – This stage is reached when the monitored RTL signals are triggered indicating the RTL is stressed in a certain stressed state. The weights directing the kind of transactions are modified in this stage.

3) Post Target Stage – This stage directs transactions which are independent of the starved resources in the RTL. Forward progress of these transactions is monitored to ensure no deadlocks or livelocks are encountered. This stage stresses the design's ability to recover from bottlenect scenarios.

The modified weights required to generate targeted RTL resource constraints are dependent on the micro-architecture specifications. For instance, stressing preferential arbiters in the design require varying patterns of traffic from the sequences. On the other hand, stressing buffers or FIFOs can be achieved with binary switching of weights, where, mutually independent traffic-types are issued in the pre-target and post-target stages.

D. Performance benchmarking Sequences

This testbench was also used to enable early performance benchmarking of memory subsystem. Traditionally, industry standard benchmarks that focus on memory bandwidth and load-to-use latency (e.g. lmbench) are run as programmatic stimulus on the full design including cores even though the focus is on memory subsystem. In this approach we were able to map these into transactional sequences. We were successfully able to run those simulations faster and with a smaller memory footprint than system simulations and identified performance issues early in design cycle.

III. TESTBENCH HIERARCHY

This memory subsystem test bench implementation follows a standard OVM test bench hierarchy that facilitates reuse of unit level test bench components and sequences. In order to meet needs of all the different stimulus generation approaches as explained in previous section, the sequences were layered as lower layer sequences which executes on a specific interface and a set of higher layer virtual sequences that co-ordinates execution of lower layered sequences. Figure 3 illustrates this layered sequence implementation along with the sequencer/driver connections on each of the interface.

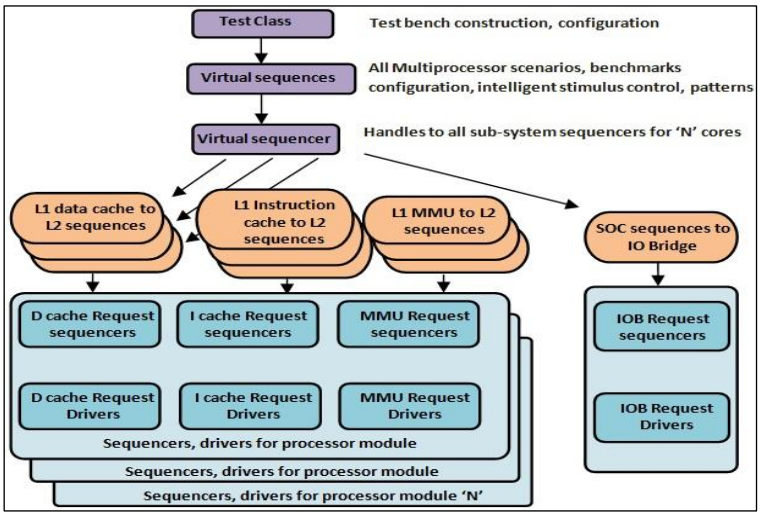

Figure3. Test bench hierarchy and Layered Sequences

A. *Sequence Implementation details*

In this verification methodology, higher layer scenarios like multiprocessor algorithms or complex system scenarios are implemented as virtual sequences. The virtual sequence has information about active agents in the configuration and can decide on which of the L2 cache agents, it can start a lower layer sequence. A custom API is implemented between the higher layered virtual sequence and the lower layered sequences for all the interfaces in the subsystem. Based on the higher layer scenario, the virtual sequence would create one or more of lower layered sequences, configure them by passing correct parameters and also start them on correct sequencer. The responses from the RTL can also flow back from the lower layer sequence to the virtual sequence to decide on further generation as the simulation progress.

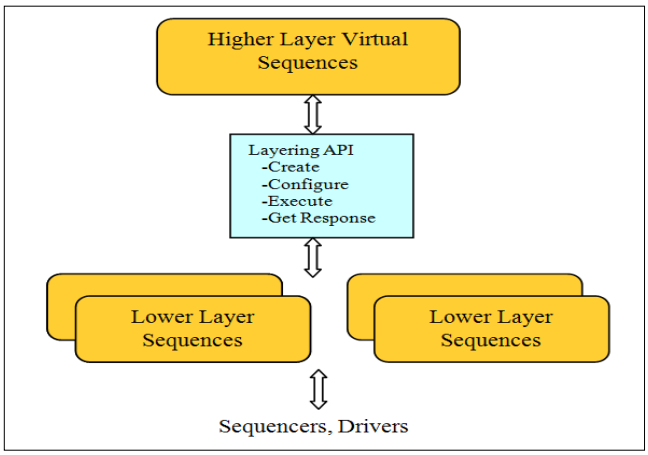

Figure 4. Sequence Layering

B. *Reuse from unit level verification*

The test bench was designed to reuse most of the unit level verification components while also focusing on not over-constraining stimulus for a subsystem level verification. Referring to Figure 3 which illustrates the test bench components and sequences, all of the lower level sequencers and drivers for the different processor interface (Data cache, Instruction Cache and Memory Management Unit) and SOC interface (IO Bridge Request) were reused from unit level verification as enabled by the OVM methodology. The base class sequences for each of the unit verification environment were also reused and specific APIs were added to these sequence classes so as to communicate with higher layer virtual sequences. The virtual sequences and the test class were the only one that had to be implemented with a subsystem level focus to control the stimulus generation across each of the lower level interfaces.

All other components like interface monitors, unit level checkers, assertions and configuration objects were also reused as enabled by the OVM methodology.

C. *Configuration of testbench and DUT*

The memory sub system design is highly configurable in terms of active design units for any given test. One or more of the design units can be replaced with a simulation model at run time and this helps in picking up the most efficient configuration for a category of tests. The minimum configuration supported includes the central switch fabric along with a single instance of L2 cache which can be simulated with dual core stimulus. This approach also helped in early integration testing and stimulus generation. We could start with the minimal configuration and enable other units to be part of subsystem with almost no effort as soon as the corresponding unit level verification stabilizes.

Another aspect of configuration was the large number of configuration options supported in the design spread across multiple design units. Some examples of these include the various addressing modes, interleave options, cache size configurations etc. We followed a uniform configuration mechanism across all of the units which are part of the sub-system in such a way that configuring these units at sub-system level was just as easy as re-using certain knob settings used for unit level verification.

D. Test and User Controls

Figure 5 shows how a test can specify one or more virtual sequences in a test and the levels of control in terms of test bench and design configuration that can be done. In the example, test starts a producer-consumer pattern on a set of cores while running atomic increment patterns on a same or a different set of core/s. A random irritator sequence is also enabled which again can run on any number of cores concurrent with other defined patterns.

```
//Specify which scenarios
+virt_seq_name=producer_consumer
+virt_seq_name=atomic_increment
+random_irritator_enb=1
//More further controls etc

//Testbench configuration example
+Core0.config=rtl
+Core1.config=rtl
+Core2.config=model
+Core3.config=model

//User control for random stimulus
+addr_wt_dram=50

//User control for design config
+addr_mode=interleave
```

Figure 5. Test interface and configuration controls

IV. RESULTS

The test bench was developed to focus on rapid and continuous integration testing for memory subsystem and to find harder multi-processor coherency and consistency bugs. A secondary but equally important design point was to improve the simulation efficiency compared to more traditional approaches which involve running programmatic stimulus on CPUs. We could achieve both of these to a satisfactory level. The integration testing could start as soon as a minimum of two of the unit designs were stable and other units being integrated in parallel. Most of the multi-processor scenarios were verified at subsystem level efficiently thus saving in terms of simulation bandwidth, debug times and effort compared to programmatic stimulus.

This effort helped in thorough testing of memory subsystem across multiple generations of designs. The feedback from subsystem level verification helped to identify deficiencies in our unit level environments and enhance them. This helped in improving our confidence in the overall verification of our platform.

We were further able to expand usage of this approach on our next generation design for early performance benchmarking and finding performance bottlenecks coincidental with unit level design and verification. This alleviated the functional DV bottle neck, where performance validation starts behind functional verification.

We were also able to reduce time, effort and resources instead of over-constraining existing unit level environments to cover system level scenarios, by leveraging the base OVM layers and building a new mechanism for system level co-ordination.

REFERENCES

[1] An Effective Verification Solution for Modern Microprocessors by Ilya Wagner. A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in The University of Michigan, 2008

[2] I. Wagner and V. Bertacco. MCjammer: Adaptive verification for multi-core designs. In *DATE, Proceedings of Design, Automation and Test in Europe Conference*, pages670–675, May 2008

[3] Andreas Meyer and Alan Hunter: So you think you have good stimulus: System level distributed metrics analysis and results, DVCon 2013

[4] Mark Peryer: Seven Separate Sequence Styles Speed Stimulus Scenarios. DVCon 2013

[5] Rich Edelman and Raghu Ardeishar: Sequence, Sequence on the wall – Who's the fairest of them all? DVCon 2013

[6] Shai Fine and Avi Ziv, Coverage driven Test Generation for Functional Verificaiton using Bayesian Networks , IBM Research Laboratory, Haifa