Coherency Verification & Deadlock Detection Using Perspec/Portable Stimulus

Moonki Jang, Jiwoong Kim, Hyerim Chung
Samsung Electronics,1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do 18488, Korea
moonki.jang@samsung.com, jiwoong7.kim@samsung.com, hyerim.chung@samsung.com

Phu L. Huynh, Shai Fuss
Cadence Design Systems, 2655 Seely Ave, San Jose, CA 95134, USA
phuynh@cadence.com, Shai@cadence.com

***Abstract* - Modern SoC, designed for automotive and mobile applications, typically has multiple processor cores, multi-level cache hierarchy, and other subsystems that shared memory and system resources. Cache coherency and deadlock avoidance are two critical areas that need to be verified for this type of design. This paper describes how the Portable Stimulus Standard (PSS) and EDA tools, such as Perspec, were used to shorten the design verification of these two areas: memory coherency and deadlock avoidance.**

I. Introduction

Many System-on-Chip (SoC) designs for automotive and mobile applications consist of multiple processor cores and IP subsystems that share SoC resources such as memory, system interconnect (IC), and IO subsystems. The effort required to manually create sufficient test suites to verify such a design is significant due to the number of processor cores, number of coherent IO masters, multiple level of caches, and the various combination of how these processor cores and IO masters can access memory and shared resources. EDA tools and reuse methodology can help reduce the time and effort required to implement these test suites. With the release of the Accellera "Portable Test and Stimulus Standard (PSS), Version 1.0" [1] in June 2018, the SoC-level verification automation tools and test scenarios reuse will be significantly improved similar to what UVM has done for the IP-level verification.

There are many papers and articles that discuss the cache coherency issue and how to tackle this issue at the protocol-level [2], block-level [3][4], and SoC-level [5]. The main contribution of our paper is to discuss our experience with using PSS and how PSS and EDA tools, such as Cadence® Perspec™ System Verifier, helped shorten the design verification cycle of this type of SoC in two specific areas: memory coherency and deadlock/livelock [6] detection. This was achieved by taking advantage of the test generation automation and coverage collection capabilities provided by Perspec and the reuse of the PSS scenarios from project to project. For coherency testing, we also reduced our verification time further by building our coherency test suite using the actions provided by the Perspec coherency library. Deadlock detection is another area that we focused on in this project; deadlocks caused by the coherency protocol violations, such as PCIe deadlocks, occur only when specific sequence of transactions and timing occurs; it is very difficult to do root cause analysis when the deadlocks occur at the post-silicon level; it is also very costly to uncover these issues at this late stage of the project. To ensure that our design is free of deadlocks and to uncover any such issues before tape out, we have created a "deadlock detection" verification environment to reproduce and verify the root cause of the deadlocks at the simulation level.

Deadlock detection testing presents a unique set of challenge and requires special features to be added to the verification environment to detect deadlocks and to handle these deadlock events automatically when they occur. To check for deadlock condition and to handle it automatically, we setup an auto regression test suite and implemented the following monitors in our verification environment:

- System Monitor receives periodic heartbeat response from the CPU which indicates normal CPU operation. Missing hearbeat responses from the CPU after a predetermined period of time will indicate to the System Monitor that a possible deadlock has occurred; when a deadlock occurs, the System Monitor will send a deadlock alert to the System Tracker.

- Transaction Latency Monitor which records the transactions latency and this recorded data can be used later on to determine the transaction that caused the deadlock.
- System Tracker which captures the system state and the relevant signals for post-run root cause analysis of the deadlock condition.

We will discuss the following topics in more details in the paper:

- Overview of typical automotive SoC architecture
- Using Perspec library and PSS scenarios for coherency verification
- Deadlock detection verification environment and auto regression test suite
- PSS scenarios to detect PCIe deadlock conditions
- Results & lessons learned from an actual project

II. Overview of Typical Automotive SoC Architecture

Due to the computational demand of the Advanced driver-assistance systems (ADAS), coupled with the needs to do real-time image processing, pattern recognition processing, and other "cognitive computing" related tasks, the SoC designed for automotive applications required a large number of processor cores, one or more Graphics Processing Unit (GPU) and Neural Processing Unit (NPU). These processor cores, GPU, and NPU all share memory resources, interconnect bus, and IO subsystems (e.g., PCIe, Ethernet, USB, etc.).

Figure 1 shows the block diagram of a typical SoC for Automotive application. There are multiple processor cores organized into multiple clusters. To reduce the effect of the shared memory bottleneck, there are multiple level of caches. Each processor core has its own private L1 cache (not shown in the diagram); each processor cluster has an L2 cache shared between the cores in that cluster; some design even has an LLC (Last Level Cache, which resides in the Coherent Interconnect) and is shared between all the processor clusters and other coherent masters, such as the GPU.

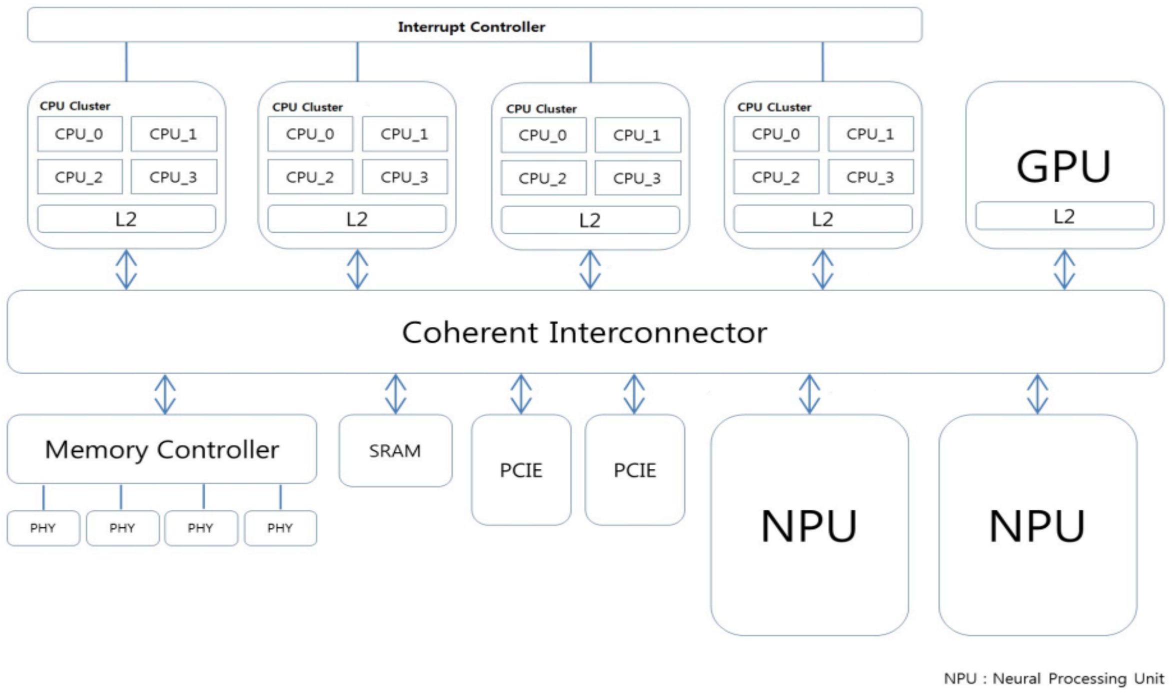


Figure 1: Typical SoC architecture for automotive applications

All these processor cores, GPU, NPU, and IO peripherals will generate their request transactions and receiving requested data through the Coherent Interconnect using the ACE (AXI Coherent Extension) and ACE-Lite protocol. Due to the shared resources and multi-level of caches in this type of design, it is critical to verify that the SoC is free of coherency issues and livelock/deadlock issues.

In section III, we will discuss our approach for coherency verification. Section IV and V will cover our approach for livelock/deadlock verification. Perspec and PSS were used to speed-up the test scenarios development and to facilitate the reuse of these test scenarios for future projects.

III. Using Perspec Library and PSS scenarios for coherency verification

To ensure that there are no coherency issues in the design, we created a set of coherency tests that consist of false-sharing tests, true-sharing tests, exclusive access tests, and DVM (Distributed Virtual Memory) tests. We used the Perspec libraries and its utilities to speedup the development of the above test scenarios. Perspec libraries provided the basic memory read/write actions, coherency actions, and a built-in test suite for memory and coherency verification of the processor-memory subsystem. For the remaining of this section, we will go through the process of creating our processor-memory model, the PSS scenarios using the atomic actions provided by the Perspec coherency library, and how to write reusable PSS scenarios; what we are hoping to convey here is to show how EDA tools (e.g., Perspec) and PSS helped speeding up the test development and also allowed the reuse of the PSS test scenarios from project to project.

The test development process consists of the following steps:

1. Model the compute (i.e., processor-memory) subsystem
2. Develop memory coherency test scenarios for the compute subsystem
3. Develop PSS model for other subsystems
4. Develop subsystems test scenarios

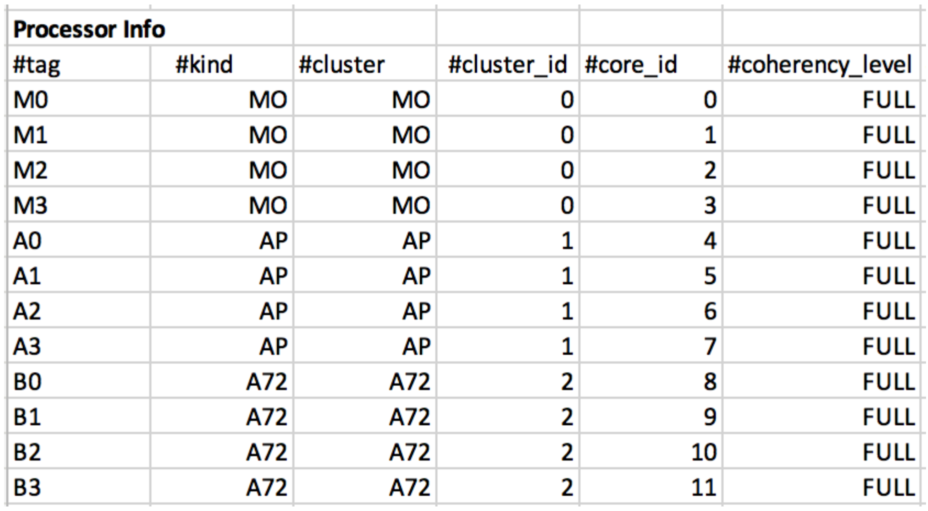
We will only go through steps 1 and 2 in this section; steps 3 and 4 will be described in Section V (PSS Scenarios to Detect PCIe Deadlock Conditions).

<u>Step 1: Model the compute subsystem</u>

Most compute subsystems for mobile and automotive SoC are based on a multi-core processor and interconnect architecture; the differences between them are mainly in the types of processor used, the number of cores, the number of clusters, the cache hierarchy structure, the memory types and sizes. For this reason, the compute subsystem PSS model and its supported actions are suitable candidates for encapsulating in a reusable and configurable library and this is what was provided by Perspec. Using the Perspec coherency library, the "modeling" process of the compute subsystem required no coding; we just needed to fill out the information related to the processor cores, the clusters, the memory types/sizes, the cache structure, etc., in the Perspec configuration tables; these tables were captured in an Excel/csv configuration file; this "modeling" process of our SoC compute subsystem was done in a couple of hours; most of this time was spent tracking down the information required to fill out the Perspec configuration tables. Table 1 and 2 show an example of the processor and memory configuration tables.

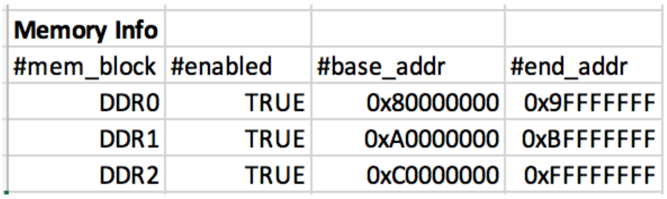
- Table 1 - "Processor Info" table: this table describes the processor subsystem of the design; the columns in this table represent the attributes of the design; some key attributes are:
    - #tag: name of the processor cores; there are 12 of them: M0 to B3
    - #kind: the kind/type of processor
    - #cluster: name of the processor clusters; there are 3 of them: MO, AP, A72
- Table 2 - "Memory Info" table: this table specifies the different memory blocks and their address ranges; in this example, we have:
    - Three different memory blocks: DDR0, DDR1, DDR2
    - All of them are enabled in the design (#enabled column)

Note that, to simplify the discussion, we only show the key columns (i.e., attributes) in Table 1 and Table 2. Users can also add additional columns/attributes to these tables that are specific to their design.



Processor Info					
#tag	#kind	#cluster	#cluster_id	#core_id	#coherency_level
M0	MO	MO	0	0	FULL
M1	MO	MO	0	1	FULL
M2	MO	MO	0	2	FULL
M3	MO	MO	0	3	FULL
A0	AP	AP	1	4	FULL
A1	AP	AP	1	5	FULL
A2	AP	AP	1	6	FULL
A3	AP	AP	1	7	FULL
B0	A72	A72	2	8	FULL
B1	A72	A72	2	9	FULL
B2	A72	A72	2	10	FULL
B3	A72	A72	2	11	FULL

Table 1: **Processor Info** Table



Memory Info			
#mem_block	#enabled	#base_addr	#end_addr
DDR0	TRUE	0x80000000	0x9FFFFFFF
DDR1	TRUE	0xA0000000	0xBFFFFFFF
DDR2	TRUE	0xC0000000	0xFFFFFFFF

Table 2: **Memory Info** Table

Once these configuration tables were filled out, we were able to bring-up Perspec and created memory and coherency tests using its GUI (Graphical User Interface) and writing the PSS code directly.

Step 2: Develop memory coherency test scenarios

In this step, we created simple memory read/write scenarios first to ensure that the processor-memory subsystem was working correctly. Once the basic memory read/write operations were functional, we created more complex false-sharing, true-sharing, and DVM test scenarios. We also used the built-in coherency test suite of the Perspec coherency library, reviewed the coverage results, and added additional tests to achieve the desired coverage. In the remaining of this section, we will show some simple PSS memory read/write scenarios to illustrate how to write reusable PSS scenarios using the actions and utilities provided by the Perspec libraries.

Figure 2 shows a simple PSS memory read/write scenario using the atomic actions provided by the Perspec SML library; two solutions of this scenario are also shown in this figure to illustrate how the tests are generated from this scenario.

This PSS scenario is simple but it does illustrate some important points:

- `sml_processor_tag_e`: is an enumerated type created automatically, by Perspec, using the #tag column in the "Processor Info" table. It consists of the following values: M0, M1, M2, M3, A0, A1, A2, A3, B0, B1, B2, B3.
- write_data, copy_data, read_check_data are atomic actions provided by the Perspec SML library.
- The PSS action, `pss_wr_cp_rc`, can be reused by other projects (even though these projects might have different processor and memory configurations).
- Notice that in the generated tests: proc_tag (processor core), mem_seg_addr (memory buffer address), and mem_seg_block_tag (memory block ID) are randomly picked by Perspec for each test solution.

```
action pss_wr_cp_rc {
   rand sml_processor_tag_e proc_tag_l;

   activity {
      chain {
         do sml_sw_ops_c::write_data with {
            proc_tag == proc_tag_l;
         };
         do sml_sw_ops_c::copy_data with {
            proc_tag == proc_tag_l;
         };
         do sml_sw_ops_c::read_check_data with {
            proc_tag == proc_tag_l;
         };
      };
   }
};
```

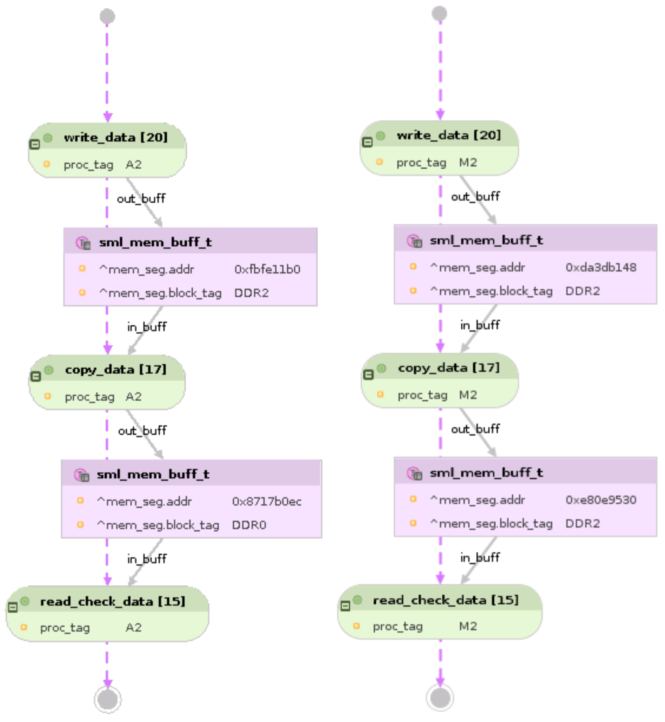


Figure 2: Simple PSS memory read/write scenario

We can use this simple `pss_wr_cp_rc` action as a building block for more complex PSS scenarios. Figures 3 shows the code of the `pss_wr_cp_rc_3cores` action; this action uses the `pss_wr_cp_rc` action to create a scenario that has three processor cores doing write_data, copy_data, and read_check_data in parallel.

The action `pss_wr_cp_rc_3cores` has three control knobs: `proc_tag1, proc_tag2, proc_tag3`; these control knobs allow the test writer to select the processor cores to be used for his tests. Notice also that there are three constraint statements in the code to ensure that the processor cores selected by the test writer (or by Perspec) will be unique since these cores are executing the `pss_wr_cp_rc` action in parallel.

Figure 4 shows an example of a false-sharing scenario (`false_sharing_rw_base` action) executing in parallel to our `pss_wr_cp_rc` action. This scenario illustrates two other convenient features:

- `sml_proc_subset_select_s` is a control knob structure, automatically created from the "Processor Info" table. This control knob structure consists of all the knobs that allow the test writer to select the processor cores, the cluster, etc., to be used in his tests.
- "constraint foreach" in the code ensures that the processor core selected for the `pss_wr_cp_rc` action will be different from the cores selected for the `false_sharing_rw_base` action.

```
action pss_wr_cp_rc_3cores {
    rand sml_processor_tag_e proc_tag1;
    rand sml_processor_tag_e proc_tag2;
    rand sml_processor_tag_e proc_tag3;
    constraint proc_tag1 != proc_tag2;
    constraint proc_tag1 != proc_tag3;
    constraint proc_tag2 != proc_tag3;

    activity {
       parallel {
          do pss_wr_cp_rc with {
             proc_tag_l == proc_tag1;
          };
          do pss_wr_cp_rc with {
             proc_tag_l == proc_tag2;
          };
          do pss_wr_cp_rc with {
             proc_tag_l == proc_tag3;
          };
       };
    }
 };
```

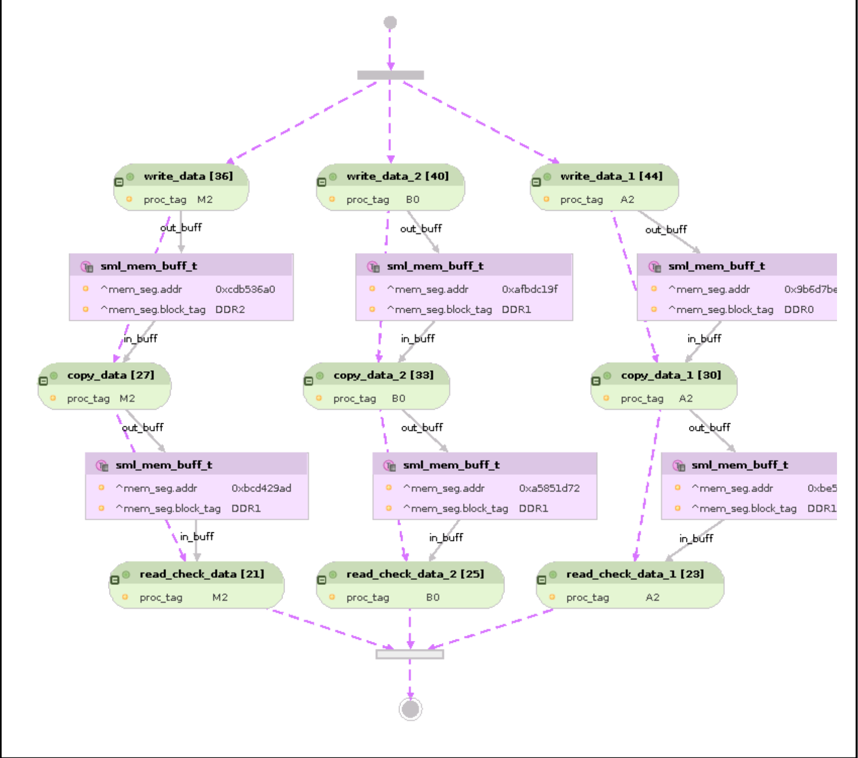


Figure 3: Example of a more complex PSS scenario using `pss_wr_cp_rc` action

```
action pss_fs_rw_mem_rw {
   rand sml_processor_tag_e proc_tag_mem_rw;
   rand sml_proc_subset_select_s procs_subset_fs;
   constraint soft procs_subset_fs.size == 2;
   constraint soft procs_subset_fs.num_clusters == 2;
   //Ensure that "false-sharing" cores != "pss_wr_cp_rc" core
     constraint foreach (val: procs_subset_fs.selected[index]){
        val -> proc_tag_mem_rw != (sml_processor_tag_e)index;
     };
   activity {
      parallel {
         do pss_wr_cp_rc with {
            proc_tag_l == proc_tag_mem_rw;
         };
         do cdn_coherency_ops_c::false_sharing_rw_base with {
            foreach (procs_subset.selected[index]) {
               procs_subset.selected[index] == procs_subset_fs.selected[index];
            };
         };
      };
   }
};
```

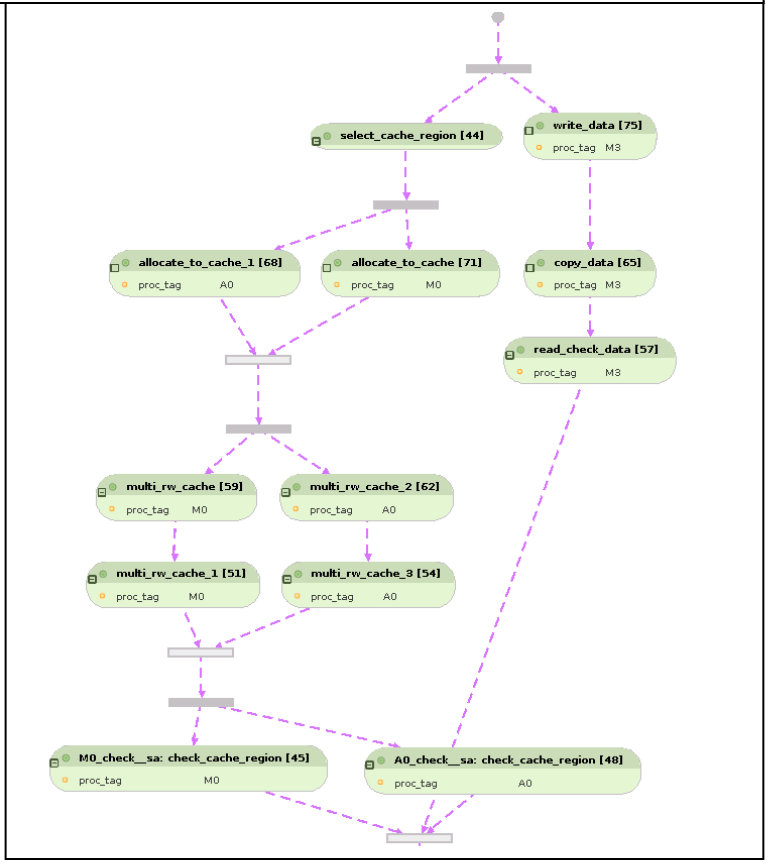


Figure 4: false-sharing scenario using the Perspec `cnd_coherency_ops_c` library action and our `pss_wr_cp_rc` action

IV. Deadlock Detection Verification Environment and Auto Regression Test Suite

The purpose of the Deadlock Detection Verification Environment is to verify that known deadlock conditions are reproduced by our test suite. Depending on the SoC configuration/design, a deadlock may not occur even though the deadlock conditions are recreated. In this case, we can confirm that the deadlock conditions occurred through our coverage measurement results, and that the SoC design is free of deadlocks.

Figure 5 shows the block diagram of the Deadlock Detection Verification Environment. If a deadlock actually occurs, we can use the deadlock diagnostic features described below to identify and debug the cause of the deadlock.

1. System Monitor: a status-check timer is used to generate periodic interrupts to the CPU; when this interrupt occurs, the CPU checks the status of the system and sends a "heartbeat response" to the System Monitor. In case of SError (kernel panic), livelock, or other failures, the CPU will deliver a "failure response" instead of a "heartbeat response" to the System Monitor. If there is no "heartbeat response" from the CPU after a certain period of time, the System Monitor considers the CPU to be deadlocked and informs the System Tracker.
2. Transaction Latency Monitor: During the execution of a scenario, the Transaction Latency Monitor records the latency of all transactions originating from the CPU(s). If the System Monitor has detected a deadlock, the Transaction Latency Monitor will pass the information about the oldest transaction that has not completed to the System Tracker. With this information, the System Tracker will be able to infer which transaction and which master caused the deadlock.
3. System Tracker: if the System Monitor has detected a deadlock, the System Tracker will start gathering the following information for root cause analysis.
    - CPU status (PC, GPR, PSTATE, Fault status, Exception/Error syndrome register, etc.)
    - Current test scenario
    - Information about the suspected transaction
    - Estimated time when the deadlock occurred.

   After gathering this information, the System Tracker will create a log file and end the simulation.
4. Simulation Manager: manages the whole process of creating and executing the Perspec scenarios for simulation and analyzes the results. After the simulation has finished, the Simulation manager will phase the log file from the System Tracker specifically for the deadlock case and will perform a waveform dump around the point where the deadlock occurred using the save & restore.

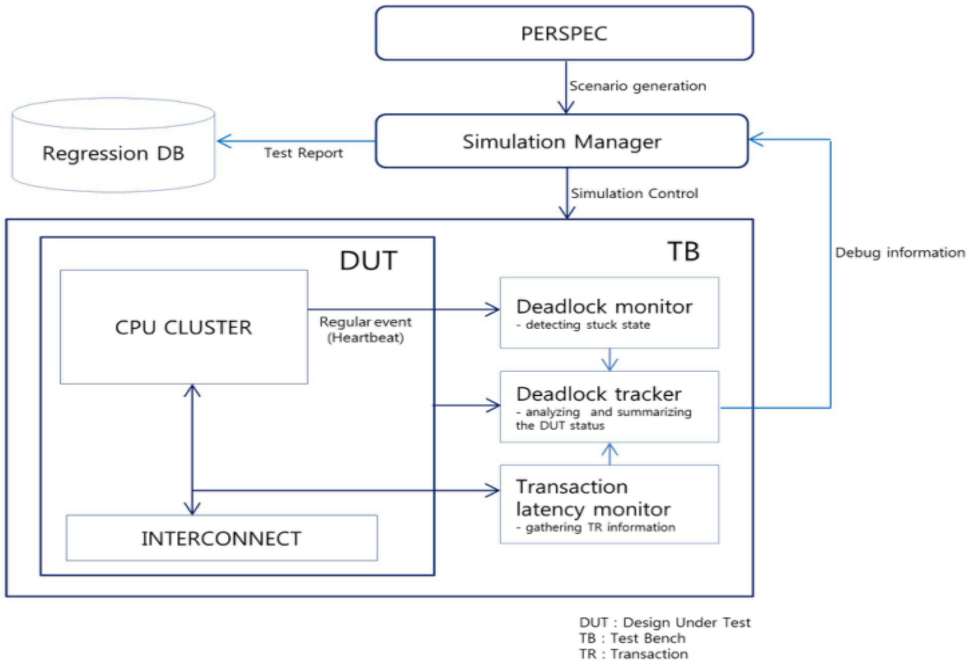


Figure 5: Deadlock Detection Verification Environment

V. PSS SCENARIOS TO DETECT PCIE DEADLOCK CONDITIONS

According to the PCIe ordering rules, non-posted requests (i.e. reads and configuration writes) to a PCIe slave interface might be stalled while a posted write from the corresponding PCIe master interface is stalled. This is because a read or write completion is unable to overtake a posted write that is unable to leave the master interface. This potentially can cause a deadlock condition when the right combination and timing of the PCIe transactions and cache snoop transactions occur [7][8]. There are design guidelines from the processor core vendor on how to prevent deadlocks; however, we do need to verify that our design is free of deadlock.

The test scenarios to verify that a design is free of deadlock are difficult to write manually due to the multiple parallel activities from different masters and slaves and the variation of timing of the transactions. Figure 6 shows the block diagram of a design that we will use to illustrate a simple scenario that can create deadlock.

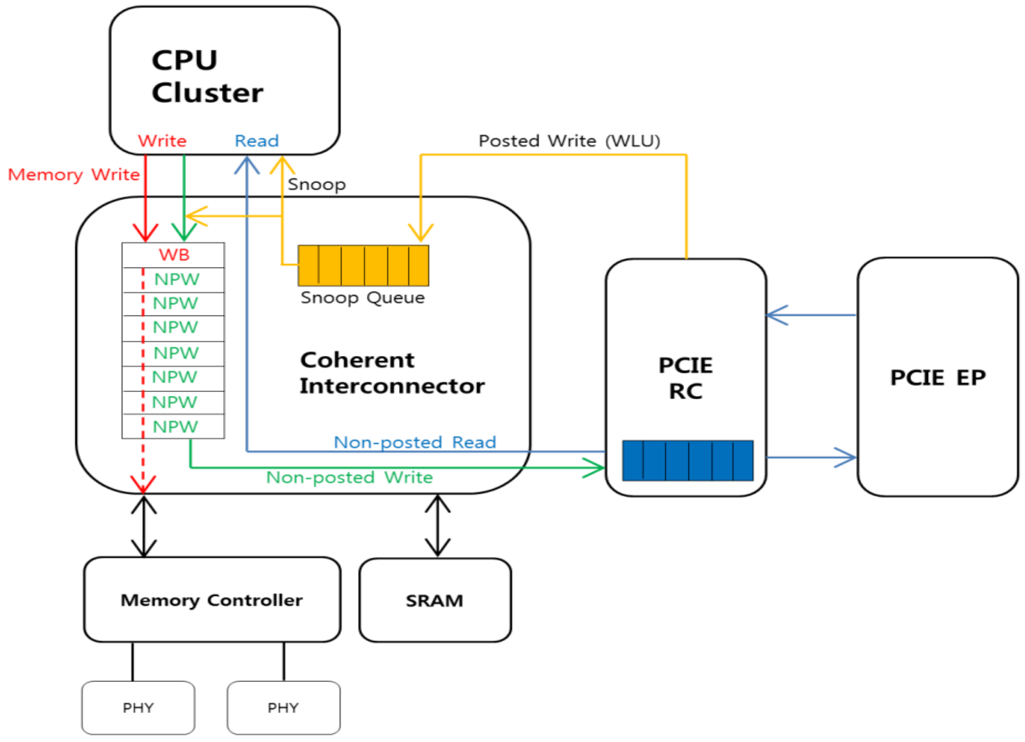


Figure 6: Block diagram of a design used to illustrate potential PCIe deadlock condition

A possible deadlock condition that can occur in figure 6 is as follows:

1. CPU cluster performs a number of non-posted reads from a PCIe Endpoint(EP). These non-posted read requests will fill up the PCIe RC request buffer.
2. CPU cluster performs a number of non-posted config writes to the PCIe Endpoint(EP). These can't be accepted because the non-posted request channel is now saturated due to the previous non-posted reads (in step 1).
3. CPU cluster generates a Writeback of address A. Writeback can't proceed because the write channel has been blocked.
4. PCIe Endpoint issues a read of address A. If PCIe EP issues a ReadOnce transaction for address A, the coherent interconnector will generate a ReadOnce snoop transaction to the CPU cluster. But this snoop can't proceed because of the outstanding WriteBack.

5. PCIe Endpoint issues a number of coherent reads. These ReadOnce transactions will fill up the snoop queue.
6. PCIe Endpoint issues a coherent write(WLU) of address A. WriteLineUnique(WLU) transaction will generate Clean&Invalidate snoop before memory update. but this snoop can't deliver to the CPU cluster because the snoop queue has been saturated.
7. Deadlock is now achieved.

As you can see, even for this simple deadlock test scenario, it's not easy to create a test manually. However, using PSS and Perspec, the code is pretty simple since the PSS syntax makes it easy to describe the scheduling of different actions and Perspec handles all the resource scheduling and constrained randomization automatically. The pseudo code for this scenario is as follows:

```
//Setup step: setup PCIe RC and PCIe EP
do pcie_config;
do pcie_rc_mem_write;
do write_cache;
parallel { //randomly pick a core to do the following actions
     do pcie_rc_mem_read multiple times in sequence; //core A3 in Fig 7
     do pcie_rc_mem_read multiple times in sequence; //core A0
     do pcie_rc_mem_read and invalidate_cache;       //core A1
     do pcie_rc_mem_read and pcie_config_write;      //core A2
     //PCIe EP
        sequence {
           do pcie_ep_mem_write;
           do pcie_ep_mem_read multiple times;
        };
};
```

Figure 7 shows one of the tests generated from the above scenario. Note that the "Setup step" is not shown in figure 7 since this step is straightforward; the UML diagram is smaller and is easier to understand without the "Setup step".

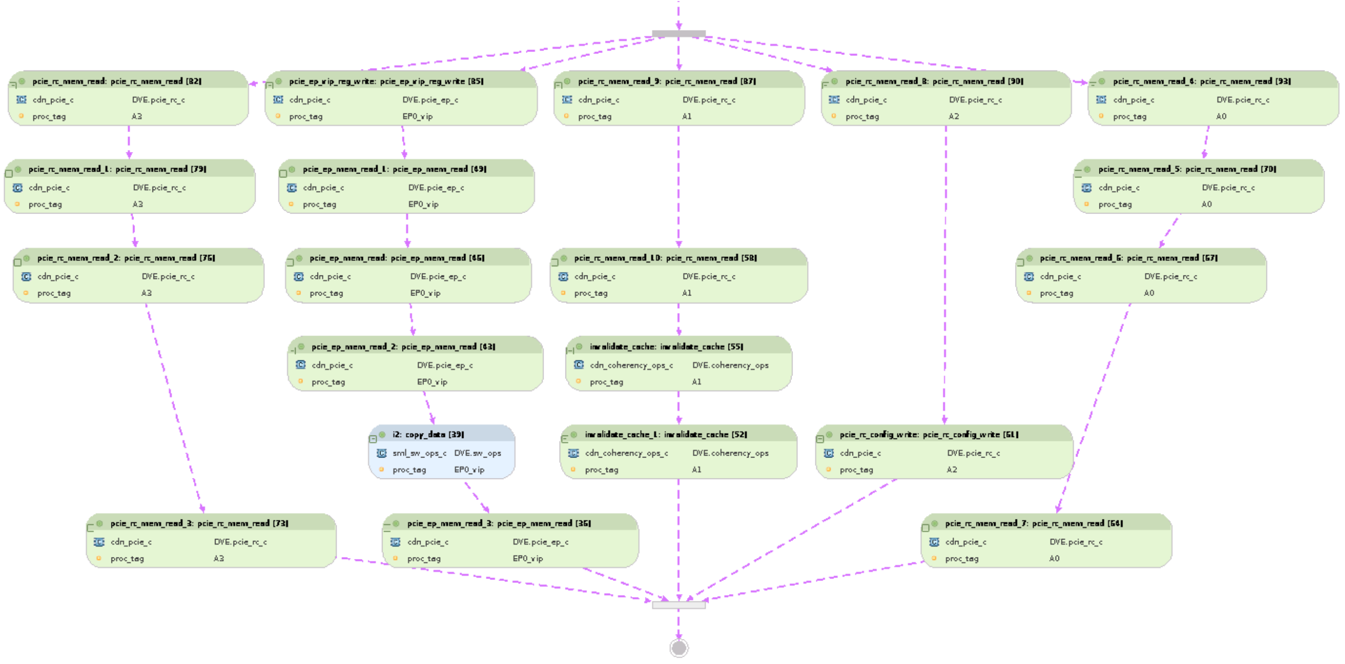

pcie_rc_mem_read: pcie_rc_mem_read [82]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A3
pcie_ep_vip_reg_write: pcie_ep_vip_reg_write [85]
cdn_pcie_c DVE.pcie_ep_c
proc_tag EP0_vip
pcie_rc_mem_read_9: pcie_rc_mem_read [87]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A1
pcie_rc_mem_read_8: pcie_rc_mem_read [90]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A2
pcie_rc_mem_read_4: pcie_rc_mem_read [93]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A0
pcie_rc_mem_read_1: pcie_rc_mem_read [79]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A3
pcie_ep_mem_read_1: pcie_ep_mem_read [49]
cdn_pcie_c DVE.pcie_ep_c
proc_tag EP0_vip
pcie_rc_mem_read_5: pcie_rc_mem_read [70]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A0
pcie_rc_mem_read_2: pcie_rc_mem_read [76]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A3
pcie_ep_mem_read: pcie_ep_mem_read [46]
cdn_pcie_c DVE.pcie_ep_c
proc_tag EP0_vip
pcie_rc_mem_read_10: pcie_rc_mem_read [58]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A1
pcie_rc_mem_read_6: pcie_rc_mem_read [67]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A0
pcie_ep_mem_read_2: pcie_ep_mem_read [43]
cdn_pcie_c DVE.pcie_ep_c
proc_tag EP0_vip
invalidate_cache: invalidate_cache [55]
cdn_coherency_ops_c DVE.coherency_ops
proc_tag A1
i2: copy_data [39]
sml_sw_ops_c DVE.sw_ops
proc_tag EP0_vip
invalidate_cache_1: invalidate_cache [52]
cdn_coherency_ops_c DVE.coherency_ops
proc_tag A1
pcie_rc_config_write: pcie_rc_config_write [61]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A2
pcie_rc_mem_read_3: pcie_rc_mem_read [73]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A3
pcie_ep_mem_read_3: pcie_ep_mem_read [36]
cdn_pcie_c DVE.pcie_ep_c
proc_tag EP0_vip
pcie_rc_mem_read_7: pcie_rc_mem_read [64]
cdn_pcie_c DVE.pcie_rc_c
proc_tag A0

Figure 7: One of the generated test from the PCIe deadlock scenario

VI. Results & Lessons Learned from an Actual Project

We have completed the generation and running of the coherency test suites on our design; this task was completed quickly since we took advantages of the capabilities provided by the Perspec coherency library and we were also able to re-use the Perspec scenarios developed on previous project. Regarding the PCIe deadlock scenarios, we are in the process of running these scenarios to verify that our design is free of deadlocks. We are also developing more PCIe scenarios to verify our design further at the SoC-level. We should be able to discuss the final results at DVCon 2019 in February.

Preliminary results showed that Perspec and PSS helped us shorten the design verification cycle significantly; this was especially true in the case of coherency verification due to the availability of the Perspec coherency library and the reuse of the Perspec scenarios from previous project. We also "learned" that writing PSS scenarios were a lot simpler than writing C code manually; also, the Perspec library helped with the reuse of test scenarios from project-to-project since it automatically creates the data types, structures, and model of the processor and memory subsystems. To create reuse scenarios, we did have to make a conscious effort not to refer to the actual name of the processor/memory resources in these scenarios since these will change from project-to-project. One of the lessons that we learned from this is that to create "portable stimulus" scenarios, we need more than the PSS language; we also need methodology and library; this is similar to UVM where a methodology for building reusable verification component is also part of the standard.

VII. Summary

This paper shows how PSS and EDA tools, such as Perspec, can be used to speed-up the design verification of a multi-core SoC in two specific areas: memory coherency and deadlock detection. The main technical contributions are:

- Application of PSS to typical SoC-level verification tasks
- Horizontal (project-to-project) reuse methodology with PSS
- Deadlock detection verification environment

References

[1] Accellera, Portable Test and Stimulus, Version 1.0, (June 2018); http://accellera.org/downloads/standards/portable-stimulus

[2] Fong Pong and Michel Dubois, "Verification Techniques for Cache Coherency Protocols", ACM Computing Surveys, March 1997

[3] M. Fromovich and T. Meshulum, "Building Your UVM Verification Environment for Cache Coherent Interconnects", https://www.design-reuse.com/articles/31757/uvm-verification-environment-for-cache-coherent-interconnects.html

[4] R. Ranjan, "Verifying Cache Coherence", EE Times, https://www.eetimes.com/document.asp?doc_id=1279017

[5] A. Hamid, D. Koogler, T. Anderson, "Using Portable Stimulus to Verify Cache Coherency in a Many-Core SoC", DVCon 2016

[6] Deadlock and livelock: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0425/ch04s07s02.html

[7] ARM CoreLinkTM CCI-550 Cache Coherent Interconnect, Revision:r0p1, Technical Reference Manual

[8] ARM-EPM-107877 Considerations for Integrating PCI Express with CCI-500 and CCI-550