Post-Silicon Performance Validation Using PSS

Dayoung Kim, Jaehun Lee, Daeseo Cha
Samsung Electronics,1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do 18488, Korea
d131.kim@samsung.com, jaehunn.lee@samsung.com, dscha@samsung.com,

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

***Abstract*- There are many challenges in post-silicon validation[1][2] and these challenges are not getting any easier due to the increase in the complexities of modern SoC designs. One aspect of this is the increase in the number of embedded processor cores in the SoC as well as the firmware contents needed for proper operation of the design; therefore, putting more demands on the time and resources required to create sufficient embedded test software as part of the post-silicon performance validation effort; the embedded test software will need to verify for proper operation, communication, synchronization, and performance of the processor subsystem, the memory subsystem, and the I/O subsystems in the design. In this paper we will discuss the challenge of verifying multiple versions of our test chip, implemented using advanced design implementation methodology exploring cell libraries, on-chip monitoring IPs and new physical implementation techniques, and how we used the Portable Test and Stimulus Standard (PSS)[3] in addressing this challenge. We will also discuss the lesson learned from this project, the advantage of using PSS, and possible future enhancements.**

I. INTRODUCTION

New semiconductor technologies and design methodology improvements need to be proven in a test chip before mass production. These improvements consist of library cells, on-chip monitor IPs, and newly developed physical implementation techniques to enhance performance and power consumption in SoC design. The test chip is fabricated and used as the vehicle for evaluating the effectiveness of the new technologies and design methodologies. The information collected through the test chip is used to improve our semiconductor technologies and design methodology continuously and to help our customers with making the design tradeoffs in selecting the technologies used to implement their designs.

Fig. 1 shows the architecture of our test chip; there are five CPU blocks in the design: BLK_CPU0 to BLK_CPU4; each CPU block has quad-cores. New design methodologies are implemented on the CPU blocks and the improvement in terms of “Power, Performance, Area” (PPA) are measured by de-facto CPU benchmark programs and long-running stress tests; each CPU block is implemented using different cell libraries and physical implementation techniques to support different set of requirements. Since this test chip is built on a platform, the number of CPU blocks can easily be extended and system IP like CMU/PMU can also be adjusted automatically.

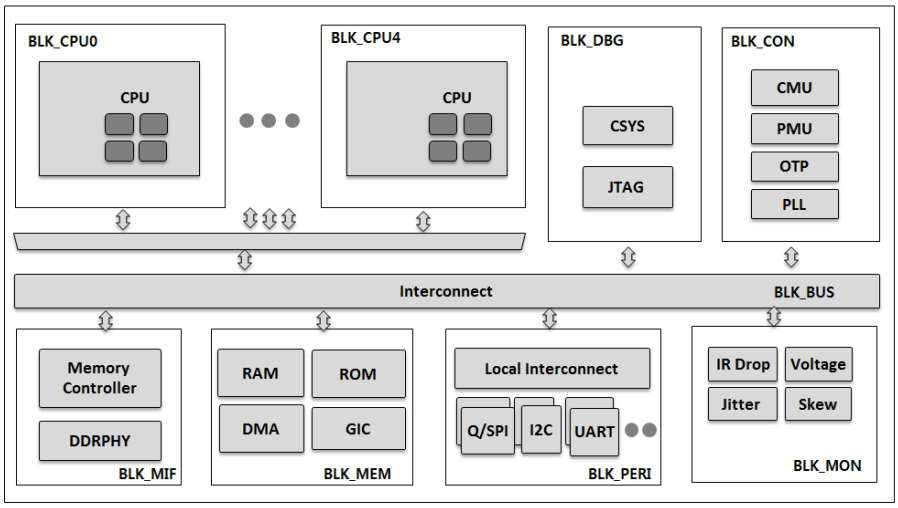

Figure 1. Test-chip (TC) Block Diagram

Verification Requirements and Solutions: One of the goals of our test chip is to use it as the vehicle for evaluating the performance and power consumption of targeted design methodology. This requires us to create test scenarios that can be adapted to different configurations as well as different implementations of the test chip. To simplify this task, we decided to use PSS and Perspec to create re-usable test scenarios and let the tools handle the randomization, resource allocation, synchronization, and actual C code generation. In the remaining sections of the paper, we will give more detailed description of our test scenarios and sample PSS implementation of these scenarios; we will also discuss our experience with using PSS and our estimate of the improvement in the productivity as compared to manual coding. Finally, we will discuss the lessons learned and how we can incorporate more PSS in our post-silicon performance validation effort for future projects.

II. Approach & Implementation

We followed a 4-step approach for performance validation of our test chip using PSS:

A. **Creating test scenarios to measure performance**: It is important to have suitable test scenarios targeting the differentiations of each CPU block. The implementation details like STA (Static Timing Analysis) timing report are used to determine the clock speed and maximum bandwidth when generating a test scenario. A Traffic Profile Generator (TPG) developed using PSS, generates intended test scenarios easily.

B. **Creating de facto standard benchmark test**: Dhrystone, coremark and lmbench are widely used CPU benchmark programs for performance. Gzip has frequently been used in measuring the performance of monitoring IPs. These benchmark programs have been implemented in PSS as part of the tool library; this simplifies the task of incorporating these programs into our performance test suite. The performance differences due to the differences in design methodology can be recognized clearly with credible benchmark tests.

C. **Synthesizing long-running test scenario**: Since the performance measurement will be measured on PCB board, the test scenarios need to be sufficiently long. They can be synthesized by combining the simpler and shorter tests in our test plan; these scenarios are also put in run-time loops to further increase the test duration.

D. **Automated and reusable PSS environment**: In future versions of our test chip, the number of CPU blocks will vary depending on the requirements. The current PSS environment is designed to be reusable without any modification.

The following sections describe how these test scenarios are implemented in PSS for our current test chip. Table 1 and table 2 show the processor and memory configurations of our current test chip.

Core Name	Cluster Name	Cluster ID	Core ID
A0	CL0	0	0
A1	CL0	0	1
A2	CL0	0	2
A3	CL0	0	3
B0	CL1	1	4
B1	CL1	1	5
B2	CL1	1	6
B3	CL1	1	7
C0	CL2	2	8
C1	CL2	2	9
C2	CL2	2	10
C3	CL2	2	11
D0	CL3	3	12
D1	CL3	3	13
D2	CL3	3	14
D3	CL3	3	15
E0	CL4	4	16
E1	CL4	4	17
E2	CL4	4	18
E3	CL4	4	19

Table 1: Processor configuration of current version of test chip

Memory Block Name	Base Address	End Address
IRAM	0x0005_0000	0x0007_0FFF
DDR1	0x8000_2000	0x9FFF_FFFF

Table 2: Memory configuration of current version of test chip

A. Test scenarios to measure performance

To measure memory read/write performance from the CPU clusters, we created memory traffic profile (TP) scenarios that have one or more cores accessing the memory concurrently. Some of the key parameters that we want to vary for these scenarios are:

- Number of processor cores to participate in the test
- Direction: read only, write only, random selection
- Bandwidth in megabytes/sec
- Number of transactions

The memory TP scenario can be broken down into simpler sub-scenarios; each sub-scenario has exactly one processor core performing memory access. The sub-scenario is implemented as a PSS action, `sml_traffic_profile` action, with the following control knobs:

- Processor core to execute this action
- Direction: read only, write only, random selection
- Bandwidth in megabytes/sec
- Number of transactions

Now that we have the base traffic profile action: `sml_traffic_profile`, we can implement our memory TP scenario; Fig. 2 shows the PSS code for the memory TP scenario.

```
action multicore_sml_tp {
   rand sml_proc_subset_select_s procs_subset;
   rand sml_atp_direction direction;
   rand bit[32] mbps;      //in MB/s
   rand bit[32] numOfTransaction;

   activity {
      parallel {
         foreach (procs_subset.selected[index]) {
            do sml_traffic_profile with {
               proc_tag == procs_subset.selected[index];
               mbps == this.mbps;
               direction == this.direction;
               numOfTransactions == this.numOfTransactions;
            }
         }
      }
   }
}
```

Figure 3. Memory TP Scenario

The `multicore_sml_tp` scenario allows the test writer to select a number of processor cores to execute the `sml_traffic_profile` action in parallel. As you can see from the PSS code of our memory TP scenario in Fig. 3, the `multicore_sml_tp` action has the following control knobs (shown in **bolded** font in the code):

- **`procs_subset`**: this data structure consists of multiple control knobs; these control knobs allow the test writer to select one or more processor cores to be used for this scenario.
- **`direction`**: specify memory read or write transactions
- **`mbps`**: specify the bandwidth in megabytes per second
- **`numOfTransactions`**: specify the number of read/write transactions

Note that the **`procs_subset`** control knob is of type `sml_proc_subset_select_s` which consists of the following key control knobs:

- `int size`: how many cores (1 to 20 for our test chip)
- `bool A0_selected`: whether A0 is selected or not
- `bool A1_selected`: whether A1 is selected or not
- …
- `bool E2_selected`: whether E2 is selected or not
- `bool E3_selected`: whether E3 is selected or not
- `selected[]`: a list contains the cores selected by the test writer (through other control knobs)
- `int num_CL0`: number of cores in cluster CL0
- `int num_CL1`: number of cores in cluster CL1
- `int num_CL2`: number of cores in cluster CL2
- `int num_CL3`: number of cores in cluster CL3
- `int num_CL4`: number of cores in cluster CL4

Note that these control knobs can be created automatically from table 1 by a PSS tool; this is how the type `sml_proc_subset_select_s` was created by the PSS tool.

Using these control knobs, the test writer can easily specify memory traffic profile for different CPU clusters. Fig. 4 and Fig. 5 show how the test writer can constrain the memory TP scenario for cluster 0 (CL0) and cluster 4 (CL4).

```
action tp_cl0 {
  activity {
     do multicore_sml_tp with {
        proc_subset.size == 4;       //total no. of cores = 4
        proc_subset.num_CL0 == 4;  //use all 4 cores in CL0
     }
  }
}
```

Figure 4. Memory TP Scenario for Cluster 0

```
action tp_cl4 {
  activity {
     do multicore_sml_tp with {
        proc_subset.size == 4;       //total no. of cores = 4
        proc_subset.num_CL4 == 4;  //use all 4 cores in CL4
        direction == SML_ATP_WRITE;
        mbps == 800;
        numOfTransactions == 12;
     }
  }
}
```

Figure 5. Memory TP Scenario for Cluster 4

Fig. 6 shows one of the tests generated for the memory TP scenario specified in Fig. 5. All four cores in CL4: E0, E1, E2, E3 are performing memory write operation at a rate of 800 megabytes/sec; profile_delay action in the UML diagram determines the delay between write operations so that the specified rate is achieved. The loop of 12 was specified by the value of `numOfTransactions` in the `tp_cl4` action.

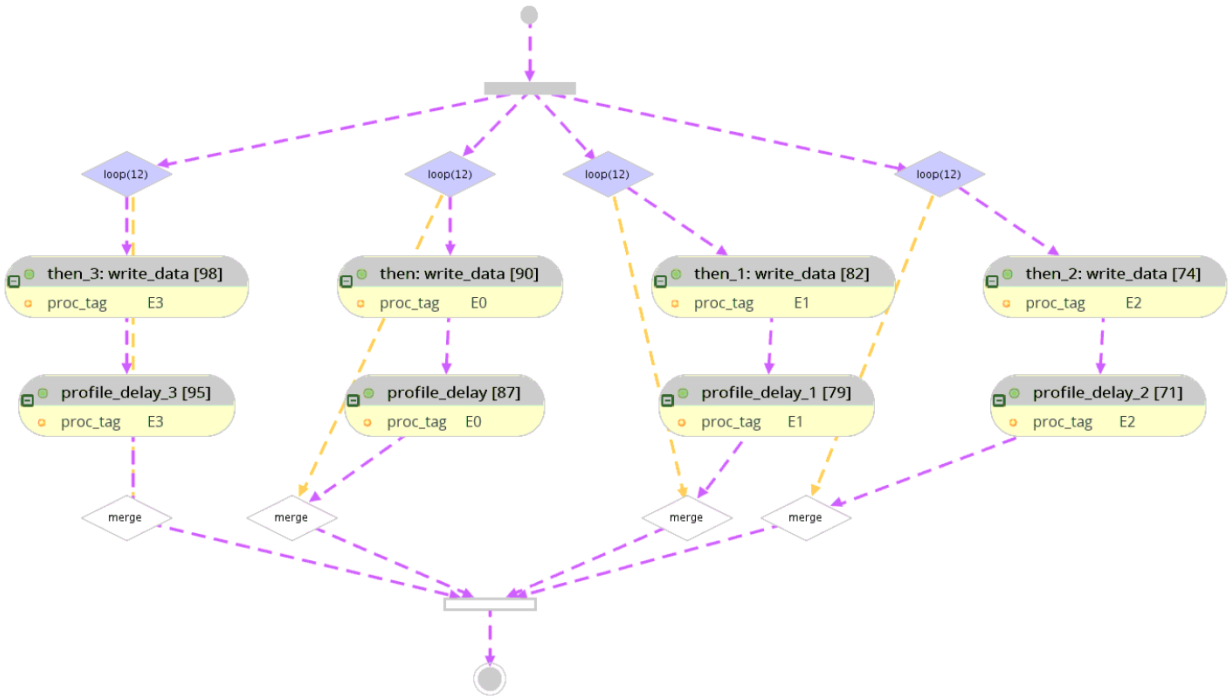

Figure 6. One of the tests generated for the memory TP scenario specified in Fig. 5

B. *Standard benchmark test*

Performance for each CPU cluster should be measured individually to evaluate the effectiveness of the technology/methodology used to implement that particular CPU cluster. It is a useful and credible way to use de facto standard benchmark tests to measure the performance index. We selected four standard benchmark tests for our project: dhrystone, coremark, lmbench and gzip. Dhrystone and coremark are good measurements of the general-purpose CPU performance; lmbench provides good measurements of the memory bandwidth and latency performance; gzip benchmark can figure out the efficiency on both CPU and memory system.

To use these benchmarks in PSS scenarios, we "wrapped" the open source code in PSS actions. Key attributes of these benchmarks, such as processor core, memory block, etc., are provided as control knobs of these actions. This simplifies the task of creating benchmark tests for multiple processor cores and different memory configurations. The benchmarks can also be combined with other test scenarios, such as DMA scenarios, to evaluate the performance of the CPU cluster when there are other system activities.

To measure the performance, we also created two other actions:

- `start_pm_counter`: this action resets the ARM "Performance Monitors Cycle Count" [5] register (PMCCNTR) and enables it to start counting.
- `stop_pm_counter`: this action stops the ARM "Performance Monitors Cycle Count" register and returns its current value.

Fig. 7 shows the PSS code for a simple benchmark scenario. In Fig. 7, the coremark_scenario starts with resetting and enabling the ARM PMCCNTR; it then runs the coremark; once the coremark is done, it stops the PMCCNTR and gets the total cycles for this test. The `coremark` action can be replaced with other benchmark action to run a different benchmark. More complex PSS scenario can be written to allow the test writer to select which of the benchmark actions to execute or to combine the benchmark actions with other system traffic.

```
action coremark_scenario {
   rand sml_processor_tag_e proc_tag;

   activity {
      sequence {
         do start_pm_counter with {
            proc_tag == this.proc_tag
         }
         do coremark with {
            proc_tag == this.proc_tag
         }
         do stop_pm_counter with {
            proc_tag == this.proc_tag
         }
      }
   }
}
```

Figure 7. Coremark scenario

C. *Long-running test scenario*

One of the main purpose of post-Silicon validation is to check for design correctness, whether the design meets the intent or not. It needs to be done with the real use-case scenarios on PCB board. Traditionally, these use-case scenarios were created manually and they usually didn't have very good coverage. In order to improve on this, we leverage PSS to generate more tests and to get better coverage using randomization and automation provided by the PSS tools. For post-silicon validation, we can run much longer tests and these long running test scenarios are preferred because they can uncover hidden bugs that are not detectable with short tests.

Our test chip provides the complete system connectivity among interconnect, memory sub-system LPDDR4, internal RAM/ROM and the multi-ports peripherals such as UART/I2C/SPI/QSPI while running CPU workload. To check the system connectivity thoroughly, we need to activate as many H/W resources as possible at the same time; this helps us find corner case problems as well as measure power consumption. Fig. 8 shows the block diagram describing system connectivity. For example, DMA0 moves data from LPDDR4 to UART for display and DMA1 moves data from QSPI/SPI to LPDDR4 for reading storage data while CPU is performing algorithm via SRAM/LPDDR4.

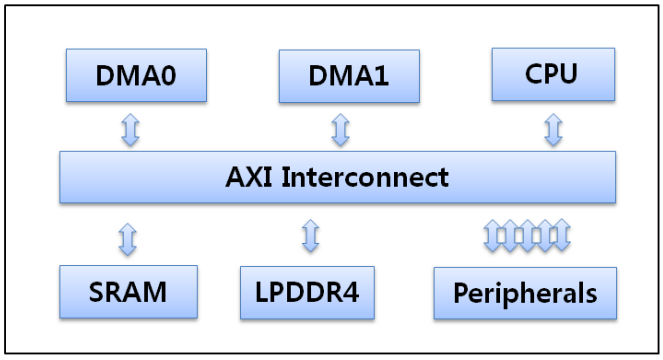

Figure 8. Test chip system connectivity/data paths

The challenge of creating these long running test scenarios with multiple bus masters running concurrently is to resolve the resource conflicts, memory allocation, inter-processor communication and synchronization. In addition, in order to facilitate the creation of long running and meaningful tests, we also need a way to compose more complex scenarios from simpler scenarios. Using PSS, many of these issues were handled automatically for us; as a result, we can generate corresponding long running test easily and extend it to new test scenario quickly.

Fig. 9 shows PSS code for a DMA and memory traffic scenario. In this `dma_tp` scenario, the test writer can select the processor cluster to run the test. The constraints (not shown for brevity) in this scenario will ensure that only the cores in the specified cluster are used to execute this scenario. In this case, one of the cores will be used to execute the following actions `start_pm_counter`, `dma_scenario`, `stop_pm_counter`: the other three cores are used to execute the memory TP scenario `multicore_sml_tp`. Note also that the `dma_tp` scenario is constructed from simpler scenarios (actions) that we developed and discussed in previous sections.

```
action dma_tp {      //perform DMA and memory TP in parallel
   rand sml_cluster_tag_e cluster_tag;
   rand sml_processor_tag_e proc_tag1;
   rand sml_processor_tag_e proc_tag2;
   rand sml_processor_tag_e proc_tag3;
   rand sml_processor_tag_e proc_tag4;
   //constraints to ensure the proc_tag? are unique and are
   //…consistent with cluster_tag; example:
   //   if cluster_tag==CL1 then proc_tag? are in [B0,B1,B2,B3]
   activity {
      sequence {
         do start_pm_counter with {
            proc_tag == proc_tag1;
         }
         parallel {
            do dma_scenario with {
               proc_tag == proc_tag1;
            }
            do multicore_sml_tp with {
               //cores selected for this action: proc_tag2,3,4
               //   constraints block to select the cores
            }
         }
         do stop_pm_counter with {
            proc_tag == proc_tag1;
         }
      }
   }
}
```

Figure 9. DMA and Memory TP Scenario

Fig. 10 shows the PSS code for the dma_tp scenario running on cluster CL0; Fig. 11 shows the UML diagram for one of the generated tests for this scenario.

```
action dma_tp_cl0 {
  activity {
     do dma_tp with {
        cluster_tag == CL0;
     }
  }
}
```

Figure 10. Running dma_tp scenario on cluster CL0

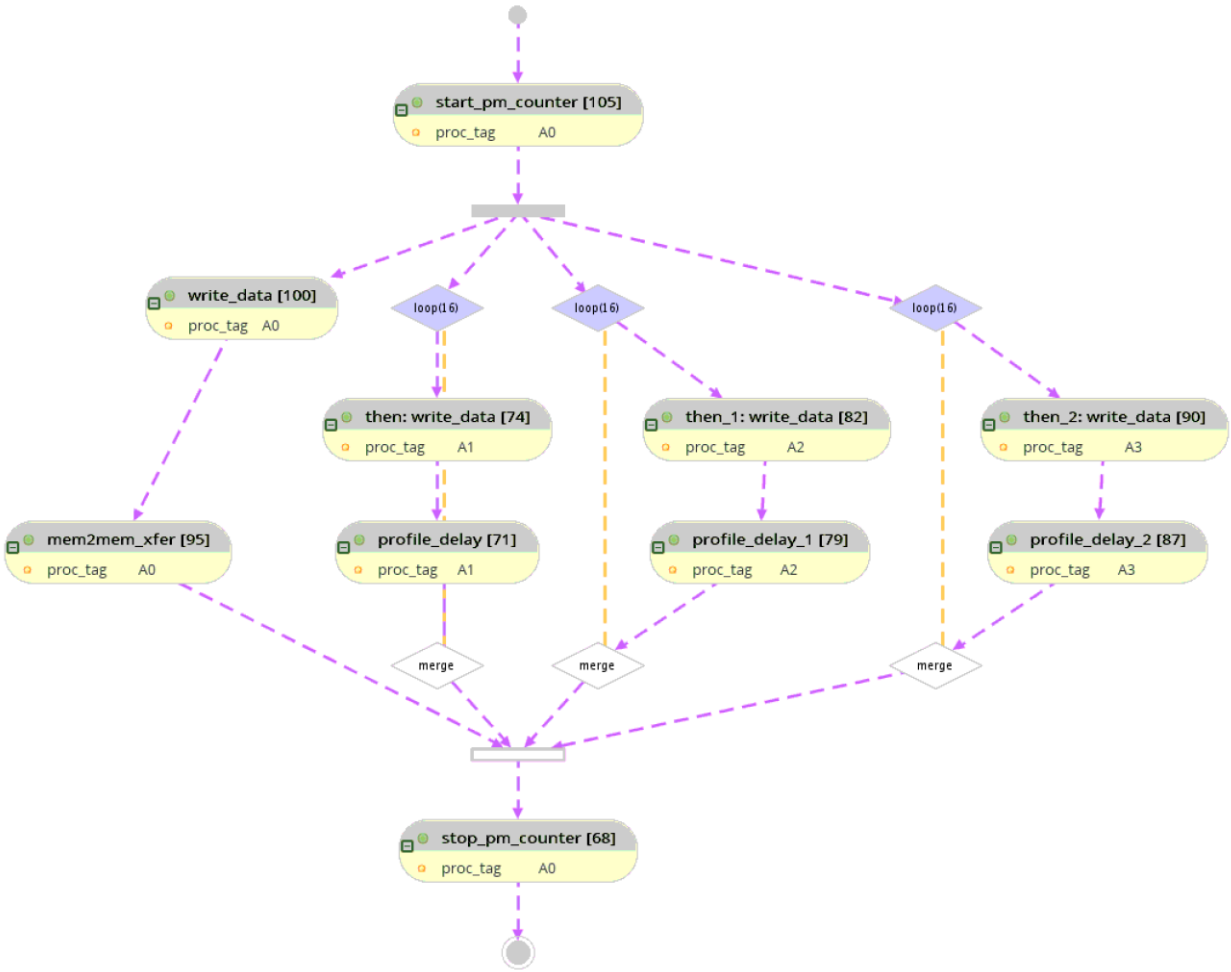

Figure 11. UML diagram of a `dma_tp` test on cluster CL0

Sample C Code generation:

The UML diagram in Fig. 11 shows that four cores are used for this scenario. Note that there are implicit synchronization points shown in this diagram:

- Sync point 1: At the beginning of the test, cores A1, A2, A3 have to wait for core A0 to complete the execution of the `start_pm_counter` action before they can execute the first `write_data` action.
- Sync point 2: At the end of the test, core A0 has to wait for cores A1, A2, A3 to complete the execution of their TP scenario (which consist of executing a sequence of `write_data` and `profile_delay` in a loop of 16 times) before it can execute the `stop_pm_counter` action.

Fig. 12 shows the code snippets for core A0. Fig. 13 shows the code snippets for core A1. The IPC (inter-processor communication) messages for the two sync points are annotated in the code.

In Fig. 12, the two sync points are:

- Line 1063: core A0 sends the IPC message to cores A1, A2, A3; once receiving this message, these cores will start executing their scenarios.
- Lines 1094-1101: core A0 is waiting for core A1 to complete; after that A0 will need to wait for A2 and A3 to complete (lines 1102-1117; this code is not shown in Fig.12). Once cores A1, A2, and A3 finish executing their scenarios, core A0 will execute the code of the `stop_pm_counter` action (lines 1118-1125)

In Fig. 13, the code for the first sync point are in lines 1023-1030. The code for the second sync point is line 1042.

```
1048 /* Entry point for Perspec execution of thread A0 */
1049 void sln_main_A0(void) {
1050     /* Code for scenario: start_pm_counter [1] on thread A0 */
1051     {
1052         perspec_scenario_clean_mailbox_1_A0();
1053         /* Code for start_pm_counter [106] on thread A0 declared at line 82 in cdn_pslib_performance */
1054         {
1055             slnp_write_ints0_50062(72);/* Start action msg */
1056             {
1057                 start_counter(0);
1058             }
1059             Dmb();
1060             slnp_write_ints0_50062(73);/* End action msg */
1061             /* thread A0 sending a 'start action' message to repeat [113],repeat [115],repeat [117] */
1062             slnp_write_ints0_50062(74);
1063             perspec_local_ipc_send(A0_0_local_mbox, 3, 0);
1064         }
1065         /* Code for write_data [101] on thread A0 declared at line 207 in sml_sw_ops */
1066         {
1067             slnp_write_ints0_50062(75);/* Start action msg */
1068             {
1069                 {
1070                     uint8_t * dst_ptr = (uint8_t *)0x56a70;
1071
1072                     {   int li;
1073                         for (li=0; li < 2; li++) {
1074                             *(uint8_t volatile *) (dst_ptr++) = (uint8_t) 0xeb;
1075                             *(uint8_t volatile *) (dst_ptr++) = (uint8_t) 0x91;
1076                             *(uint8_t volatile *) (dst_ptr++) = (uint8_t) 0xac;
1077                             *(uint8_t volatile *) (dst_ptr++) = (uint8_t) 0x36;
1078                         }
1079                     }
1080                 }
1081             }
1082             slnp_write_ints0_50062(78);/* End action msg */
1083         }
1084         /* Code for mem2mem_xfer [96] on thread A0 declared at line 15 in dma */
1085         {
1086             slnp_write_ints0_50062(81);/* Start action msg */
1087             {
1088                 {
1089                     DMA_MAIN(dma_1, 0x56a70, 0x89fd8110, 4 ,16, 1);
1090                 }
1091             }
1092             slnp_write_ints0_50062(82);/* End action msg */
1093         }
1094         /* thread A0 waiting for a 'start action' message from thread A1 */
1095         slnp_write_ints0_50062(83);
1096         while (1) {
1097             int perspec_read_value = 0;
1098             perspec_read_value = perspec_local_ipc_receive(A1_1_local_mbox, 0);
1099             if (perspec_read_value) { break;}
1100             SLNP_YIELD();
1101         }
...          ...

1118         /* Code for stop_pm_counter [69] on thread A0 declared at line 89 in cdn_pslib_performance */
1119         {
1120             slnp_write_ints0_50062(86);/* Start action msg */
1121             {
1122                 unsigned long total_cycles;
1123
1124                 stop_counter(0);
1125                 total_cycles = get_total_cycles(0);
```

Sync Point 1

Sync Point 2

Figure 12. Generated C code for core A0 (`dma_tp` test on cluster CL0)

```
1017 /* Entry point for Perspec execution of thread A1 */
1018 void sln_main_A1(void) {
1019     perspec_run_start_code_A1();
1020     /* Code for scenario: start_pm_counter [1] on thread A1 */
1021     {
1022         perspec_scenario_clean_mailbox_1_A1();
1023         /* thread A1 waiting for a 'start action' message from thread A0 */
1024         slnp_write_ints0_50061(64);
1025         while (1) {
1026             int perspec_read_value = 0;
1027             perspec_read_value = perspec_local_ipc_receive(A0_0_local_mbox, 0);
1028             if (perspec_read_value) { break;}
1029             SLNP_YIELD();
1030         }
1031         /* Code for repeat [113] on thread A1 declared as decision node */
1032         slnp_write_ints0_50061(65);/* Start action msg */
1033         {
1034             for (perspec_batch.scenario.loop_counter_113 = 0; perspec_batch.scenario.loop_counter_113 < 16; per
1035                 slnp_write_ints1_50061(67, perspec_batch.scenario.loop_counter_113+1);
1036                 perspec_scenario_2_A1();
1037                 slnp_write_ints1_50061(68, perspec_batch.scenario.loop_counter_113+1);
1038             }
1039             slnp_write_ints0_50061(69);/* End action msg */
1040             /* thread A1 sending a 'start action' message to stop_pm_counter [69] */
1041             slnp_write_ints0_50061(71);
1042             perspec_local_ipc_send(A1_1_local_mbox, 1, 0);
1043         }
1044     }
1045
1046     perspec_run_end_code_A1();
1047 }
```

Sync Point 1

Sync Point 2

Figure 13. Generated C code for core A1 (dma_tp test on cluster CL0)

As you can see from the code snippets in Fig. 12 and Fig.13, our PSS tool handled all the IPC and synchronization automatically; we did not have to worry about any details related to the C code generation. Using PSS, our productivity increased significantly since most of the time we worked at the "action-level"; we designed, analyzed, and debugged our test scenarios at the "action-level" using the UML diagrams and the debug tools provided by Perspec. If a test failed, we could isolate the issue to a certain action quickly by reviewing the "test progress" messages; once the failed action was identified, the next step was to review the generated C code for that action; this was the only time that we needed to work at the "C-level".

Post-silicon Tests:

For post-silicon tests, we want to create very long running tests; we used the PSS `repeat` statement to execute the scenarios that we used in simulation multiple times. Fig. 14 shows the PSS code for a post-silicon scenario that runs the DMA and the memory TP scenarios on a specific processor cluster; Fig. 15 shows the UML diagram of one of the tests for this scenario running on cluster CL0; note that, in Fig. 15, the PSS `repeat` statement generated a scenario that repeatedly execute the `dma_tp` scenario 9999 times (the outer loop).

```
action si_dma_tp {  //post-si scenario: dma_tp on a cluster
   rand sml_cluster_tag_e cluster_tag;
   rand int count;

   activity {
      repeat (count) {
         do dma_tp with {
            cluster_tag == cluster_tag;
         }
      }
   }
}
```

Figure 14. Post-Silicon - DMA and Memory TP Scenario

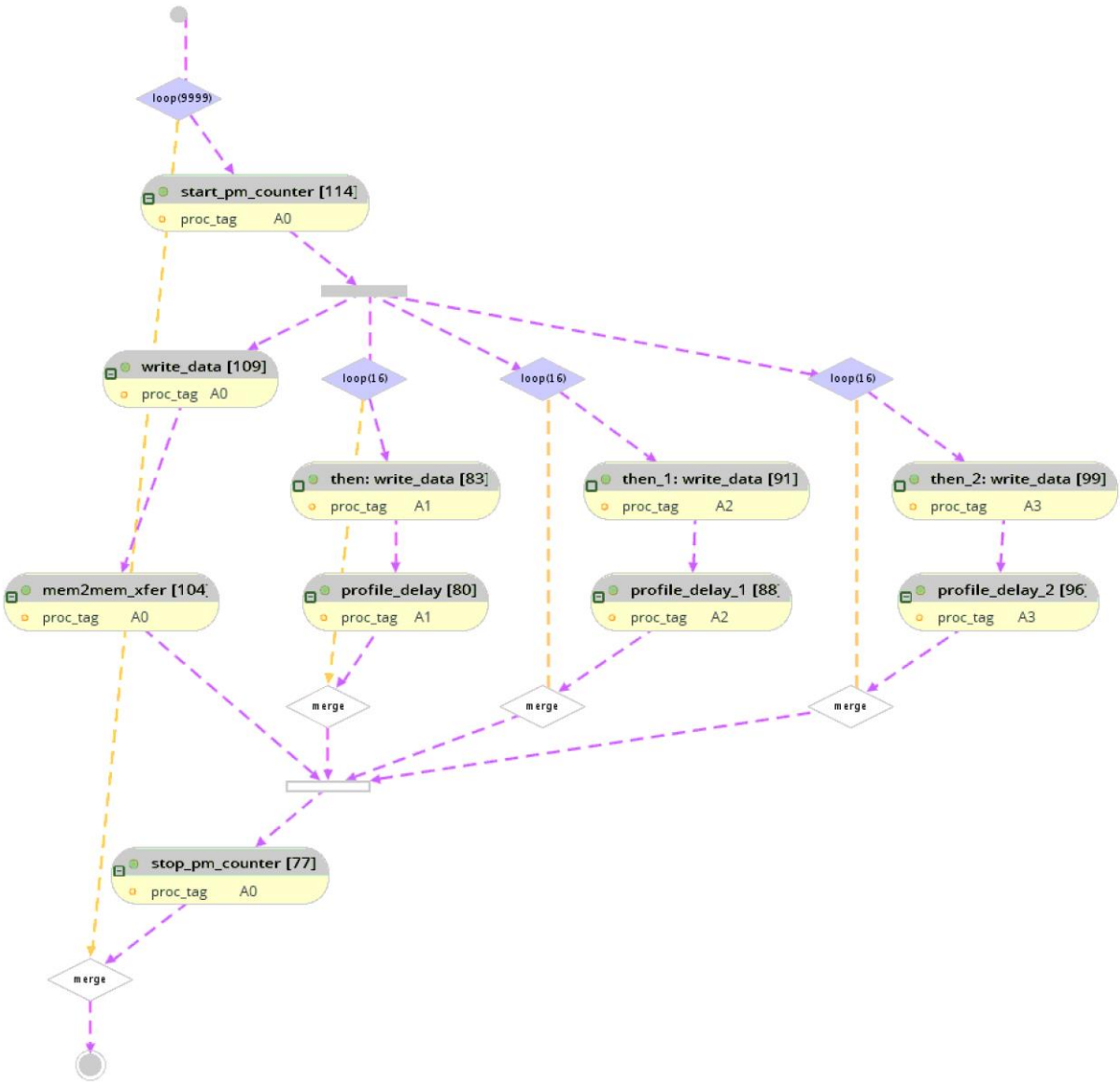

Figure 15. Post-Silicon – UML diagram of DMA and Memory TP test on cluster CL0

D. Reuse of PSS environment and scenarios

It is a redundant work to develop the PSS environment every time the test chip changes. We need to consider the reusability for future projects while developing the current PSS environment. Several components will be changed in the next version of our test chip, such as the number of CPU clusters, number of DMA and peripherals. Memory configuration can also vary. Therefore, all the project-dependent configurations are put in a configuration file, written in CSV format. The configuration file is organized as tables that contain project-dependent information of the test chip. Some typical configuration tables are: Processor table, Memory table, Page table, etc; table 1 and table 2 show the type of information contain in the Processor configuration table and the Memory configuration table.

The PSS tool (Perspec in our case) reads the configuration file and automatically generates the PSS model of the processor and memory subsystems; this model consists of common data types, control knobs, data structures to represent the state of the system, and helper functions to access the model primitives and their state. To create reuseable PSS atomic actions and scenarios (i.e., compound actions), we don't refer to project-specific attributes; we used the common data types, buffers, and control knobs generated from the configuration file. For example, in the `dma_tp` scenario in Fig. 10, we used the following data types: `sml_cluster_tag_e` and `sml_processor_tag_e`;

these data types are generated by the PSS tool from table 1; the values for `sml_cluster_tag_e` (for the current version of our test chip) are: CL0, CL1, CL2, CL3. For the next version of the test chip we might have 10 clusters and the cluster names might change to C0, C1, C2, …, C9. Since the `dma_tp` scenario does not refer to any of the cluster name, it can still be reused in this new design.

When setting up the environment for the new design, the only thing to change is the configuration file. By changing this information, we can easily reuse the environment, the PSS scenarios, and most of the tests; tests that refer to specific project attributes, such as the `dma_tp_cl0` action in Fig. 11 will need to be rewritten.

III. Results

Fig. 16 shows the performance results of each Samsung "Design Methodology (DM)" measured by the PSS test scenarios. BLK_CPU0 is a normal block where traditional Samsung DM is used. BLK_CPU1 and BLK_CPU2 are blocks where newly developed Samsung DM is applied on top of BLK_CPU0. New Samsung DM shows a 4~9% performance enhancements measured by the PSS test scenarios consisted of de-facto CPU benchmark tests and newly developed traffic profile generator. Traffic profile generator has big advantage on injecting traffic easily as much as we want. Thus, we can generate long-running stress test scenarios by mixing DMA tests and traffic profile tests. All of those test scenarios can be run in simulation and emulation environment without any modifications and show the same results.

Type	Test Scenario	BLK_CPU0	BLK_CPU1	BLK_CPU2
Memory Access	Write_Read	1	0.98	0.91
Benchmark	Dhrystone	1	0.97	0.91
	Gzip compress	1	0.97	0.93
	Lmbench	1	0.96	0.91
Stress	Traffic Profile + DMA	1	0.95	0.94

Figure 16. Performance result based on simulation time

In addition, the current PSS environment is highly reusable because all the configurations of the test chip platform are parameterized. In case that the numbers of CPU cluster are increased or decreased in the next project, new PSS environment can be created by just modifying the corresponding configuration tables and the same set of test scenarios can be validated in a few days. Therefore, the overall productivity is highly increased using PSS due to the reusability and the automation.

IV. Lessons Learned & Future Enhancements

This paper shows how PSS can be used to generate test scenarios for measuring performance of new design methodology as well as test scenarios for system validation in post-silicon. By modeling them using PSS, we can automate the test scenarios generation and reduce the code generation efforts significantly. This gives us more time to focus on defining additional test scenarios which have not been thought off before. Also, PSS provides the randomization of the attributes in generating new tests and supports the composition of new scenarios by leveraging existing actions so that we can compose interesting test scenarios quickly. It leads to improve the quality of post-silicon validatation.

Acknowledgment

The authors would like to acknowledge the contributions made by Cadence for developing the CPU benchmark tests and a traffic profile generator required for our project.

References

[1] Keshava, J., Hakim, N. and Prudvi C. (2010) Post-silicon Validation Challenges: How EDA and Academia Can Help", DAC'10, June 13-18, 2010

[2] Mitra, S., Seshia, S.A. and Nicolici, N. (2010) Post-Silicon Validation Opportunities, Challenges and Recent Advances. 2010 47th ACM/IEEE Design Automation Conference (DAC), 12-17.

[3] Accellera, Portable Test and Stimulus Standard, Version 1.0a, (February 2019).

[4] Jang, M., Kim, J., Chung, H., Huynh, P., Shai, F. (DVCon 2019) Coherency Verification & Deadlock Detection Using Perspec/Portable Stimulus
[5] ARM Cortex-A53 MPCore Processor, Technical Reference Manual, Chapter 12: Performance Monitor Unit