2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

An Enhanced Stimulus and Checking Mechanism on Cache Verification

Chenghuan Li
Mediatek.inc, Beijing, China
Chenghuan.Li@mediatek.com

Xiaohui Zhao
Mediatek.inc, Beijing, China
Xiaohui.Zhao@mediatek.com

Yunyang Song
Mediatek.inc, Beijing, China
Yunyang.Song@mediatek.com

***Abstract*- Cache verification has been regarded as one of the most challenging problems due to increasing design complexity. In this paper, we demonstrate a flexible UVM-based verification solution to cache. In order to fulfill cache's typical scenarios, an elaborately-designed layered sequence using the feedback mechanism is adopted. Scoreboard based on mesh streams is used for end-to-end data checking. Taking into order and correspondence into consideration, we demonstrate several customized checking policies.**

I. INTRODUCTION

With the prosperous growth of applications such as AI and ADAS in recent years, the application specific processor has become a hotspot. Those applications usually requires mass of data access, which raises a higher demand for memory system with high bandwidth and low latency. In order to cater these demands, a lot specific or tricky designs are introduced in current cache design. The cache verification has become a challenging task due to the increasing design complexity.

In this paper, we present a UVM-based verification solution to cache which supports multi-thread access, multi-word access, kill and pre-fetch mechanism. In order to demonstrate its effectiveness and flexibility, this solution is introduced in two parts: Stimulus generation and mesh stream scoreboard checker. Our goal is to make cache verification more manageable, scalable and unified.

II. DESIGN OVERVIEW

Our L1 Cache design (L1SYS in short) has three stage pipelines, support multi-thread double/triple word access. When one thread encountered cache miss, that thread will be scheduled to a 'shadow command buffer', fetch data in background, and switch to another thread automatically as shown in Figure 1. , in this way, the overall latency will be reduced. The introduced extra design and verification effort as the front-ground and background access may target to same line.

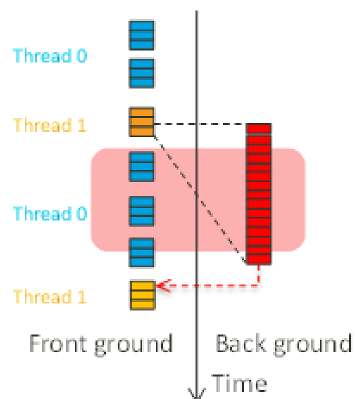

Figure 1. Demonstration of thread switch when cache miss

What's more, in order to gain better performance, some specific designs are introduced in the L1SYS, including but not limited to configurable pre-fetch mechanisms which takes cache spatial locality into consideration, more aggressive cache line replacement policy avoiding target cache line evicted by another thread, 'kill' mechanism which is aimed at reducing the invalid access to external memory when branch prediction failed. To achieve these goals, many buffers are introduced in design implementation as shown

in Figure 2. , which makes cache verification more challenging. For example, to examine whether the load/store is cache hit or miss, we need to check different combinations of cache and buffer under different situations.

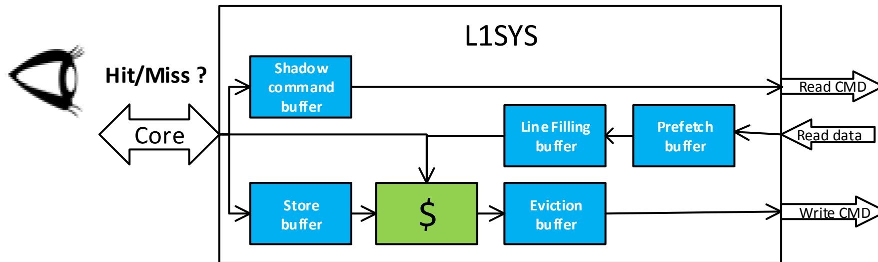

Figure 2. L1SYS internal buffers and data flow

The prime reason why cache exists is the locality of memory access, which include two different aspect:

- Temporal locality: recently referenced data is likely to be referenced again soon
- Spatial locality: it's more likely to reference data near recently referenced data

Let's revise a typical memory access process to describe the locality during memory access. As shown in Figure 3, assuming we have a memory space which can be addressed from '0a' to '7d', each address contains a data unit that core can access, and we have a cache whose cache line can hold 4 data units.

1. Core write 2b, which will be cache miss as all cache lines are invalid, the L1SYS will allocate data from 2a~2d to fill cache line. Additionally in our design, L1SYS will prefetch the data 3a~3d, store them into 'prefetch buffer' based on spatial locality
2. Core write 2b again, which will be cache hit, which is an example of temporal locality.
3. Core read 3c, which will hit 'prefetch buffer', data will be returned to core soon just like cache hit, meanwhile data in 'prefetch buffer' will be moved into cache on background
4. Core read 6c, which will be cache miss, L1SYS will allocate data 6a~6d, prefetch data 7a~7d. As data 6a~6d will occupy same cache line as 2a~2d, the dirty data 2a~2d will be evicted to external memory, those data will be firstly move into 'eviction buffer' to speed-up cache replace
5. Core write 2b before the eviction mentioned above is done, as data 2b is still inside 'eviction buffer', it will be traded as cache hit from core side, new data will be carried out to external memory directly along with other evicted data

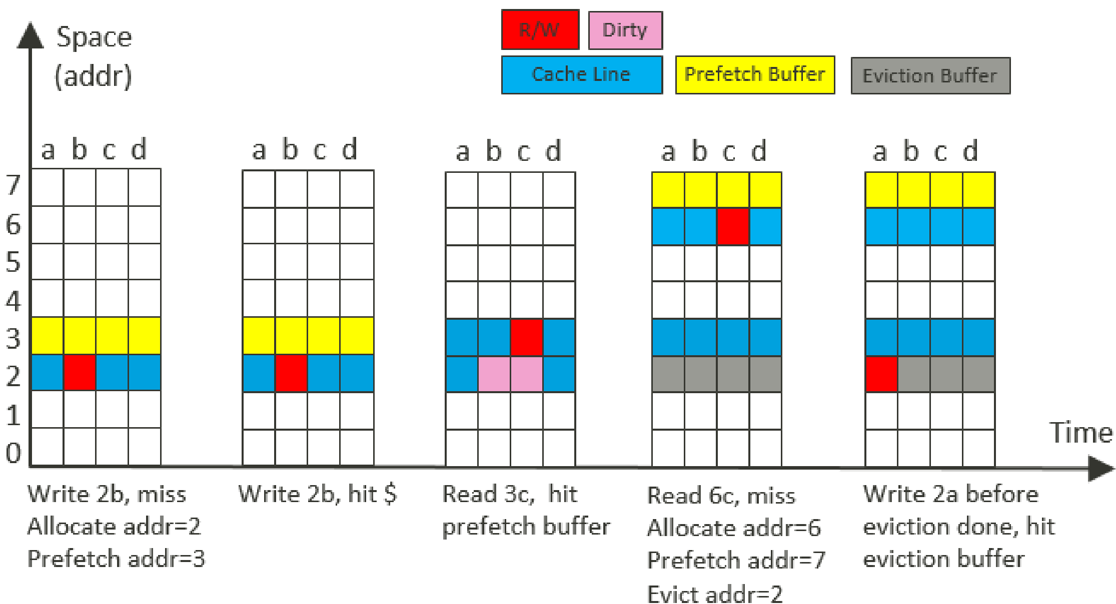

Figure 3. Example of typical L1SYS memory access

III. TEST BENCH OVERVIEW

Based on design features mentioned above, we propose a UVM-based verification architecture, as shown in Figure 4.

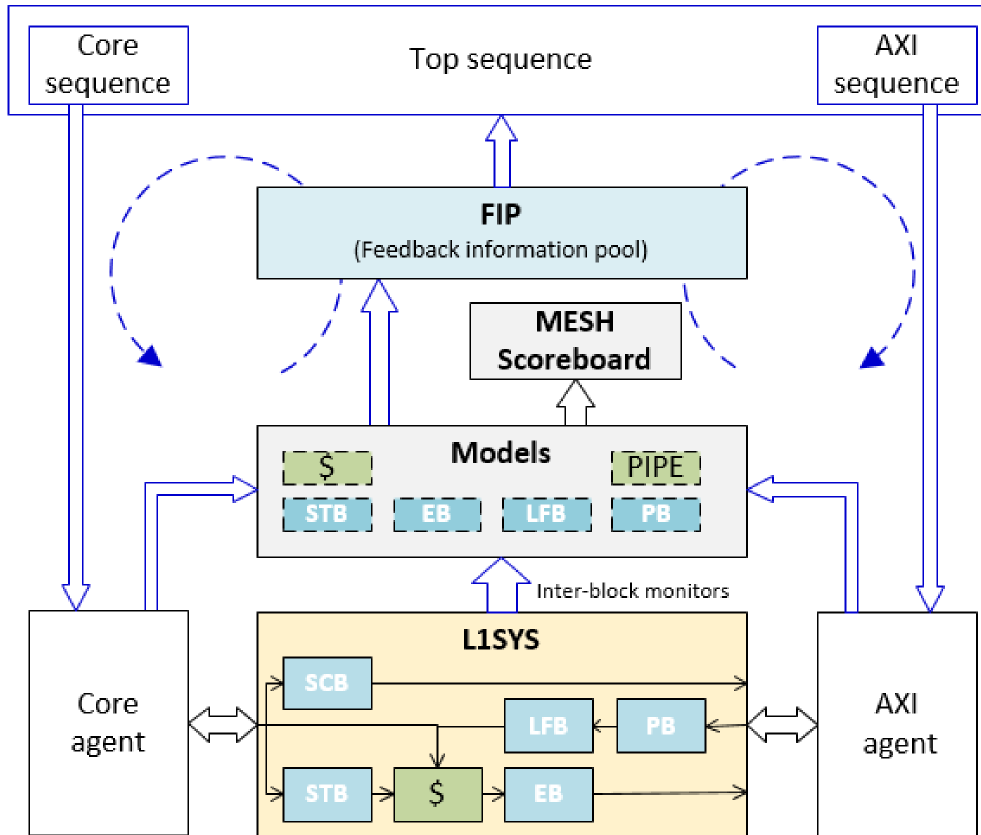

Figure 4. Demonstration of L1SYS verification architecture

The testbench is organized in gray-box style. For the purpose of accurate data and status check and predication, some key internal interfaces also need to be observed besides the external core and AXI interface. With the help of transaction level models including cache and internal buffers, transactions monitored from those internal/external interfaces are all checked using MESH scoreboard. The MESH scoreboard are combined by arbitrary number of data streams, each stream can have customized checking policy using UVM instance override, each stream can be turn on/off separately. In such way, a scalable scoreboard frame is provided for different check accuracy during different execution stage, especially when need to add or remove internal interfaces that being observed. The MESH scoreboard will be described in details in section V.

Due to complexity of L1SYS implementation, the stimulus need to provide cycle-level controllability on command types, timing, and their combination of different threads. To achieve this, the stimulus are organized in layered sequences. With the characteristics of both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items, the only simple random stimulus cannot effectively cover cache's typical behavior. FIP (feedback information pool) is introduced in our testbench, FIP which is filled with feedback information from monitors and models in testbench can provide transaction-accurate feedback to stimulus generation, which can dramatically speed up verification process. Layered sequence and FIP mechanism which will be described in details in section IV.

IV. STIMULUS GENERATION

A layered sequence with feedback mechanism is proposed in our solution for stimulus generation.

A. *Layered sequence*

Following verification requirement need be fulfilled during L1SYS verification.

1. In the normal operation of the L1SYS, it shall go through a completed process including 'initialize', 'configure' and 'data'.
2. There's multiple external interfaces that need hook up active agents, whose sequences need be executed concurrently once 'data' sequence is started
3. Due to our L1SYS' supporting multi-thread, when one thread encounters the condition of cache miss, it shall be sent to 'shadow command buffer' as shown in Figure 1. The thread in 'shadow command buffer' shall be handled in background (BG), while the other threads still execute in front ground (FG). When the BG operation of cache miss is finished, the thread switches from BG to FG. In this situation, the master sequences are required to provide the controllability for the arbitration among 'fresh' commands from FG, 'redo' commands from BG.
4. Cache related operations are divided into atomic operation called 'scenario', which should be combined freely to compose more complex stimulus.

In response to the above verification's needs, the sequences used in L1SYS verification are organized in a 4 layer structure, as shown in Figure 5. In the following section, we will discuss every layer separately.

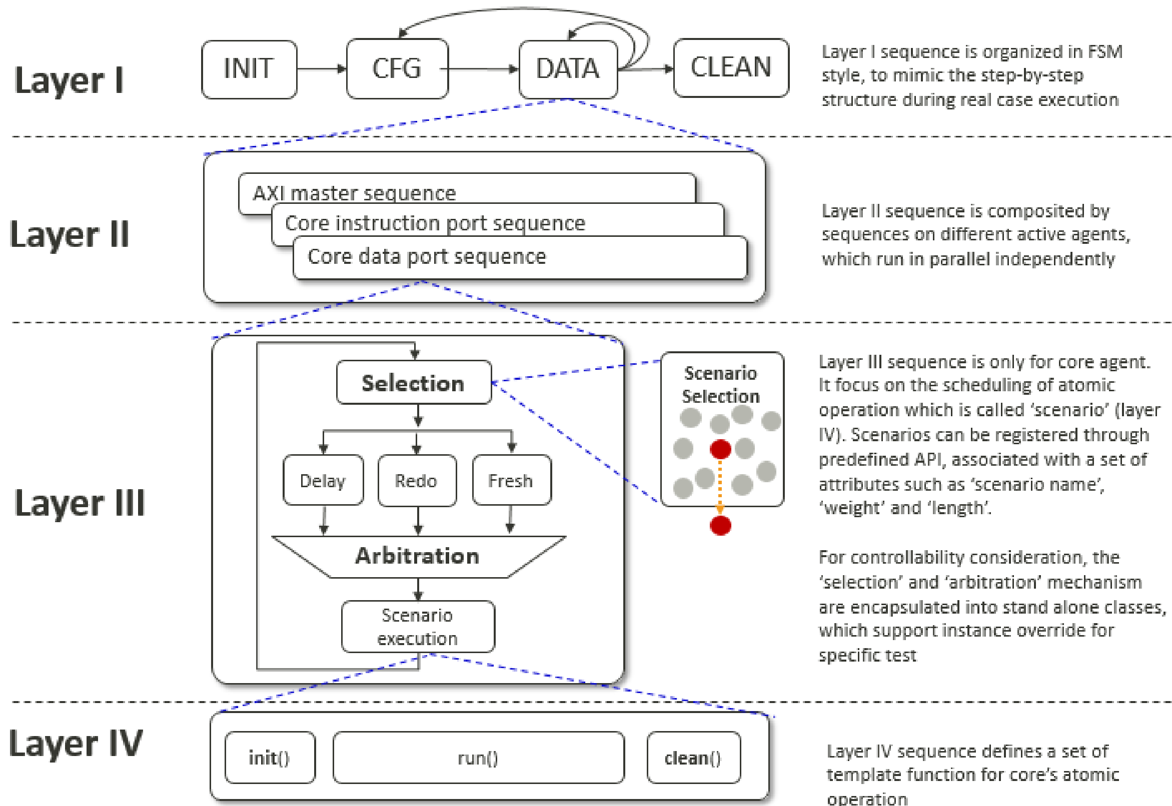

Figure 5. Overview of layered sequences

Layer I sequence is the top sequence which control the overall flow of simulation, it is executed as UVM main_phase's 'default_sequence'. To mimic the real case execution, the sequence is implemented in a FSM (Finite state machine) style. A general sequence class is extended from uvm_sequence_library class, which provides a FSM style user-defined selection mechanism.

Three concepts are introduced in that sequence, as shown in Figure 6.

- state: based on which the sequence to be executed is selected;
- arc: the sequence to be executed from state-n to state-m;
- weight: to control the statistical distribution multiple sequences which have same current state.

A couple of APIs are provided in the sequence class to build-up to FSM, also built-in functional coverage is provided in the FSM sequence on 'state' and 'arc' execution status.

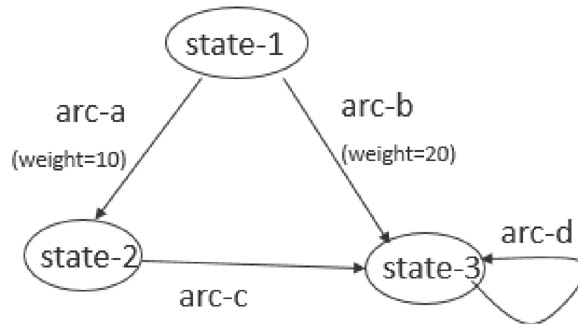

Figure 6. Concepts of FSM sequence

As shown in Figure 5., in L1SYS'verification, four states including 'INIT', 'CFG', 'DATA' and 'CLEAN' are defined. When state changes from 'INIT' to 'CFG', the design initialization sequence will be executed. When state changes from 'CFG' to 'DATA', the main data transfer sequence will be executed. State 'DATA' can jump back 'CFG', run circularly to 'DATA' state, or goes to 'CLEAN' state. When state changes from 'DATA' to 'CLEAN', some housekeeping jobs are be executed. Related pseudo code to build the FSM sequence is like bellow.

```
virtual function void build_fsm_sequence ();
   this.fsm_seq.register_state('{"INIT", "CFG", "DATA", "CLEAN"});
   this.fsm_seq.set_init_state("INIT");
   this.fsm_seq.register_arc("INIT",                              // current state
                             "CFG",                               // next state, after sequence is executed
                             my_init_sequence::type_id::get(),    // sequence to be registered
                             100                                  // weight control, optional
                             );
  // reigster other FSM 'arc'
endfunction
```

Comparing to the 'randsequence' syntax provided in SystemVerilog , the FSM sequence provide a more readable style to build sequence, also built-in functional coverage can explicitly show the execution status. As for UVM phase jump, it's always not recommended to use due to side-effect it will introduce.

Layer II sequence is composited by sequences on different active agents, which are run in parallel and independently. The scenario number that determines how many scenarios are executed on each master is controlled using run-time plusargs. This also allows pattern developer to turn off certain masters while avoid affecting each other, which is useful during the early stage of verification. Related pseudo code of this sequence is demonstrated as bellow.

```
task body () ;
   this.get_scn_number_from_plusargs() ;
   fork
      this.ex_dm_seq() ;     // Execute Data sequence
      this.ex_pm_seq() ;     // Execute Instruction sequence
      this.ex_axi_seq() ;    // Execute AXI sequence
   join_none
   wait_fork ;
endtask

task ex_dm_seq() ;           // Take DM sequence as an example ;
   repeat ( this.dm_scn_num ) `uvm_do_on ( dm_seq, dm_agent_info.sqr ) ;
endtask
```

Layer III sequence is only for core agent. It focus on the scheduling of atomic operation which is called 'scenario' (Layer IV sequence). Scenarios are registered through predefined API, associated with a set of attributes such as 'scenario name', 'weight' and 'length'. Using these information, together with UVM factory mechanism, scenario can be added and composed easily. Scheduling mechanism consists of item selection and item arbitration. When executed, item selection would selects all available item types, then item arbitration determines which type to send. Scenario selection is to decide the 'scenario_name' of all available scenarios according to its weight. Some useful APIs are provided in selection and arbitration, which enables pattern developers to customize the result depending on scenario. What's more, for controllability consideration, the 'selection' and 'arbitration' mechanism are encapsulated into standalone classes, which support instance override for specific test.

```
layer4_scenario     scn ;

task body () ;
   scn = this.select_scenario () ;   // Select Layer 4 scenario
   scn.init () ;                     // initialization for scenario context setup
   repeat (this.sequence_length) begin
      scn.run();                     // scenario main body,
                                     // one transaction per 'run' for better controllability of 'locality'
   end
   scn.clean() ;                     // housekeep after scenario
endtask
```

Layer IV sequence defines the 'scenario' mentioned above. Some scenario's execution may need certain context such as locality and state. In such situation, scenarios are extended from a common base class, which defines a set of template function for core's atomic operation, such as init(), run(), clean(). The function 'init()' is called once during this scenario to set execution context and 'clean()' is called once to clear up this context. The task 'run()' is called multiple times which is controlled by sequence length to complete the scenario. This part is what verification engineer pay most effort on after the environment is ready, a total of 60+ scenarios are created during our L1SYS verification.

```
task init();  // handle context for this scenario; endtask
task run() ;
   // allocate constraint from FIP;     // create transaction with constraint;
   `uvm_send(tr)
endtask
task clean(); // clean context ; endtask
```

The cooperation between LayerIII and LayerIV sequence is shown in Figure 7.

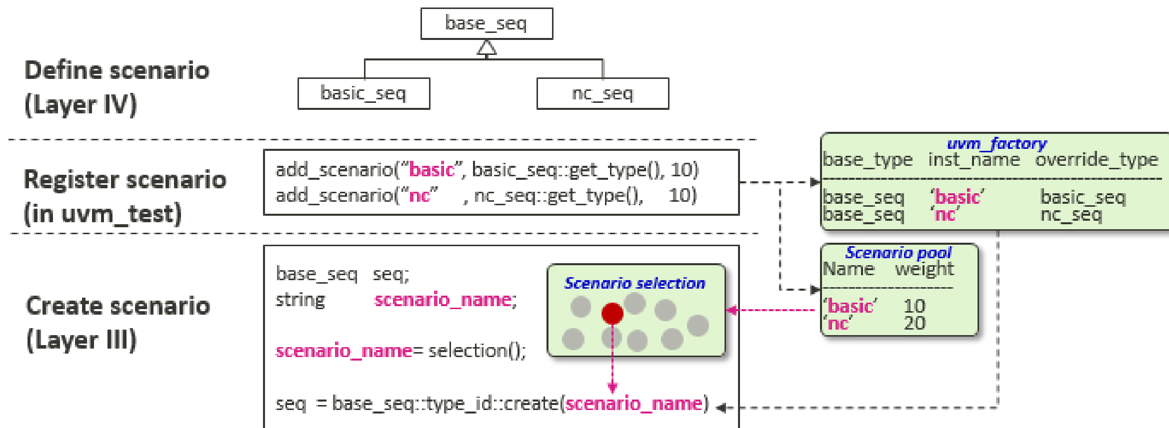

Figure 7. Cooperation between Layer III and Layer IV sequence

B. Feedback mechanism

In cache's verification, the stimulus should take cache's characteristic of both temporal and spatial locality into consideration. In order to gain better performance (cache hit rate, etc.), our L1SYS introduce a lot of buffers, which raises a higher demand for stimulus generation. Taking these buffers into account, the temporal and spatial locality means that stimulus should tend to send command whose address is same with or close to the cache lines in the buffers in a short time range.

What's more, when taking multi-thread access into consideration, we must focus more attention on the stimulus correlation among threads. For example, two threads try to fetch same cache line which is not in cache. The initial thread could get cache miss result and try to allocate this line from main memory. The second one may get different results (hit shadow command buffer, hit line filling buffer or hit cache) depending on the status of former allocation.

Our solution is to introduce feedback mechanism in our stimulus generation. A FIP (Feedback Information Pool) component is created to reflect the behavior of design, those information is collected from monitors and models in test bench. To achieve this, some extra APIs for querying usage are introduced in the monitors and models. The FIP is consisted of L1SYS's status information, including pipeline, cache/TCM SRAM and all buffers status in L1SYS. What should be emphasized here is that all status information is collected from transaction level models in testbench instead of RTL signals. In another words, the information used for feedback has been proven to be correct in scoreboard and checkers.

When a scenario start to execute, it will request an high ROI (Region-Of-Interst) address range by raising requirement that encapsulated in a 'ROI_setting' object, which include the address size, expected buffer/cache correlation, etc.. Once received the request, FIP starts to calculate all possible address ranges which fulfil the requirements, an 'addr_range' object will be returned to scenario to describe all satisfied address ranges. Scenario arbitrarily choose one address range from 'addr_range' object, then do further randomization on the transaction it will send.

The overall feedback mechanism is shown in Figure 8.

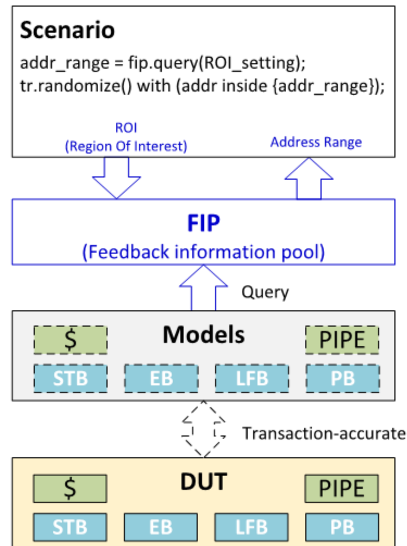

Figure 8. Demonstration of feedback mechanism

V. MESH SCOREBOARD

To accurately predict cache hit/miss result, we need to monitor couples of key node of internal data path, and do cross check on amount of the combination of those key nodes. Those nodes have different data transaction types. What's more, those nodes might be added or removed when design changes, or during different verification phase. In this situation, a flexible and extendable mechanism is needed to cater this demand.

The checking mechanism we used in L1SYS's verification is shown in Figure 9. The core component is named as MESH scoreboard, each data path from one interface node to another is named as a 'stream', all streams together compose the MESH scoreboard. VIP agents are bind to design boundary or inside design whose functionality we must pay attention to. Subscribers are connected to VIP agents by UVM TLM ports. Subscriber do pre-check on the transaction it received, and translate the transaction into MESH scoreboard item, which defines what we want to check. Those function might be delegated to separate 'transaction handler' objects if multiple scoreboard streams are involved for single subscriber.

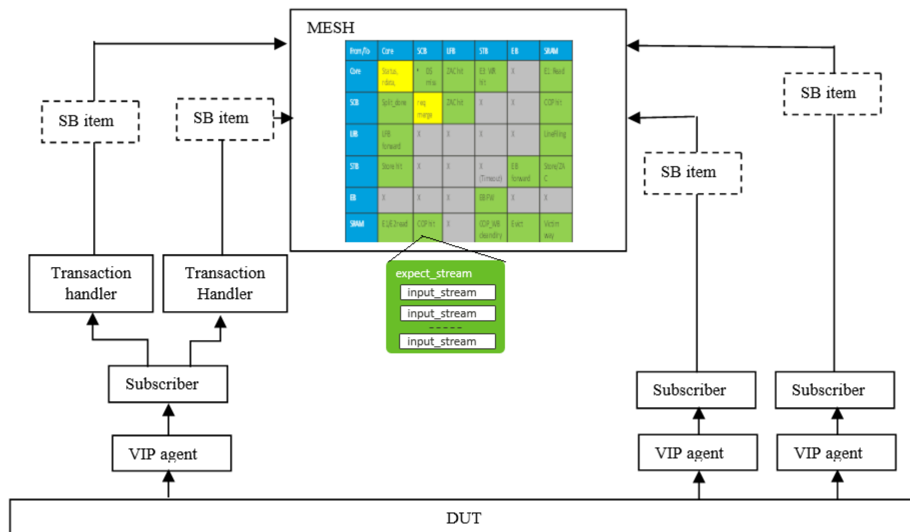

Figure 9. Demonstration of mesh stream used in Cache verification

All agents created in our L1SYS's verification share the same architecture which is consisted of a monitor, a driver, a sequencer and a configuration object. Design under test (DUT) is connected with monitor and driver via virtual interface. The monitor takes the responsibility of translating the signals of DUT into transaction and broadcasting via analysis port. Conversely, the driver translates the transaction from sequencer into signals that is used as the DUT's stimulus. What's more, a configuration object is placed in the agent and the other components share the same handle of it. Controllability of the agent is provided in the transaction and configuration.

For the purpose of accurate checking, some fields indicating the status of transaction are pre-defined in the transaction, including address, data and response status. The monitors are designed to broadcast the transaction handle via analysis port at the very first cycle of the transaction. Also, they take the responsibility of modifying the status information along the life cycle of transaction. In this way, the subscriber can do corresponding check by waiting targeted transaction status as shown in Figure 10. For example, if we want to check the command information in some ports, what we need to do is to wait the transaction's address status to become 'DONE' and start to execute the related checkers. Similarly, we can wait data and response status to become 'START' or 'DONE' to check data and response information respectively, which is easy to extend to other situations when necessary.

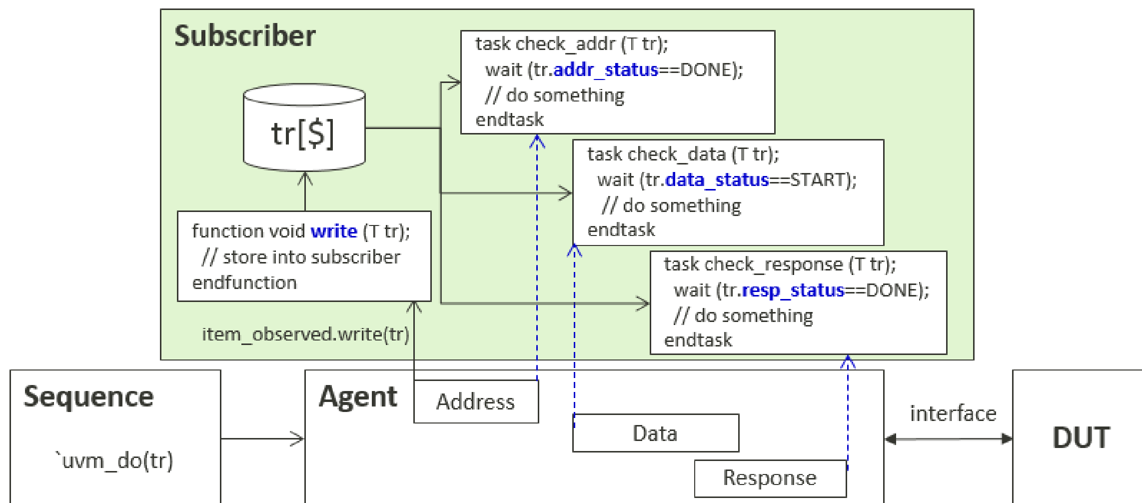

Figure 10. Interaction of agent and subscriber

The implementation of MESH scoreboard is demonstrated in Figure 11. The scoreboard is characteristic of flexible checking policy. An input stream is a stream from input side, an expected stream is stream that gathered on the output side. Many expect streams are defined in scoreboard based on the items that need be checked, each can be enabled or disabled separately. Every expect stream has its own customized checking policy which is achieved via UVM factory's instance override. Totally 16 checking polices are defined in our implementation, including but not limited to 'in-order', 'out-of-order', 'with-losses', 'any-in-order', 'either-in-order', 'MISO(Multi Input Single Output)-in-order', 'with-redundancy' etc.

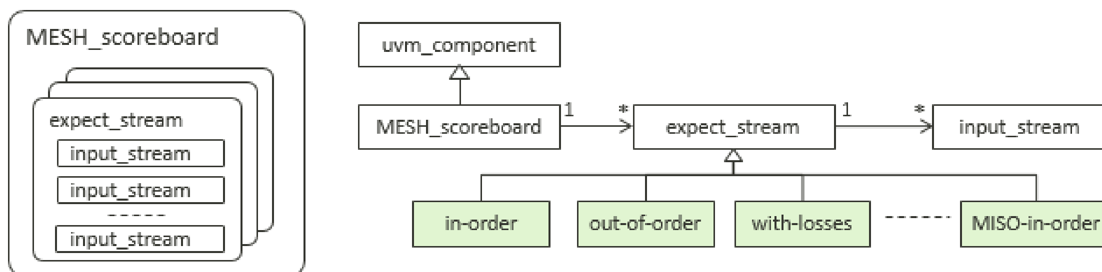

Figure 11. Scoreboard used in L1SYS' verification

APIs defined in our scoreboard is presented in Table I.

TABLE I APIs defined in MESH scoreboard

API	Description
define_exp_stream()	Define expect/input streams and the checking policy used for checkout
override_policy()	Add customized checking policy
checkin()	Check in mesh scoreboard item
checkout()	Check out mesh scoreboard item
delete()	Delete some items from mesh scoreboard
disable_stream()	Turn off check of specific stream
enable_stream()	Turn on check of specific stream

In order to illustrate how the scoreboards works, an example is presented in the following part of this section. Due to L1SYS's supporting multiple threads, if more than one threads try to fetch the same cache line when cache is miss, only one external memory access shall be issued to reduce cache miss's penalty caused by unnecessary operation. Aimed at checking this behavior, the checking policy of 'MISO-in-order' is created, where MI(Multi-input) means requests from multiple thread on core side, while SO (Single Output) stands for single memory access on AXI side. Once created, this policy is appended to scoreboard.

```
class asip_sb_expect_miso_in_order extends asip_sb_expect_policy ;
   virtual function asip_base::events_e check_out (input xxx, ouput xxx) ;
      // implementation of MISO-in-order
   endfunction
endclass

this.mesh_sb.override_policy ("MISO-in-order", asip_sb_expect_miso_in_order::type_id::get() ) ;
```

Using this policy, when checked out, it is judged to be correct as long as this item is found in at least one stream. What's more, items same as the checked-out one in all checked-in streams are all removed as shown in Figure 12.

MISO-in-order

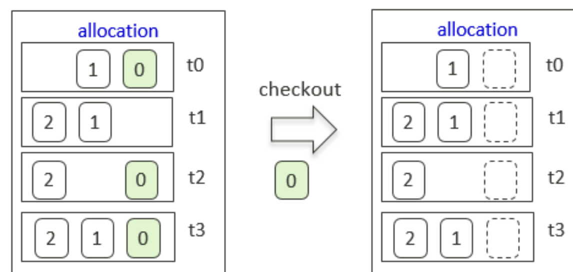

Figure 12. Demonstration of MISO-in-order

In order to check this behavior, first of all, an expect stream named 'allocation' with four input streams named 't0', 't1', 't2' and 't3' is defined. And 'MISO-in-order' is chosen as this stream's checking policy.

```
this.mesh_sb.define_exp_stream ( "MISO-in-order"                , // checking policy
                                 "allocation"                   , // expect stream name
                                 '{"t0", "t1", "t2", "t3"}        // input stream name
                               ) ;
```

If one of threads encounter cache miss, the transaction detected by monitor shall be translated in subscriber into scoreboard item. Then, the item shall be inserted into the 'allocation' expect stream. The selection of the input stream is depended on the thread information of the transaction.

```
this.mesh_sb.checkin ( item                    ,  // checking item
                       "allocation"            ,  // expect stream the item shall be inserted into
                       $sformatf("t%0d", tr.tid)  // input stream the item shall be inserted into
                     ) ;
```

Once detected, the external memory access shall also be translated into scoreboard item. The checkout method is called according to its thread.

```
this.mesh_sb.checkout ( item,                      // checking item
                        "allocation"               // expect stream the item shall be checked out
                        $sformatf("t%0d", tr.tid) // input stream the item shall be checked out
                      ) ;
```

Flexibility is characteristic of the MESH scoreboard checking mechanism. First of all, the verification precision can be controlled flexibly by binding VIP agents. The verification can be executed loosely by only covering the design boundary or tightly by binding VIP agent at the key module inside the design, depending on project schedule and DV resources. Secondly, the items defined in the scoreboard item can be customized per stream. Thirdly and most importantly, as a lot of tricky designs are adopted in cache to gain high performance, the simple one-by-one correspondence checking mechanism is not able to meet the demand, using the facility provided by UVM factory mechanism, the checking policy can be customized and overridden flexibly stream by stream. What's more, the verification task can be easily partitioned based on streams, which is of great benefit for resource management.

VI. COVERAGE MODEL

Coverage model is consisted of code and functional coverage. Functional coverage has two parts: one for coverage defined in test bench (ex. Monitor/models/subscriber etc.) on transaction level, the other one for coverage derived from RTL signals. The second part of functional coverage is automatically generated as shown in Figure 13.

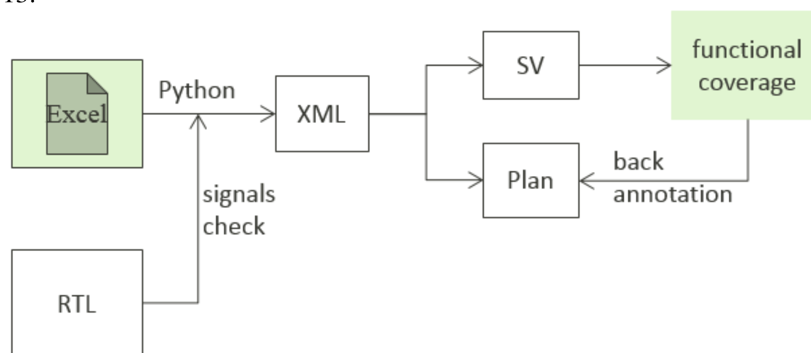

Figure 13. The automation of functional coverage

The excel information is provided by designers based on RTL signals. The Python script is used to translate information in excel to XML format. What's more, the script also takes the responsibility of checking the correctness of the information in excel by comparing with signals in RTL. Coverage groups and verification plan are generated based on XML. The plan is used to manage the verification schedule, which is back annotated based on functional coverage.

VII. SUMMERY

In this paper, we present an enhanced UVM-based verification solution for cache type design. We elaborately designed a layered sequence to fulfill cache's temporal and spatial locality. MESH scoreboard is used as checking mechanism due to its flexibility and extendibility. Also, trading the test bench as an organic whole, we adopts feedback mechanism from test bench's passive part for stimulus generation to accelerate the coverage convergence.

The coverage trend curve based on our project is shown in Figure 14.. Totally 62 scenarios are created in 6 weeks to reach coverage closure, including both code coverage and a total of 26949 functional coverage bins. We think that the mechanism we proposed in this paper is a practical solution to complex cache design verification.

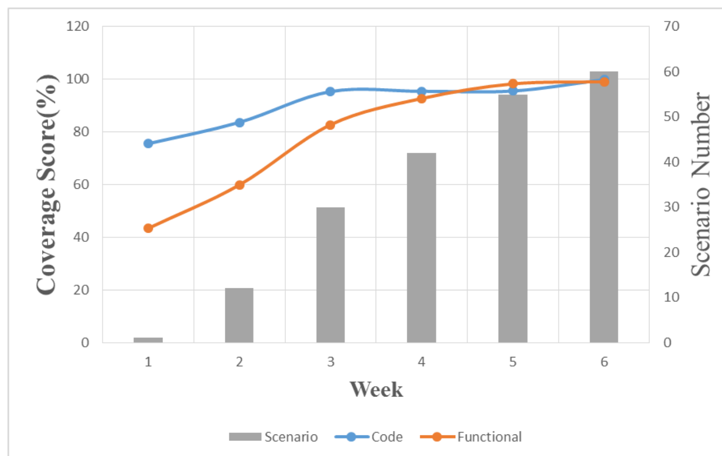

Figure 14. Coverage trend curve