2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Building Confidence in System Level CPU Cache Coherency Verification for Complex SoCs through a Configurable Flexible and Portable Test Bench

S Shrinidhi Rao, Ruchi Misra, Alok Kumar, Garima Srivastava,
YoungSik Kim, Seonil Brian Choi

SAMSUNG

accellera SYSTEMS INITIATIVE

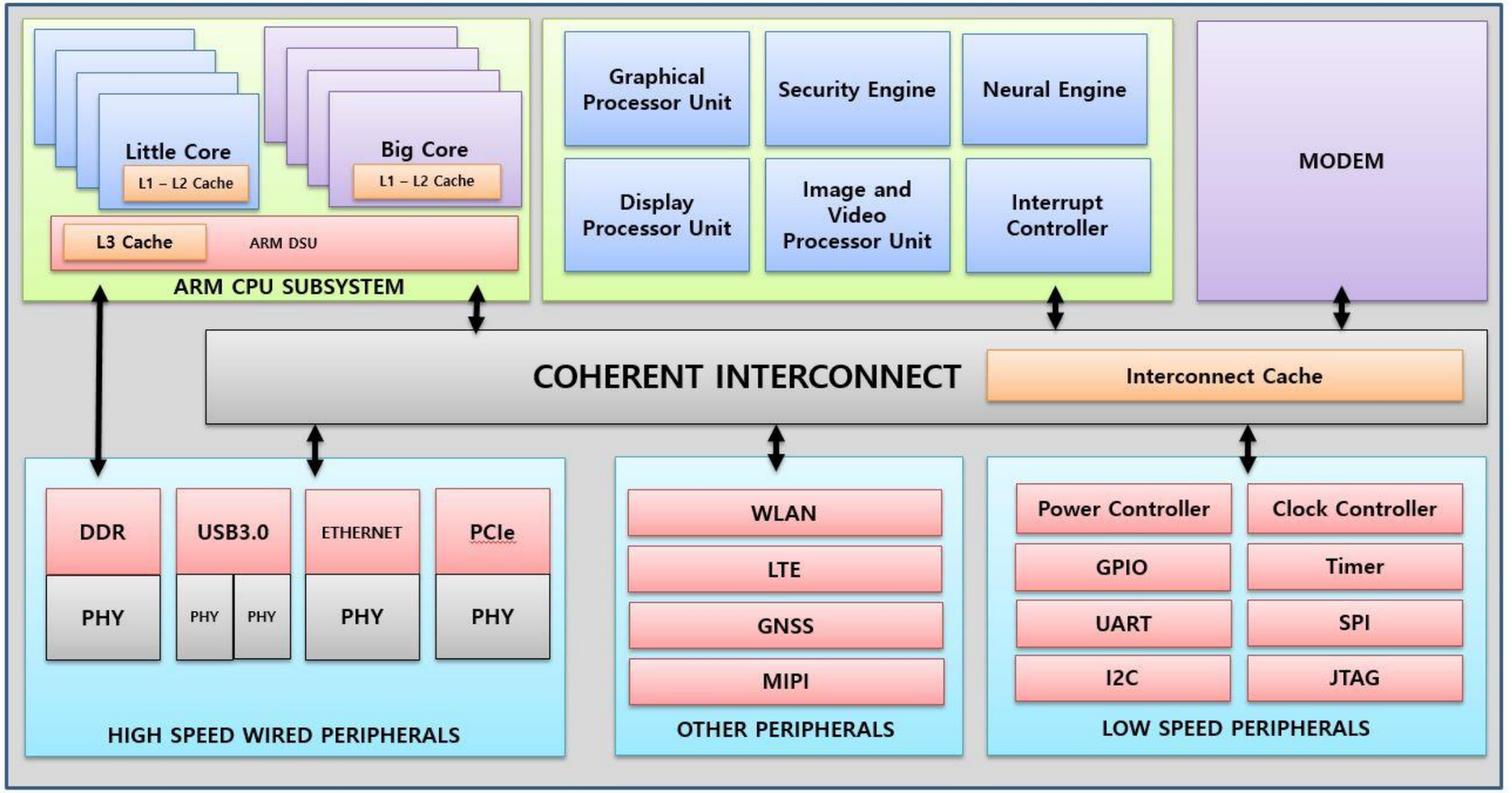
Agenda

- Motivation
- A Typical Multi-Core System Architecture
- Scope of PSS in CPU Verification
- PSS : An Accelera Standard
- Methodology
- TestBench Modification
- Solution Generation
- Tool View
- Results
- Conclusion
- Future Scope

Motivation

- Coherence protocols trigger complex interactions between heterogeneous mixes of caches and other system masters
- Our regular SoC scenarios are less aggressive on cache functionalities as compared to IP level testbench.
- Traditional approaches can sometimes miss bugs that arise only in a complex system level traffic.
- Writing large numbers of effective tests is difficult, time consuming, and error prone.

A Typical Multi-Core System Architecture



Scope of PSS in CPU Verification

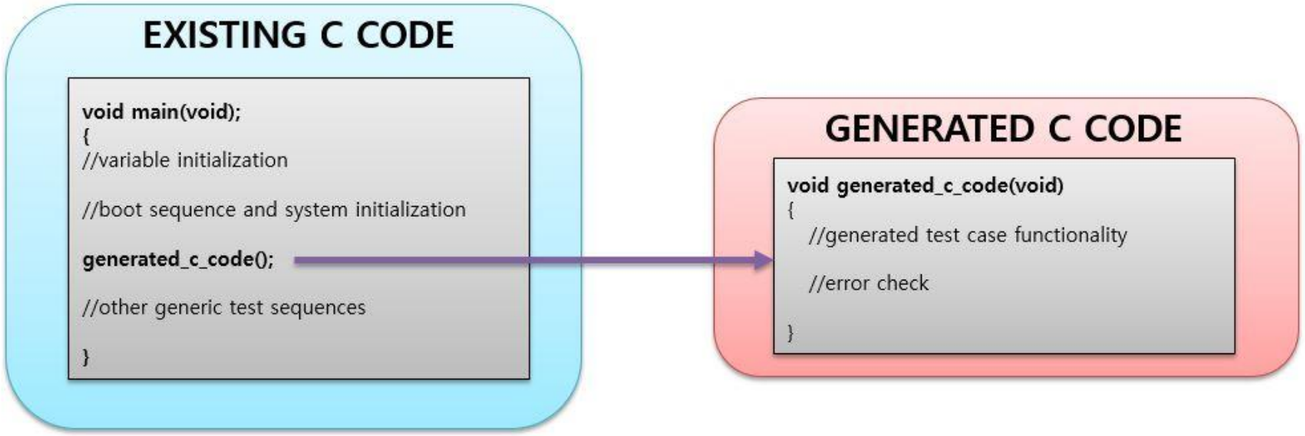
- Portability of reusable test cases has long been a goal for semiconductor verification and validation teams. No one wants to ‘reinvent the wheel’ by having to rewrite similar tests again and again.
- Verification of the CPU design in our multiple complex mobile/automotive or wearable SoCs is mainly done through C tests.
- This configurable testbench takes the specific configurations of the SoC as user-defined inputs, generates generic C code for some standard operations, and combines them to generate final C scenarios that run on embedded CPUs and exercises the system through diverse relevant solutions.
- The code generated for the simulation platform can then be re-used by the other 3 dependent environments, allowing reuse across various verification platforms.

Portable Test And Stimulus: An Accelera Standard

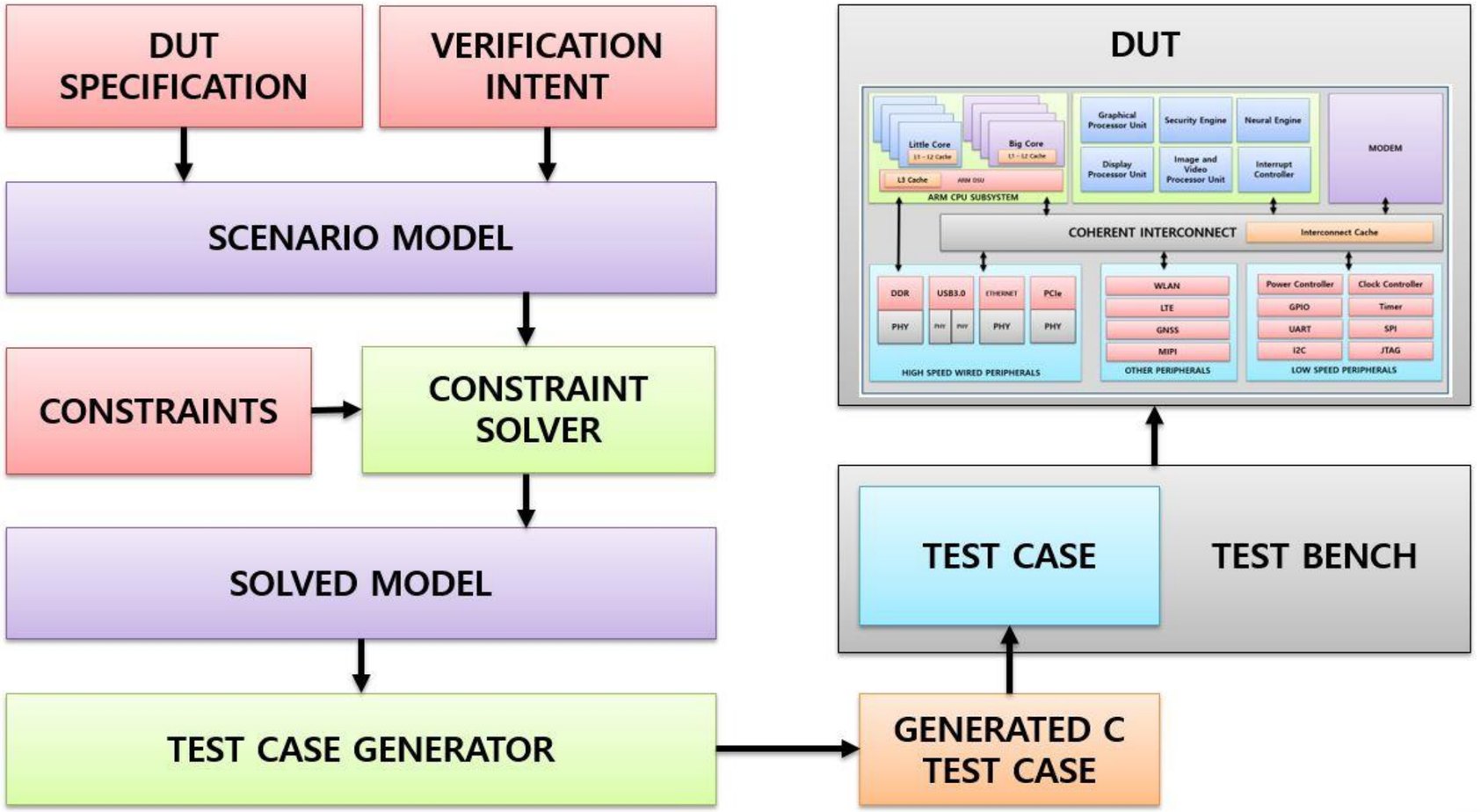
- PSS has been tried and tested in the industry
- Well established Vertical and Horizontal Reuse strategies
- Proven to enhance performance and verification efficiency

Methodology

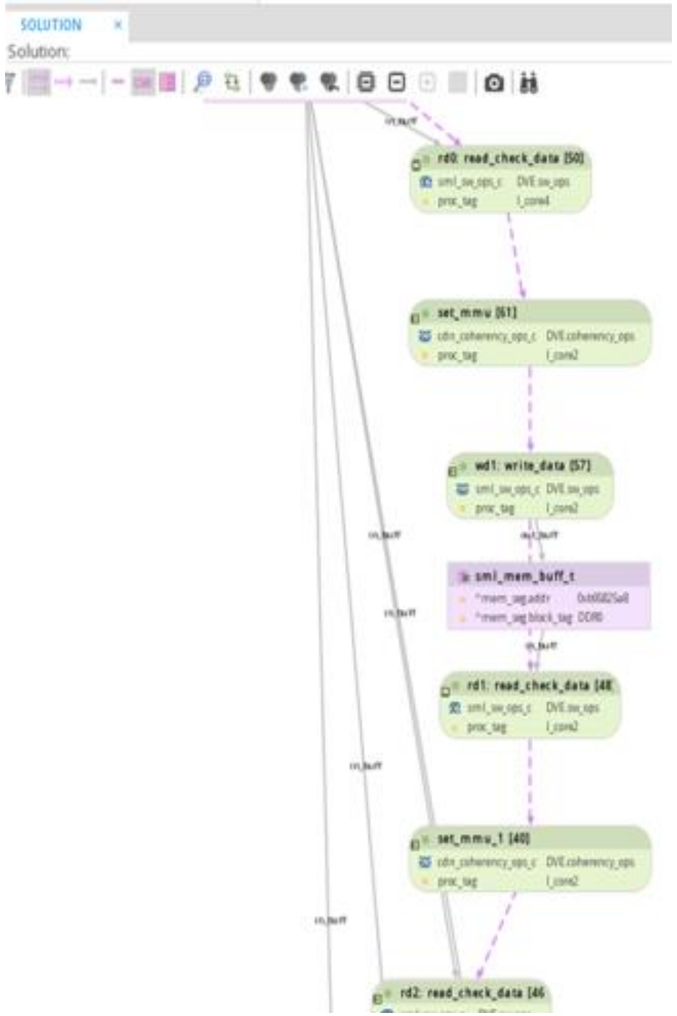
- C – based existing test bench
- Input to Solver
  - Design specification
  - Test and Design constraints
  - Pre-built action/function modules
  - Verification intent
- Output of Solver
  - Solution for verification intent
  - C code based on solutions generated
  - Coverage report
- Plug and Play

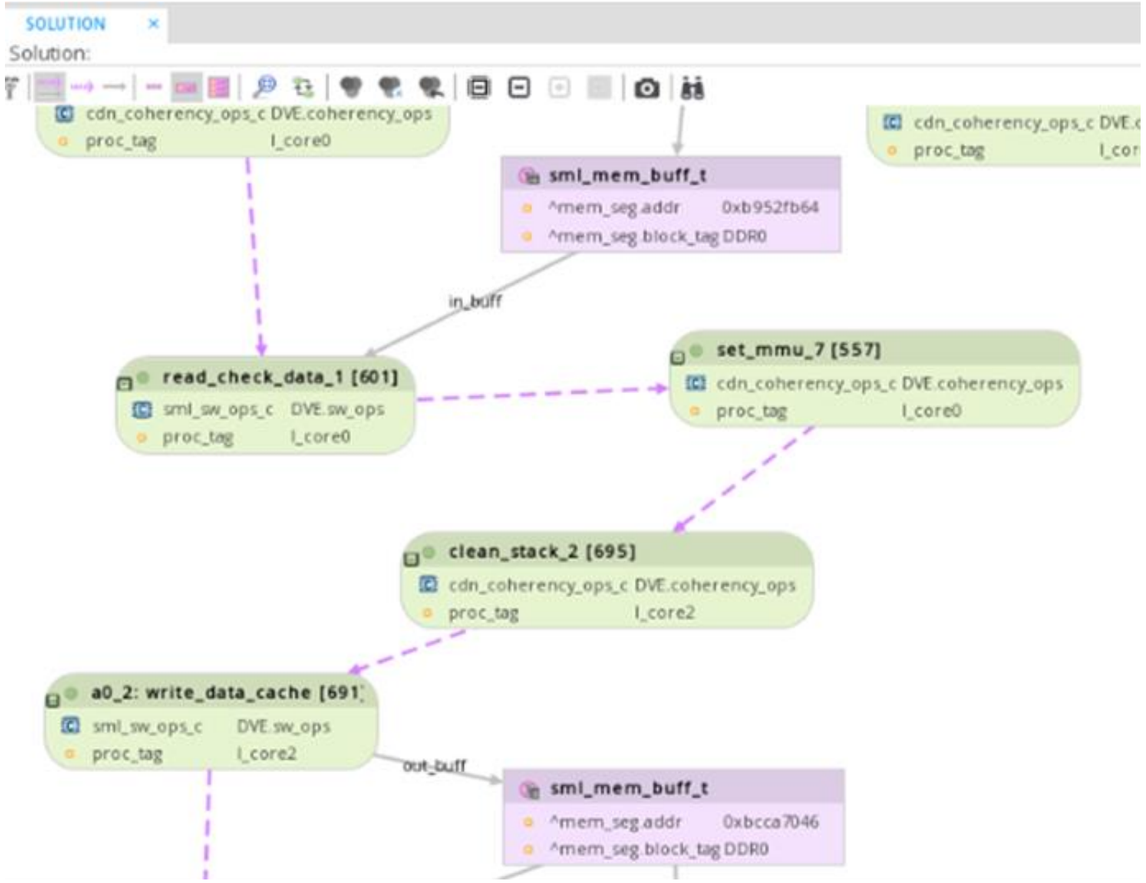


Test Bench Modification

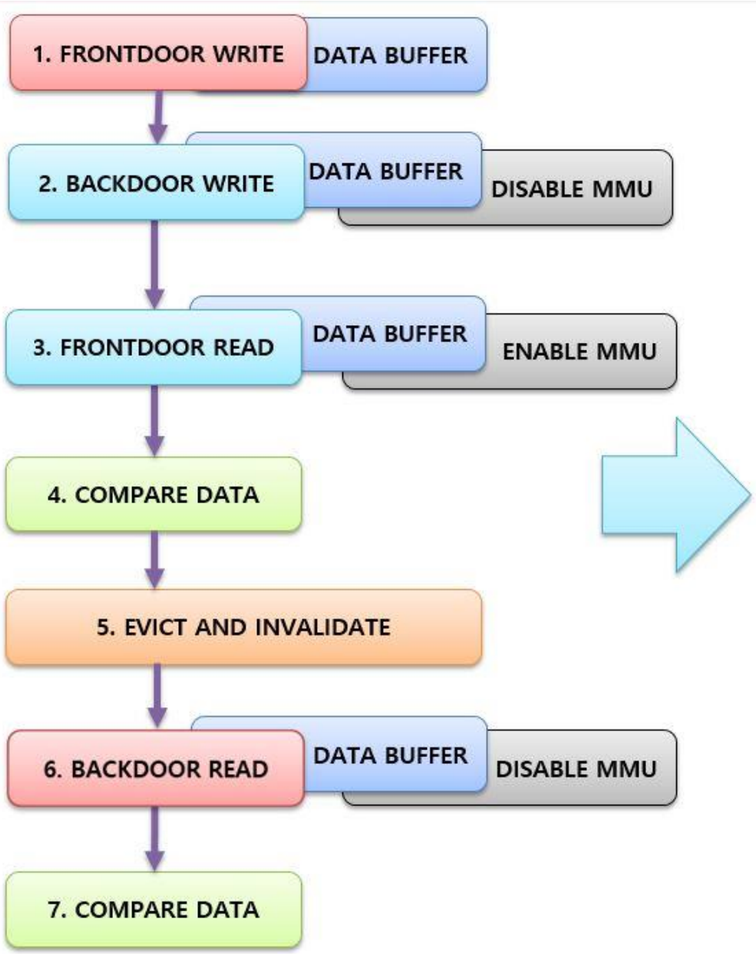


Solution Generation



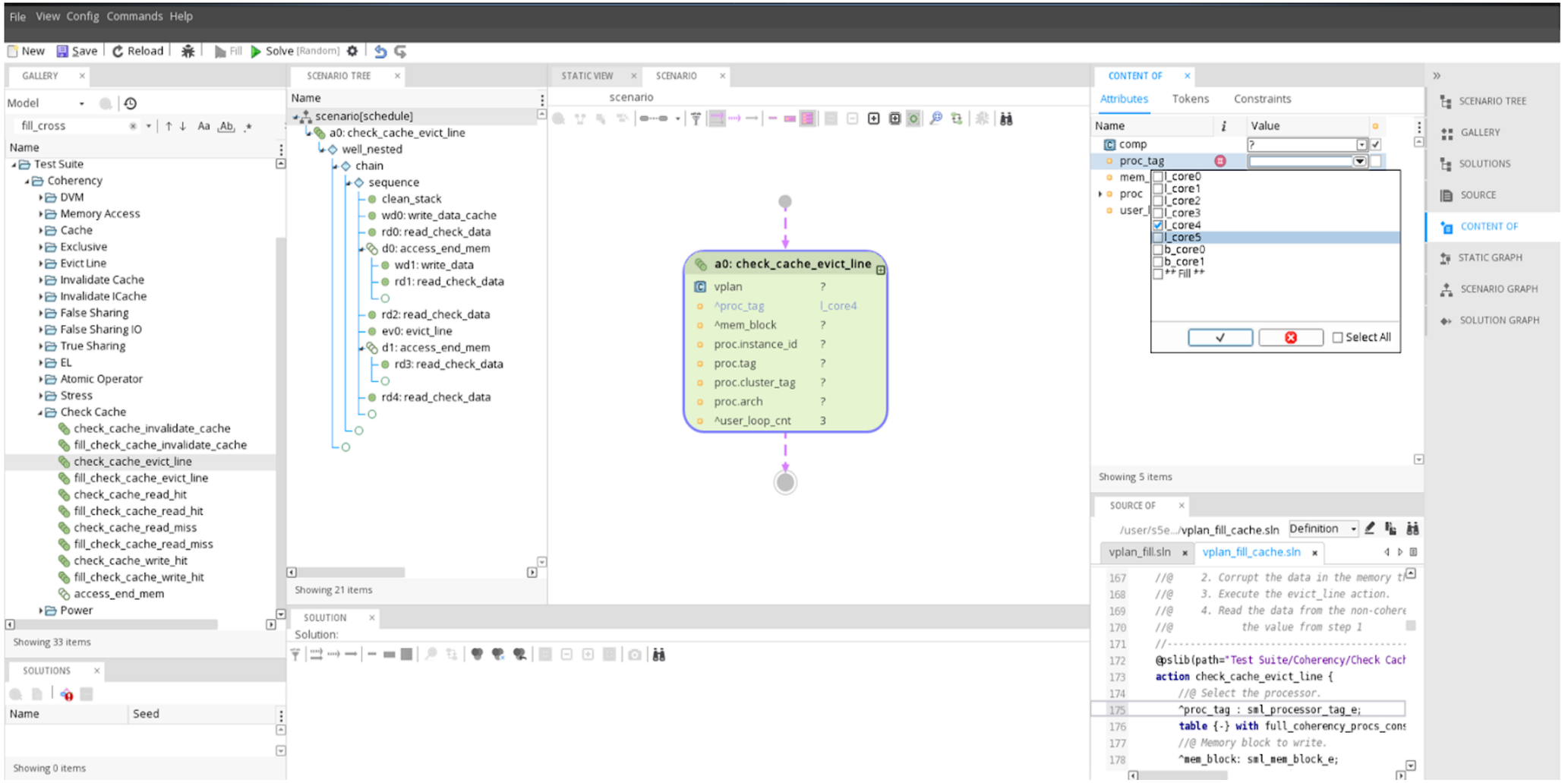


Solution to Test Case




```c
void read_hit_miss_check(void)
{
  unsigned int write_data_frontdoor, write_data_backdoor, read_data;
  unsigned int * write_address = (unsigned int *) 0x1000;
  unsigned int * read_address = (unsigned int *) 0x1000;

  write_data_frontdoor = 0xaaaa
  write_data_backdoor  = 0x5555;


  *write_address++ = write_data_frontdoor;
  *write_address++ = write_data_frontdoor;
  *write_address++ = write_data_frontdoor;
  *write_address++ = write_data_frontdoor;

  write_address = read_address;

  BACKDOOR_WRITE(write_address, 4, write_data_backdoor);

  for (i=0;i<4 ;i++){
      read_data = *(read_address++);
      if (read_data != write_data_frontdoor)
          printf("ERROR : READ MISS : Data read is fetched from Memory");
      else
          printf("INFO : READ HIT : Data read is fetched from Cache");
  }

  invalidate_and_evict_cache_();

  for (i=0;i<4 ;i++){
      BACKDOOR_READ(read_address, read_data);
      if (read_data != write_data_frontdoor)
          printf("ERROR : Cache Eviction failed");
      else
          printf("INFO : Cache Eviction Success");
  }
}
```

Tool View

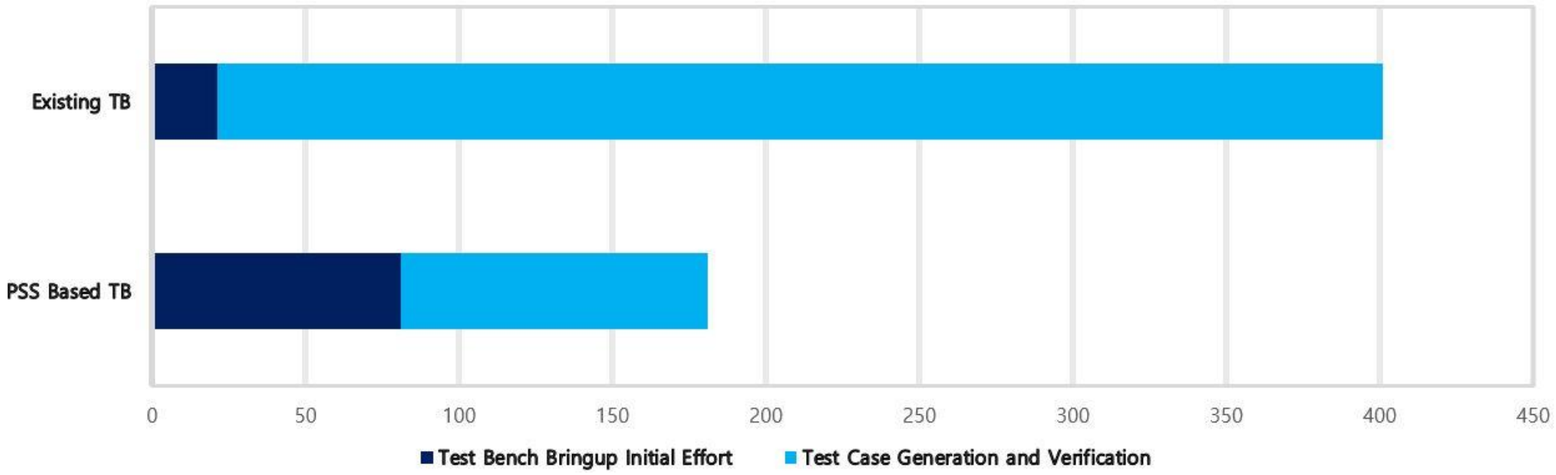

File View Config Commands Help
New Save Reload Fill Solve [Random]
GALLERY
Model
fill_cross
Name
Test Suite
Coherency
DVM
Memory Access
Cache
Exclusive
Evict Line
Invalidate Cache
Invalidate ICache
False Sharing
False Sharing IO
True Sharing
EL
Atomic Operator
Stress
Check Cache
check_cache_invalidate_cache
fill_check_cache_invalidate_cache
check_cache_evict_line
fill_check_cache_evict_line
check_cache_read_hit
fill_check_cache_read_hit
check_cache_read_miss
fill_check_cache_read_miss
check_cache_write_hit
fill_check_cache_write_hit
access_end_mem
Power
Showing 33 items
SOLUTIONS
Name
Seed
Showing 0 items
SCENARIO TREE
Name
scenario[schedule]
a0: check_cache_evict_line
well_nested
chain
sequence
clean_stack
wd0: write_data_cache
rd0: read_check_data
d0: access_end_mem
wd1: write_data
rd1: read_check_data
rd2: read_check_data
ev0: evict_line
d1: access_end_mem
rd3: read_check_data
rd4: read_check_data
Showing 21 items
SOLUTION
Solution:
STATIC VIEW
SCENARIO
scenario
a0: check_cache_evict_line
vplan ?
^proc_tag l_core4
^mem_block ?
proc.instance_id ?
proc.tag ?
proc.cluster_tag ?
proc.arch ?
^user_loop_cnt 3
CONTENT OF
Attributes
Tokens
Constraints
Name
Value
comp
proc_tag
mem_
proc
user_
l_core0
l_core1
l_core2
l_core3
l_core4
l_core5
b_core0
b_core1
** Fill **
Select All
Showing 5 items
SOURCE OF
/user/s5e.../vplan_fill_cache.sln
Definition
vplan_fill.sln
vplan_fill_cache.sln
167 //@ 2. Corrupt the data in the memory th
168 //@ 3. Execute the evict_line action.
169 //@ 4. Read the data from the non-cohere
170 //@ the value from step 1
171 //----------
172 @pslib(path="Test Suite/Coherency/Check Cach
173 action check_cache_evict_line {
174 //@ Select the processor.
175 ^proc_tag : sml_processor_tag_e;
176 table {-} with full_coherency_procs_cons
177 //@ Memory block to write.
178 ^mem_block: sml_mem_block_e;
SCENARIO TREE
GALLERY
SOLUTIONS
SOURCE
CONTENT OF
STATIC GRAPH
SCENARIO GRAPH
SOLUTION GRAPH

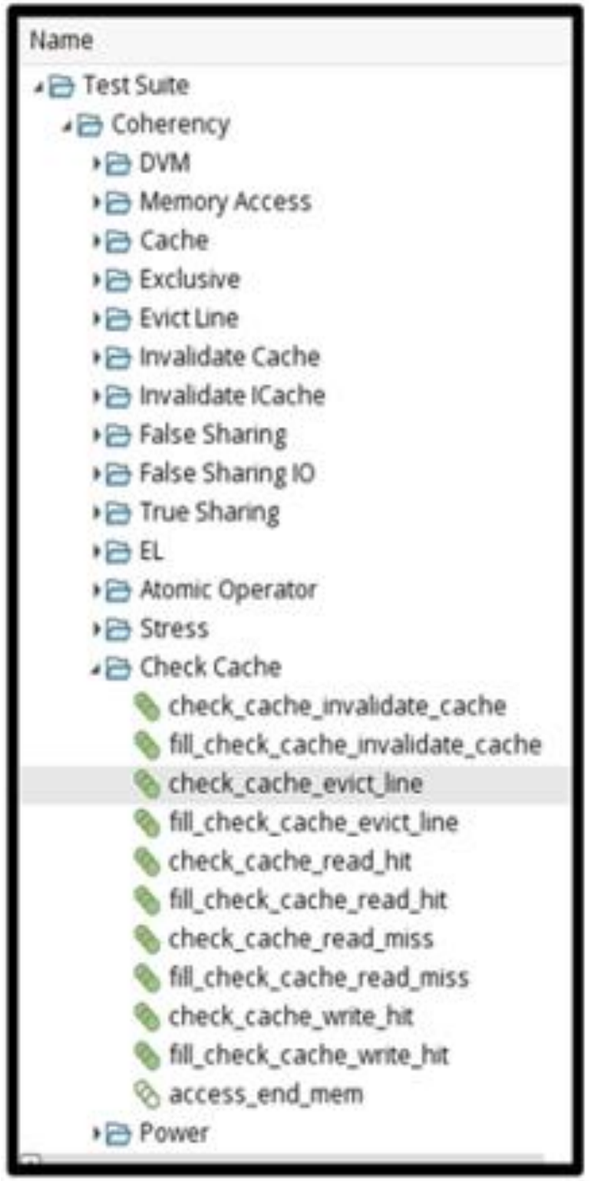
Some Example Scenarios

- Fill Cache Boundaries: A stand-alone Allocate to Cache task allocates a memory block to cache.
- Cache State Transitions: Using the different variety of opcodes supported, this complex test covers various cache state transitions with the help of an in-built checker.
- Data Sharing Scenarios: True Sharing and False sharing of data between cores in a coherent setup can significantly degrade cache performance, in systems when smallest data size accessed is smaller than cache line size

Results

- Case study run on Exynos Mobile SoC environment
- Initial setup needed multiple iterations to fix compatibility issues
  - 2 months to clean the setup and 3 months to run the complete set of tests
- Target Cache verification vectors
  - Verify Cache information such as size, allocation, overflow
  - Cache State Transitions : evict, invalidate, cache hit/miss
  - Cache performance : True Sharing and False Sharing





Conclusion

- Though initial effort was high, effort required for future projects is minimal
- Generate coverage reports at each step of test case generation
  - Help identify verification loopholes
  - Better coverage
- Ease of portability across multiple projects
  - Portable, re-usable and scenarios can be easily reproduced
- Confidence in verification due to constrained random scenario generation
  - Flexibility to modify test at each step
- Ease of portability across projects
  - Number of test cases that can be generated is high compared to manual development

Future Scope

- Recommended to be implemented in subsequent projects
- Opportunity to stress cache operations at SoC context
- Extend suite to Last Level Caches and I/O Coherency
- Enhance libraries to include newer ARM architectures
- Target power and clock control for power scenarios
- Combine power and Cache operations to stress design

Acknowledgement

- *The authors would like to thank Samsung Semiconductors India Research for enabling the work mentioned in this paper. We would also like to thank DVCon Europe team for giving us the opportunity to participate in the conference and present our work.*