# Simulation Performance improvement with Dynamic memory load & C model export

Mangesh Kondalkar (mkondalk@qti.qualcomm.com)
Varsha Antony (vantony@qti.qualcomm.com)
Qualcomm India Pvt. Ltd. Bangalore, Karnataka

***Abstract*-IPs such as compression engine, image processing accelerators and cache controller-based data management hardware, processes data or image that is stored in external system memory. Such IPs may support multiple hardware processing threads that can access images stored in the memory concurrently. Testbench for the verification of such IPs often uses backdoor preloading of memory with random data. Simulation performance is significantly affected by the number of images and the size of each image preloaded in memory. If a test is only exercising access to a part of the image which is dynamically configured, preloading entire image is an ineffective approach. Virtual memory size and CPU time needed for the execution of the test may be very high with large static memory preload. This could result in incomplete tests in regression if a machine with sufficient memory is not available. Higher virtual memory requirement and incomplete tests in regression can also limit the scope of the verification. The methodology proposed in this paper improves simulation performance by dynamic memory preloading. Testbench also comprises of C model used for transaction prediction. This C model also uses same memory by C-DPI export. To use dynamic memory preloading some changes are needed in C model that was originally using local static memory. The adopted methodology has improved the simulation performance by as high as 10 times compare to previous testbench for the test that uses maximum supported images of large size with fewer access to each image. It also eliminated need to debug timeouts and better regression management.***

## I. INTRODUCTION

Simulation performance is an important factor that can limit the scope of the verification. The design under test (DUT) is a cache-memory based tile management hardware block placed between the Neural Processing Unit (NPU) and memory that is used to access compressed buffers. This hardware block converts the linear data access of NPU into the tile-based access and uses large, shared SRAM pool. The design supports up to maximum of 64 surfaces with each surface having maximum aperture of 64MB for a 32-bit system. The core DMA can access each of the active surfaces completely or a part of it in random order. This portion of the surface is referred to as Region of Interest (ROI). A high level testbench architecture shown in Fig. 1. The design has two interfaces to send and receive transactions, QNS – a custom AXI protocol on input side and Multi Stream Fifo (MSF) on output side for tile data. A reference C Model is used to create an expected request and response data packets from input request on QNS. MSF VIP memory model is preloaded with random data for each surface that will be exercised in the give test. C model uses a local memory which is also statically preloaded with the same random data. C model uses data stored in local memory and a reference to DUT register configuration to predict expected packets. The scoreboard is used to check data integrity by comparing the observed packets on interfaces and expected packets generated by C model. This approach is highly inefficient due to memory preloading to both C model and VIP memory.

If the size of the ROI is very small compared to the surface aperture, then most of the preloaded data will not be accessed. To verify back-to-back traffic scenario with different configurations in a single test, preloading overhead increases proportionately with number of traffic loops in the given test. For each configuration, surface format and address may change, and hence existing preloaded data cannot be used. For the worst-case configuration that uses all the 64 surfaces with maximum aperture, smaller ROI size and multiple traffic loops dual static memory preload approach results in extremely poor simulation performance. As system platform is upgraded from 32 bits to 42 bits, the number of surfaces required to be supported by DUT is increased from 64 to 128 with maximum aperture size per surface increasing considerably from 64MB to 32GB. This creates a major bottleneck as simulation performance becomes worst resulting in many incomplete tests in regression. Handling incomplete tests requires additional debug efforts and bandwidth which may affect project timeline. More importantly it limits the scope of the verification which is not acceptable.

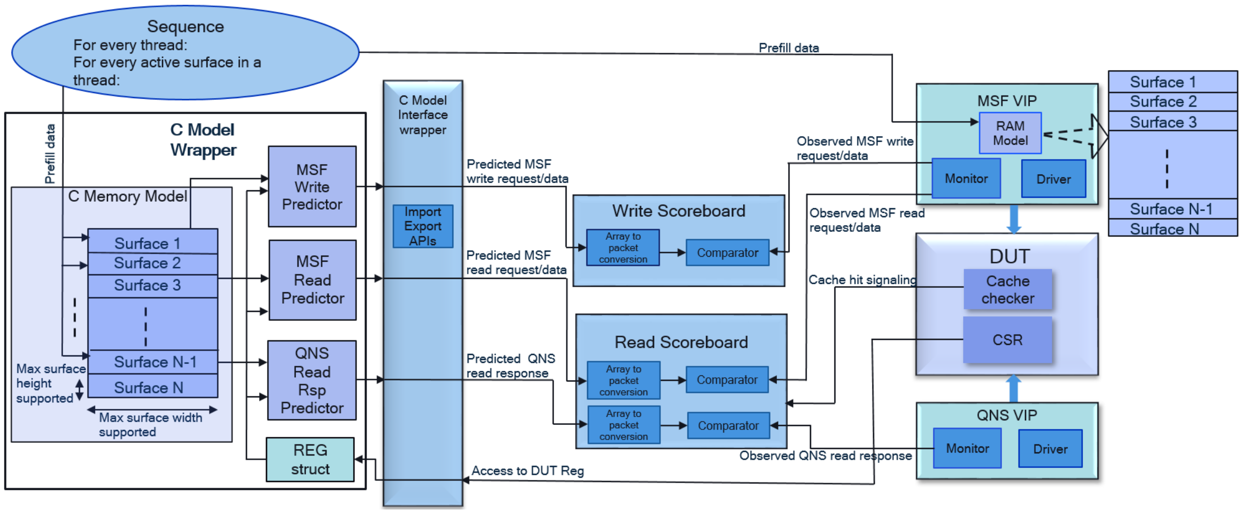
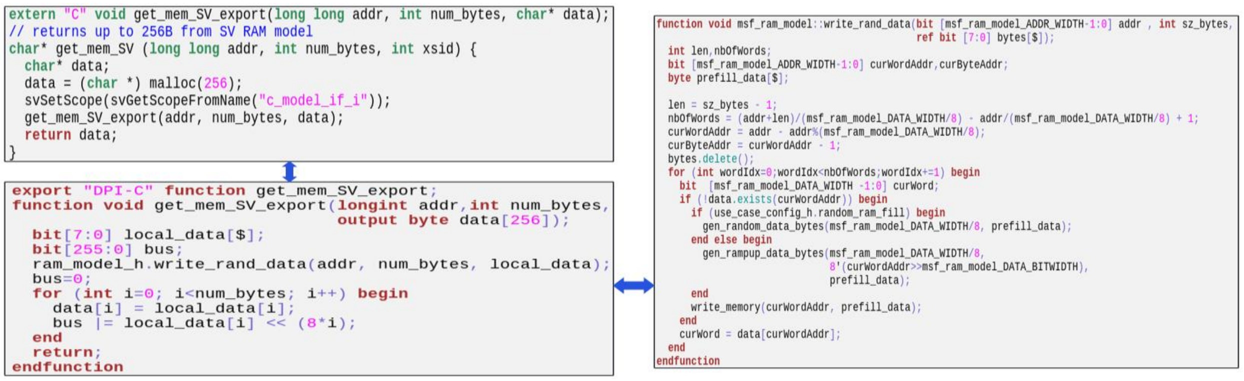


**Figure 1.** Testbench Architecture with C model and memory preloading

Proposed solution in this paper eliminates the need for preloading two memory models. It also uses dynamic preloading approach instead of static which means only the portion of the image that is accessed in a test is written in the memory. With this approach preloading overhead is directly proportional to the number of memory access which is ROI and not the size of the image. C model uses same memory data instead of local memory using System Verilog (SV) memory access function which is exported. With dynamic single memory preloading approach simulation performance is not affected by the configuration parameters such as size of the image, number of images and number of traffic loops if the size of the ROI is relatively small. This approach enabled the verification of full 42bits platform with larger aperture without affecting simulation performance.

## II. IMPLEMENTATION

In single memory, dynamic preloading approach, local memory used in C model is removed. C model uses same VIP memory data for generating expected packets. This method reduces preloading overhead by half in all the cases. Access to SV memory data by C model is achieved using a function with standard C-SV export import provided by Direct Programming Interface (DPI). As shown in Fig. 2. function "get_mem_SV_export" which is defined in SV is exported to C. This function is called by another API defined in C model. Testbench uses C model interface wrapper to handle all the communication between SV domain of the testbench and C model. This interface wrapper is used to send DUT register configuration to model which is needed to generate expected packet address.



```
extern "C" void get_mem_SV_export(long long addr, int num_bytes, char* data);
// returns up to 256B from SV RAM model
char* get_mem_SV (long long addr, int num_bytes, int xsid) {
  char* data;
  data = (char *) malloc(256);
  svSetScope(svGetScopeFromName("c_model_if_i"));
  get_mem_SV_export(addr, num_bytes, data);
  return data;
}
```

```
export "DPI-C" function get_mem_SV_export;
function void get_mem_SV_export(longint addr, int num_bytes,
                                output byte data[256]);
  bit[7:0] local_data[$];
  bit[255:0] bus;
  ram_model_h.write_rand_data(addr, num_bytes, local_data);
  bus=0;
  for (int i=0; i<num_bytes; i++) begin
    data[i] = local_data[i];
    bus |= local_data[i] << (8*i);
  end
  return;
endfunction
```

```
function void msf_ram_model::write_rand_data(bit [msf_ram_model_ADDR_WIDTH-1:0] addr , int sz_bytes,
                                              ref bit [7:0] bytes[$]);
  int len,nbOfWords;
  bit [msf_ram_model_ADDR_WIDTH-1:0] curWordAddr,curByteAddr;
  byte prefill_data[$];

  len = sz_bytes - 1;
  nbOfWords = (addr+len)/(msf_ram_model_DATA_WIDTH/8) - addr/(msf_ram_model_DATA_WIDTH/8) + 1;
  curWordAddr = addr - addr%(msf_ram_model_DATA_WIDTH/8);
  curByteAddr = curWordAddr - 1;
  bytes.delete();
  for (int wordIdx=0;wordIdx<nbOfWords;wordIdx+=1) begin
    bit  [msf_ram_model_DATA_WIDTH -1:0] curWord;
    if (!data.exists(curWordAddr)) begin
      if (use_case_config_h.random_ram_fill) begin
        gen_random_data_bytes(msf_ram_model_DATA_WIDTH/8, prefill_data);
      end else begin
        gen_rampup_data_bytes(msf_ram_model_DATA_WIDTH/8,
                              8'(curWordAddr>>msf_ram_model_DATA_BITWIDTH),
                              prefill_data);
      end
      write_memory(curWordAddr, prefill_data);
    end
    curWord = data[curWordAddr];
  end
endfunction
```

**Figure 2.** SV memory access in C model using DPI export

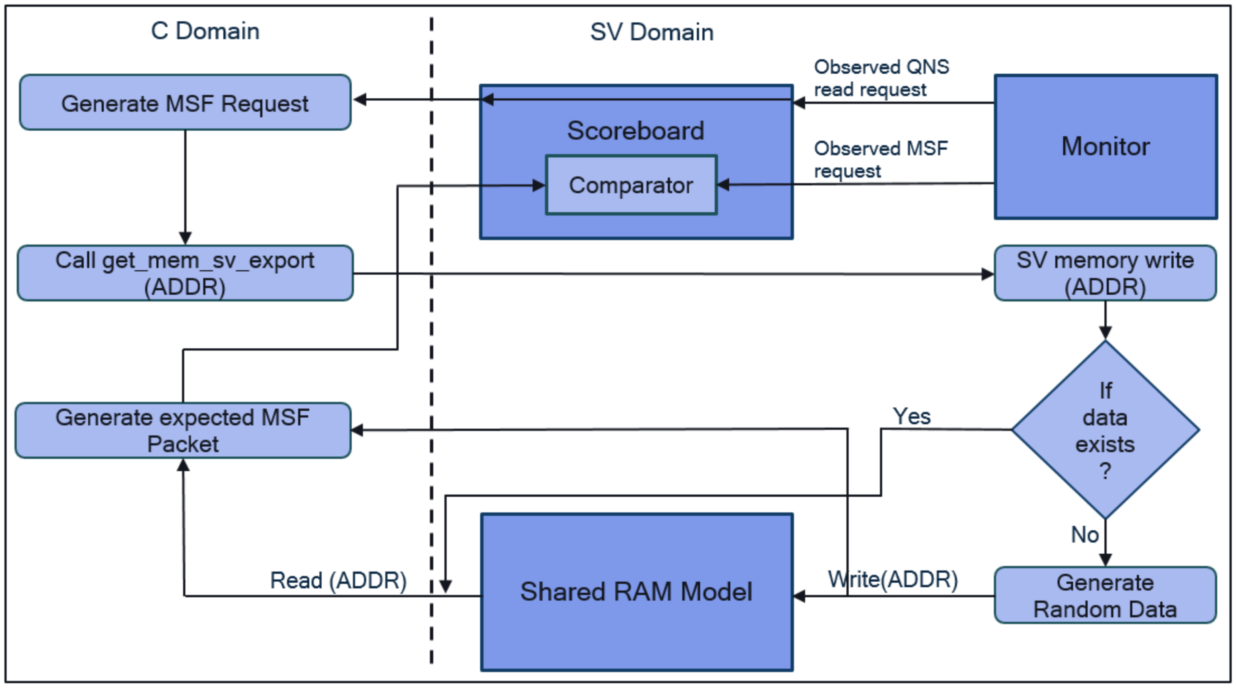


**Figure 3**. Dataflow in optimized preloading testbench implementation

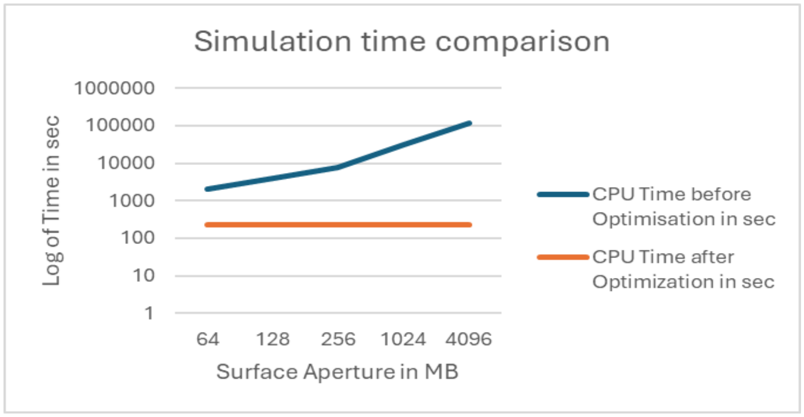
To implement optimized preloading, some changes are needed in C model as well as SV part of the testbench that includes environment top and scoreboard. A complete data flow for one input read request on QNS is shown above in Fig. 3. No random data is statically preloaded in MSF VIP memory. QNS VIP monitor send observed QNS read request to scoreboard. Scoreboard invokes C model using wrapper interface to build expected MSF packets. DUT behavior is such that for a single QNS read request it generates up to 8 MSF read requests depending on image format. C model uses DUT configuration information to generate physical MSF memory address. Once this MSF address is predicted model invokes the "get_mem_sv_export" function and send the generated MSF address and an argument this function. SV function generates random data and write this data to a memory address provided by C model. It also returns the same data to C model for generating expected MSF packet both request and response data. As same memory location can be read multiple times, this solution is further optimized to check if data already exists in memory and skip the duplicate writes. Same data flow is also used to generate and compare QNS response with observed packet. It is clear from the flow diagram that number of memory preload accesses is highly optimized thereby eliminating simulation performance bottleneck and allowing for larger verification state space simulation.

## III. RESULTS

Below graph shows simulation time and virtual memory usage comparison between static and dynamic memory load. A significant reduction in simulation time and virtual memory size is observed resulting in effective regression performance and management. Static memory preload approach shows an exponential increase of simulation time and virtual memory size as the aperture of the image increases. This methodology allows verifying worst case configuration in terms of image size and number of images without degrading simulation performance and virtual memory utilization.

TABLE I
CPU TIME AND VIRTUAL MEMORY SIZE WITH STATIC VS DYNAMIC PRELOADING

| Surface Aperture (Megabytes) | CPU Time (Seconds) | | Virtual Memory Utilization (Megabytes) | |
|---|---|---|---|---|
| | Static Preloading | Dynamic Preloading | Static Preloading | Dynamic Preloading |
| 64 | 2025 | 230 | 16988 | 1976 |
| 128 | 4002 | 230 | 33976 | 1976 |
| 256 | 7533 | 230 | 68002 | 1976 |
| 1024 | 31568 | 230 | 289772 | 1976 |
| 4096 | 119219 | 230 | 1088903 | 1976 |



**Graph 1**. Simulation time for static and dynamic preloading

## IV. CONCLUSION

This proposed solution can be used in any IP verification testbench that uses memory preloading and C model to predict expected data. It uses standard DPI interface which is easy to implement in different types of environments. Due to effective virtual memory utilization and improved simulation time, it also reduces engineering cost and time required for the verification.