Tips for Developing Performance Efficient Verification Environments

Prashanth Srinivasa
LSI India R&D Pvt. Ltd
GTP, Devarabeesanahalli
Outer Ring Rd, Bangalore, India
(+91)80-41979643
Prashanth.Srinivasa@lsi.com

Sarath Chandrababu Valapala
LSI India R&D Pvt. Ltd
GTP, Devarabeesanahalli
Outer Ring Rd, Bangalore, India
(+91)80-41979598
Sarath.Valapala@lsi.com

Varun S
Synopsys India R&D Pvt Ltd
RMZ Infinity, Benniganahalli,
Old Madras Rd, Bangalore, India
(+91)80-40188424
Varun.S@synopsys.com

ABSTRACT

SystemVerilog is being extensively used in the industry for the verification of today's complex designs. Standard verification methodologies like Universal Verification Methodology (UVM) and Verification Methodology Manual (VMM) provide sufficient guidelines and base class libraries to build efficient reusable verification environments. Despite these guidelines, verification engineers spend considerable time debugging non-functional issues like slowdowns and memory blowups in verification environments. This could be due to inappropriate use of UVM/VMM features in a verification environment or also because of the lack of a process to check these issues in an early stage. This paper deliberates over such points which can reduce the effectiveness of an exhaustive and complex testbench. It provides guidelines to improve the quality of the testbench code with the help of examples. The paper also emphasizes the importance of profiling verification environments in order to get optimal simulation time and run-time memory, citing examples based on our experience.

Categories and Subject Descriptors

D.3.3 **[Programming Languages]:** Language Constructs and Features – *abstract data types, polymorphism, control structures.* This is just an example, please use the correct category and subject descriptors for your submission.

General Terms

Verification.

Keywords

VMM: Verification Methodology Manual, by Synopsys
OVM: Open Verification Methodology
UVM: Universal Verification Methodology, by Accellera
VIP: Verification Intellectual Property
RAL: Register Abstraction Layer
DUT: Design Under Test
BFM: Bus functional model
TLM: Transaction Level Model
EDA: Electronic Design Automation

1. INTRODUCTION

Due to growing complexity of the chips and time-to-market pressures, verification cost is significantly increasing in terms of verification engineers, hardware resources and EDA licenses. Making optimal use of the existing resources saves a considerable amount of verification time and cost. For simulation-based functional verification, SystemVerilog is widely used to develop verification environments using standard methodologies like VMM/UVM. These methodologies provide sufficient guidelines and base class libraries to quickly build reusable verification environments. However, if these methodologies and the language features are not used carefully, then they might lead to simulation slowdowns and memory leaks. Debugging such issues is not easy especially when the verification environment is complex, having base class libraries, applications, VIPs, etc. In this paper, we describe the possible reasons behind such problems and steps to quickly identify and resolve such problems by sharing our experiences.

2. PERFORMANCE EFFICIENT VERIFICATION ENVIRONMENT

A verification environment is performance efficient when it consumes only the required amount of hardware resources. That means it should consume least run-time memory and least simulation time without making any compromise on its quality. We are describing various guidelines with the help of examples and case studies we followed in making our verification environments performance efficient. Although methodology related examples we are describing are based on VMM, they are applicable to UVM and OVM as well, unless explicitly mentioned.

3. RUN-TIME MEMORY MANAGEMENT

When the simulation starts, the simulation data base (having the information of the design and the testbench) is loaded for execution. Accordingly machine memory is allocated for holding data base information throughout the simulation and this run-time memory keeps changing whenever verification components get created or destroyed. If the run-time memory keeps on increasing unexpectedly, then it is called memory leak and when it reaches the maximum available memory, then the simulation aborts abruptly. This section discusses run-time memory issues, identifying and analyzing them, and steps to avoid such issues while developing testbenches.

3.1 Understanding Memory leak

SystemVerilog provides dynamic data-types like associative/dynamic arrays, queues and classes. These data types are very useful in modeling a variety of testbench components like sparse memories, transaction data models, BFMs, etc. Whenever a dynamic component gets created, machine memory gets allocated during run-time, and whenever the component gets destroyed, machine memory which was allocated earlier gets deallocated. If the dynamic objects keep getting created and not destroyed, then run-time memory keeps growing causing memory leak. This not only slows down the simulation, but also can lead to memory allocation failures resulting in simulation abortion. There is no automatic mechanism to monitor the size of the dynamic data-types. A simple example is shown in Fig 1. where objects are put into a mailbox from a generator, but are not retrieved anywhere, leading to memory blow up.

```
program P;

class transaction;
  rand bit [31:0] addr;
  rand bit [31:0] data;
  rand bit direction;
endclass

class generator;
  mailbox mbox;

  function new(mailbox mbox=null);
    this.mbox = mbox;
  endfunction

  task run();
    transaction data;
    while (1) begin
      data = new(); data.randomize();
      if (mbox != null) mbox.put(data);
      #100;
     end
  endtask

endclass

initial begin
  mailbox mbox;
  generator gen;
  mbox = new();
  gen = new(mbox);
  gen.run();
end
endprogram
```

Figure 1. Example of memory leak

3.2 Identifying root cause for Memory leak – Case study

It is often trivial to determine if there is a memory leak during the simulation, but it is not easy to find the root cause for the memory leak(s). The following describes the approach we used for one of our block level environments. The environment was VMM based using RAL application library for accessing and verifying DUT registers, internal VIPs and the testbench components like scoreboard, scenarios, etc.

3.2.1 Check for the memory leak

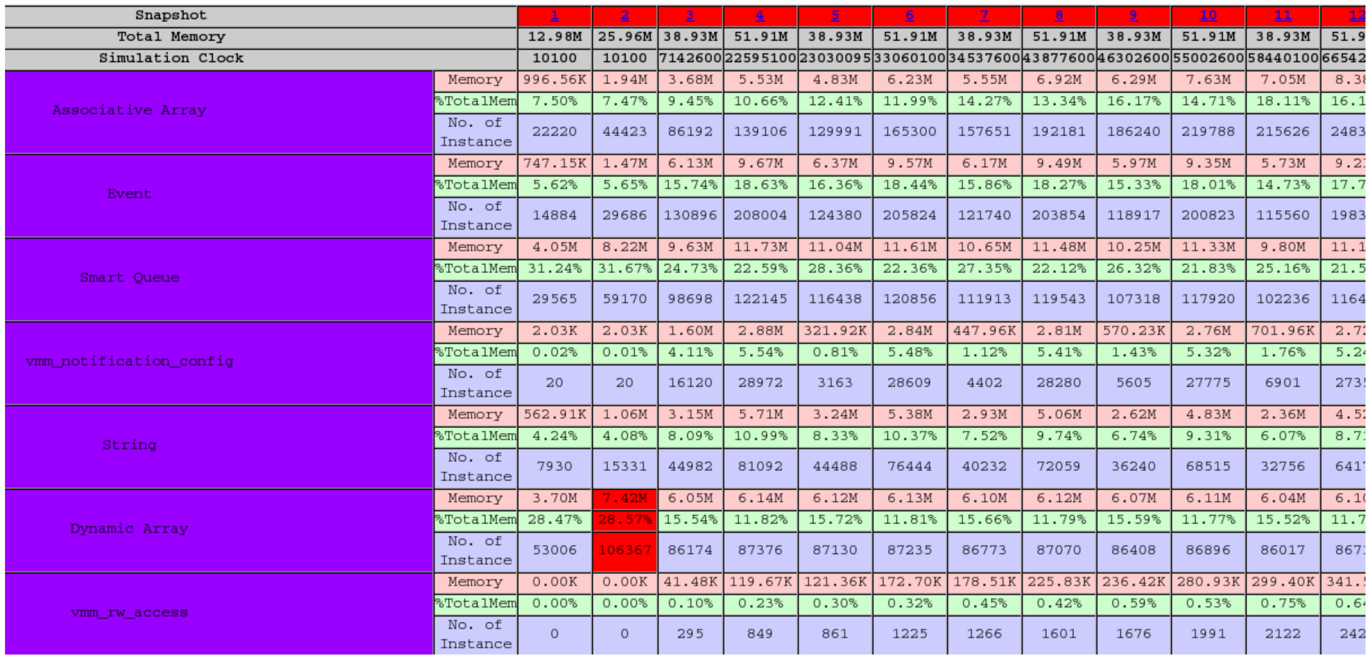
We selected a long running test case and started monitoring the run-time memory using the UNIX command *top –p <process id>*. When the simulation is started, initially at zero simulation time, there was a steep increase in run-time memory to around 140MB. We understood that it was because of initial loading of the data base. Then after few seconds again memory started increasing gradually to 352MB till the end of the simulation. Although simulation completed successfully, we realized that some dynamic objects were not getting deallocated, causing a memory leak which needed to be fixed, otherwise we would run into problems at the sub-system/system level.

3.2.2 Finding the root cause and fixing the issue

It is difficult to find the root cause if there is no profiling mechanism provided by the simulation tool. We used the simulation memory profiler provided by the tool. We ran the simulation again enabling the profile, and generated a HTML report through the tool provided utility. This report provided information about the memory consumption of different components and classes at different intervals of time as shown in the Figure 2. We noticed that associative array elements, events and some class objects were growing throughout the simulation. We started looking at the transaction related classes (since they are the ones which get created thousands of times) *axi_master_bfm_burst* and *vmm_rw_access*. For axi_master_bfm_burst, the number of instances at any simulation time was fluctuating between 5 and 2000. But for *vmm_rw_access* the number of instances was growing constantly to around 40000. The objects for this class were supposed to get created and destroyed for any register write or read through the RAL, but somehow objects were not getting destroyed. Since this class was not used directly by us, we sought guidance from the vendor providing the RAL application and found that, whenever this object gets created, it was pushed into a *vmm_object* associative array, but was not removed from it any time. The vendor addressed this issue by preventing the object from getting pushed into the associative array. When we ran the simulation again with the fixed RAL application, vmm_rw_access instances stopped growing and started fluctuating between 0 and 200. This not only reduced the run-time memory, but also improved the simulation performance as shown in Table 1.

Table 1. Comparison of run-time memory and simulation performance

Metric	Before memory leak fix	After memory leak fix
Peak run-time memory	352 MB	188MB
Simulation run-time	358.880 sec	283.690



Snapshot		1	2	3	4	5	6	7	8	9	10	11	12
Total Memory		12.98M	25.96M	38.93M	51.91M	38.93M	51.91M	38.93M	51.91M	38.93M	51.91M	38.93M	51.9
Simulation Clock		10100	10100	7142600	22595100	23030095	33060100	34537600	43877600	46302600	55002600	58440100	66542
Associative Array	Memory	996.56K	1.94M	3.68M	5.53M	4.83M	6.23M	5.55M	6.92M	6.29M	7.63M	7.05M	8.38
	%TotalMem	7.50%	7.47%	9.45%	10.66%	12.41%	11.99%	14.27%	13.34%	16.17%	14.71%	18.11%	16.1
	No. of Instance	22220	44423	86192	139106	129991	165300	157651	192181	186240	219788	215626	2483
Event	Memory	747.15K	1.47M	6.13M	9.67M	6.37M	9.57M	6.17M	9.49M	5.97M	9.35M	5.73M	9.2
	%TotalMem	5.62%	5.65%	15.74%	18.63%	16.36%	18.44%	15.86%	18.27%	15.33%	18.01%	14.73%	17.7
	No. of Instance	14884	29686	130896	208004	124380	205824	121740	203854	118917	200823	115560	1983
Smart Queue	Memory	4.05M	8.22M	9.63M	11.73M	11.04M	11.61M	10.65M	11.48M	10.25M	11.33M	9.80M	11.1
	%TotalMem	31.24%	31.67%	24.73%	22.59%	28.36%	22.36%	27.35%	22.12%	26.32%	21.83%	25.16%	21.5
	No. of Instance	29565	59170	98698	122145	116438	120856	111913	119543	107318	117920	102236	1164
vmm_notification_config	Memory	2.03K	2.03K	1.60M	2.88M	321.92K	2.84M	447.96K	2.81M	570.23K	2.76M	701.96K	2.7
	%TotalMem	0.02%	0.01%	4.11%	5.54%	0.81%	5.48%	1.12%	5.41%	1.43%	5.32%	1.76%	5.24
	No. of Instance	20	20	16120	28972	3163	28609	4402	28280	5605	27775	6901	2735
String	Memory	562.91K	1.06M	3.15M	5.71M	3.24M	5.38M	2.93M	5.06M	2.62M	4.83M	2.36M	4.5
	%TotalMem	4.24%	4.08%	8.09%	10.99%	8.33%	10.37%	7.52%	9.74%	6.74%	9.31%	6.07%	8.7
	No. of Instance	7930	15331	44982	81092	44488	76444	40232	72059	36240	68515	32756	641
Dynamic Array	Memory	3.70M	7.42M	6.05M	6.14M	6.12M	6.13M	6.10M	6.12M	6.07M	6.11M	6.04M	6.1
	%TotalMem	28.47%	28.57%	15.54%	11.82%	15.72%	11.81%	15.66%	11.79%	15.59%	11.77%	15.52%	11.7
	No. of Instance	53006	106367	86174	87376	87130	87235	86773	87070	86408	86896	86017	867
vmm_rw_access	Memory	0.00K	0.00K	41.48K	119.67K	121.36K	172.70K	178.51K	225.83K	236.42K	280.93K	299.40K	341.
	%TotalMem	0.00%	0.00%	0.10%	0.23%	0.30%	0.32%	0.45%	0.42%	0.59%	0.53%	0.75%	0.64
	No. of Instance	0	0	295	849	861	1225	1266	1601	1676	1991	2122	242

Figure 2. Testbench dynamic memory profile report for different components

3.3 Guidelines to avoid memory leaks

Development of a verification environment with the knowledge of language and methodology features from the memory consumption point of view helps in avoiding memory leaks. Following are some of the tips which help in this regard.

3.3.1 Threshold values for queues and associative arrays

Most memory leak issues are due to uncontrolled increase in the size of queues and associative arrays. Monitoring their size and avoiding their direct access externally avoids the unexpected memory growth.
In the example shown below, queue trQ of the class scoreboard is declared *protected* so that no other class can access it directly. Method *add_trans*() is provided to add transactions into the queue by the external components. In addition to adding transaction into the queue, this method monitors the size of the queue and flags an error if the queue size exceeds a reasonable threshold value of 1000. This way, the excessive addition of objects by external components can be detected and readily corrected.

```
program P;

class transaction;
  rand bit [31:0] addr;
  rand bit [31:0] data;
endclass

class scoreboard;
  protected transaction trQ[$];

  function void add_trans(transaction tr);
    if (trQ.size() >= 1000) begin
      $display("FAILURE! trQ size exceeds 1000, cannot add any more");
      return;
    end
    trQ.push_back(tr);
  endfunction
  function void remove_trans();
    trQ.pop_front();
  endfunction
endclass

scoreboard sb;
initial begin
  sb = new;
  for (int i=0; i<1005; i++) begin
    transaction tr = new;
    sb.add_trans(tr);
  end
end
endprogram
```

3.3.2 Avoid generating too many data models at a time

It is often necessary to generate thousands of transactions in a verification environment. Instead of generating a big set of transactions at one time and sending them one by one to the driver/DUT, it is better to generate transactions one by one (or a small set) and send them to the driver/DUT before generating the next transaction. This way, run-time memory can be saved. This is not a problem if the transaction data structure is small, but if each transaction consumes several kilobytes of memory, then it is better to follow the guideline. Sometimes, when even a single transaction size is too high, it might need to be generated in chunks.

3.3.3 Methodology specific memory issues

Users developing verification environments using the base class libraries provided by standard methodologies like UVM/VMM can follow the below mentioned guidelines to avoid memory related issues.

3.3.3.1 Use Callbacks/Analysis ports instead of non-blocking channels

Commonly used communication mechanisms among the verification components are channels, TLM ports and callbacks. Passive components like monitors typically send transactions through TLM

analysis ports, callbacks or non-blocking channels. It is recommended to use callbacks/analysis ports to send transactions rather than non-blocking channels.
If a transaction is sent to a non-blocking channel and if there is no subscriber (e.g., scoreboard) to pop the transaction from the channel, then transaction objects remain in the channel without getting deallocated causing a memory leak. But if Callbacks or TLM analysis ports are used to send the transactions, even if there are no subscribers, transaction objects will not get into any queues/arrays. It is always safe to use callbacks/analysis ports to send transactions when the receiving end is not known.

3.3.3.2 Using *vmm_log* in VMM based environments

In VMM based verification environments *vmm_log* class is used for message handling. Any instance of this class once created will not get destroyed till the end of the simulation (unless a method vmm_log::kill() is explicitly called). If a *vmm_log* instance in a transaction class is initialized, then whenever a transaction object gets created, *vmm_log* instance also gets created causing numerous objects to not get deallocated. This results in memory blow up. To avoid this, *vmm_log* instance has to be declared *static* and should be implicitly initialized as shown in the below example. This should not be done when *vmm_log* is instantiated in static components like transactors where the number of objects that get created are very few. If *vmm_log* instances are declared as *static* in a transactor, then messages from all the instances of the transactor come from a single instance making it difficult for the user to know which transactor is sending the messages.

```
class transaction extends vmm_data;
  rand bit [31:0] addr;
  rand bit [31:0] data;
  static vmm_log log = new(“transaction”, “class”);
  …..
endclass

class driver extends vmm_xactor;
   …..
   vmm_log log;

   function new(…);
      log = new(“driver”, instance);
   endfunction
endclass
```

3.3.3.3 Passing parent handle to vmm_data in VMM based environments

In UVM/VMM based verification environments, parent-child hierarchy can be maintained by passing the parent handle through constructor argument. This is required for traversing across the hierarchy to search for a component. But this prevents any component in the hierarchyfrom getting deallocated. This is fine for static components like transactors, scoreboards, etc (as they get created once and remain till the end of the simulation), but not for VMM based transactions which are supposed to get created and destroyed. So, for transaction classes, a parent handle should not be passed.
Passing parent handle to uvm_object based transactions is not a problem since UVM does not saves transactions in its object hierarcy list.

3.3.4 Use testbench profiling early in the testbench development.

When the testbench becomes complex, it is difficult to find the root cause for a memory leak even with a profiling mechanism. It is better to profile the testbenches at the block level so that it becomes easy to debug at higher levels. Also it helps in improving the simulation performance. Whenever there is any major change in a testbench, it is recommended to run profiling for a few selected test cases.

4. SIMULATION PERFORMANCE

Testbench constructs contribute to a considerable amount of run-time with different components like generators, monitors, scoreboards, etc. This section provides a set of guidelines in developing a verification environment to make it run as fast as possible.

4.1 Guidelines to develop a performance efficient testbench

4.1.1 Use innovative approaches in optimizing performance

4.1.1.1 Memory model optimization

If there is a requirement for modeling a centralized memory (like the system memory) in the verification environment it is natural to implement it using an associative array or a queue in SystemVerilog. For example, we can define an associative array of 8 bits (like shown below) to model a system memory and keep allocating some part of it to the user/job requesting the memory.

```
 bit [7:0]  data_mem [*];
```

We might well be deallocating the memory once the user/job is done with it (there by making sure that we are not running into memory leak issues) but there is an issue with this kind of implementation.
As the number of simultaneous users/jobs increase and their cumulative memory requirement at any particular instance is huge (around 1Mega Bytes for e.g.), the memory operations slow down. In our project we observed that the deallocation of memory (deleting the memory allocated to a job by deleting the corresponding locations of the associative array in a “loop” one at a time using the .delete() method of associative array) was consuming huge run-time because of the size of the associative array (memory model). In the project each job needed 64 Kilo Bytes of memory and some of the test cases (where the number of outstanding jobs was more) used to take approximately 7 hours to finish. Once we understood that the deallocation is taking more time because of the growing size of the associative array, we remodeled the system memory by embedding it in a class as shown below and creating an object for each job (memory allocation for that job), thus creating an array of those objects to model the entire memory.

```
class data_mem_c;
  bit [7:0]  data_mem [*];
endclass
data_mem_c  data_mem_obj_arr[*];
```

In this implementation, each job is allocated an associative array (an object of the class) and the collection of those arrays (array of the objects) form the system memory. Deallocation of memory of a job is done by deleting the corresponding object in the array of objects (data_mem_obj_arr). This is a better implementation for deleting the memory than the earlier method of deleting each location, one at a time, in a “loop”. In our project this resulted in a drastic reduction in run-time to approximately 20 minutes (from approximately 7 hours with the earlier implementation). In a nutshell, it is always better to use small chunks of memory (through objects and arrays) and operate on them instead of a single big memory, even if the requirement is to model a centralized memory.

4.1.1.2 String specific optimization

If there is a requirement to compare strings from a source with an existing database (for example, to check if a parsed string pattern matches with any element of an array of strings) and there is a chance of repeated occurrence of the same source string, then using a modeling caching mechanism instead of direct comparison will improve the simulation performance. In the example shown below, there is a *command_database* class which has an array of string patterns. There is a cache model *command_cache* to store frequently occurring string patterns. When the method *get_command()* of *command_database* is called, a search for the pattern happens first in the cache model which is very fast since the array size is small. If there is a cache miss, then normal search happens in the command database. The performance benefit here depends on the amount of frequent occurrences of the same source patterns (i.e., number of cache hits).

```
class command_cache; //Cache model
  string cmds[10];
  int num_of_occurrences[10];
  int cache_size;

  function int get_cmd(string str);
    foreach (cmds[i]) begin
      if (cmds[i] == str) begin
        num_of_occurrences[i]++;
        return cmds[i];
      end
    end
    return (-1);
  endfunction

  function void add_cmd(string regex);
   if (cache_size < 10) begin
     cmds[cache_size] = regex;
     cache_size++;
   end
   else begin
     int p[$], k;
     p = num_of_occurrences.min();
     k = p[0];
     cmds[k] = regex; num_of_occurrences[k] = 0;
   end
  endfunction

endclass

class command_database;
  command_cache cache = new;
  int commands[string];

  function int get_command(string regex);
    int val = cache.get_cmd(regex); //Fast Cache path
    if (val >= 0) begin
       return(val);
    end
    foreach (commands[i]) begin //Slow normal path
      if (i.match(regex)) begin
         cache.add_cmd(regex);
         return commands[i];
      end
    end
    return (-1);
  endfunction

endclass
```

4.1.2 Data model/Transaction optimization

During the simulation of a verification environment, most of the testbench time is consumed by the transaction related activities. Optimized transaction generation and processing mechanisms and judicious use of message features help in improving the simulation performance to a large extent.
Generation involves creation and randomization of transactions. During randomization, constraints specified get solved in parallel. However, if the constraints are complex, the constraint solver takes longer time to solve the constraints leading to slowdowns in the simulation. Partitioning the constraints and generating them sequentially, like moving some of the dependent value generation to post_randomize() from the constraint block will result in faster generation. Of course this must be done without compromising the quality and controllability of generation.

4.1.3 Methodology specific guidelines

4.1.3.1 Message optimization

Messages are the errors, warnings, and other information that is displayed to the terminal or log file to know the status and debug a test. They affect the run-time in two ways:

1. Time taken to process the message string.
2. Time taken to output the message to a terminal or log file.

All the methodologies suggest the usage of their built in log mechanism instead of using the simple "$display" task provided in SystemVerilog. For e.g. VMM provides "vmm_log" base class with methods like "start_msg", "text", and "end_msg" for displaying the messages.
The inbuilt log mechanism is effective in reducing the run-time and improving the performance of a test. It has two important features that counter the two run-time issues of messages mentioned above.

4.1.3.1.1 Use macro based/filter enabled message features

The log mechanism checks the message filters before processing the message string. This eliminates the necessity of processing the message string that would eventually be filtered out because of the filters enabled in the test. For e.g. if the default severity is "ERROR", all the lower severity message strings will never be processed. This saves a lot of time if the message strings are formatted strings, which is the case most of the time.
In VMM this feature is provided through the "start_msg" method of "vmm_log" base class. The code below demonstrates its usage.

```
if (log.start_msg (vmm_log::DEBUG_TYP,
                   vmm_log::TRACE_SEV)) begin
  log.text (tr.psdisplay());
  log.end_msg();
end
```

The "if" condition checks whether the DEBUG type message is not filtered out and the severity level is higher than TRACE. If it is filtered out or if it is not severe enough, the message text is not processed at all. i.e., tr.psdisplay() is not all executed which saves time. The above code can also be written using message macro as shown below.

```
`vmm_debug (log, tr.psdisplay());
```

The above macro does the same job of checking for the message filters and then processing the text and printing it to the output. It is advised to use the macros to improve the performance when a lot of formatted messages are to be processed.
In case of UVM/OVM, always use macro based messaging (like `uvm_error, `uvm_info, etc) as shown below instead of using the message method directly (uvm_report_error, uvm_report_info, etc). This will prevent execution of the second argument and thus improve the simulation performance.

```
`uvm_info (get_type_name(), tr.sprint(), UVM_FULL);
```

4.1.3.1.2 Use messages with appropriate severity and control them

The log mechanism forbids displaying messages that are filtered out. This eliminates the necessity to flush the message text to a terminal or log file thus saving time. For e.g. if the default severity is "ERROR", all the lower severity messages (like "WARNING", "NOTE", etc) are not displayed.
In VMM this feature is provided through the "set_verbosity", "disable_types" methods of "vmm_log" base class. Using these methods the required message type and severity level can be selected and the others can be filtered out there by reducing both the number of messages to be printed and the run-time. Simulator switches like "+vmm_log_default=<value>", and "+vmm_force_verbosity=<value>" are the alternatives for the built in methods to filter the messages and improve the performance.
Both the features described above are necessary for providing an optimum message service in terms of run-time and all the methodologies have these built in. If the test environment is developed independent of any methodology then it is important to include these features as part of the log mechanism for better run-time performance.

4.1.3.2 Using log catcher

Log catcher mechanism is provided in verification methodologies like VMM, UMM, etc. to identify a message issued by any verification component (termed as 'catching a message') and execute some specific code once the required message is 'caught'. It is a three step process:

1. Select the message to be caught.
2. Catch the message.
3. Execute the required code upon catching the message.

The first step is implemented by matching the required string pattern in all the messages (using regular expressions [regexp]). The second step is implemented using the methodology provided log catching methods (for e.g. VMM provides vmm_log::catch() method). The third step is also implemented by using the methodology provided methods (for e.g. VMM provides vmm_log_catcher::caught() and vmm_log_catcher::issue() methods). Usually the third step involves modifying the 'caught message' itself (for e.g. prefix a string "Expected Error" to an error message) and/or changing its severity, though executing any code is permissible.
Using a log catcher mechanism is performance intensive since it involves a string matching attempt for every message (the first step as described above). So, use the log catching feature of the methodology being followed (if the methodology provides it) only at the test case level for short negative test cases where some error messages are intended. At the environmental level, instead of catching a message, catch the event (may be a flag/callback) responsible for it and process it based on the requirement. Most of the applications served by the log catcher mechanism can be implemented using the other related features like message filters, events, etc. that are less severe on run-time. In summary, the log catcher mechanism should not be used as a general feature of a testbench; it has to be used (if at all there is a need) during exceptions only (like the tests where errors are forced intentionally) to improve the run-time performance.

4.1.3.3 Using run-time options

The capability of passing the desired input values at run-time is provided in SystemVerilog through the system tasks $test$plusargs and $value$plusargs. Similarly all the verification methodologies provide a mechanism that is more sophisticated than the $plusargs of SystemVerilog to pass input values through run-time options (for e.g. VMM provides methods like vmm_opts::get_object_int(), vmm_opts::get_object_string(), etc. to receive the run-time options passed through the simulator switch +vmm_opts+<option>=<value>). Usage of the run-time options reduces the overall test execution time by eliminating the necessity of compiling the source code for each option. But the run-time options are performance intensive since they undergo string processing (string operations are run-time sensitive).
Though there is an effective performance gain while executing all the tests (by saving on the compile time), the run-time of each individual test increases because of the usage of run-time options. So there is a necessity for judicious usage of run-time options. It is advised to use the run-time options in such parts of the testbench code that executes only one time or few times (for e.g., in the configuration phase and build phase that execute during the beginning of a simulation), and not to use them in a transaction or any other place where there is a repeated execution of the function scanning the run-time option as shown in the examples below. The run-time of each individual test can be controlled by following this guideline and thus achieving a better overall performance gain with the usage of run-time options.

```
//NOT RECOMMENDED
class driver;
  int max_length;

  virtual task run();
    while (1) begin //executes many times
       max_length = vmm_opts::get_object_int(….);
       if (len < max_length) …
       ….
    end
  endtask
…
endclass

//RECOMMENDED
class driver;
  int max_length;

  function new(); //executes only once
     max_length = vmm_opts::get_object_int(…);
  endfunction

  virtual task run();
    while (1) begin
       if (len < max_length) …
       ….
    end
  endtask
…
endclass
```

5. SUMMARY

SystemVerilog language aided with verification methodologies provides a rich set of features to build efficient verification environments. However, it is possible to face performance issues if the features are not used appropriately. All these issues can be resolved by proper understanding of the behavior and usage of the features.

In this paper we have discussed some of those features that can lead to memory leaks and simulation slowdowns if used inappropriately. We have also discussed the usage of profiling and threshold values to avoid memory leaks; proposed solutions like a better way of modeling a memory, message and string optimizations, and proper run-time options usage to improve the run-time performance by citing examples from our experience apart from the general guidelines of usage. The tips provided in this paper, if followed along with the methodology guidelines, will help in developing performance efficient verification environments.

6. ACKNOWLEDGMENTS

Thanks to Jayendra Dwaraka Bhamidipatti for his guidance and support.

7. REFERENCES

[1] IEEE Standard for System Verilog - Unified Hardware Design, Specification,and Verification Language. IEEE Computer Society, NewYork: IEEE 2005

[2] Janick Bergeron, Eduard Cerny Alan Hunter and Andrew Nightingale. 2006. Verification Methodology Manual for SystemVerilog

[3] Accellera Univarsal Verification Methodology (UVM) 1.0 Early Adopter California: Accellera 2010

[4] VMM Register Abstraction Layer User Guide, RAL version 1.15