Yikes! Why is My SystemVerilog Testbench So Slooooow?

By Frank Kampf, IBM
Justin Sprague, Cadence Design Systems, Inc.
Adam Sherer, Cadence Design Systems, Inc.

Abstract

It turns out that SystemVerilog != Verilog. OK, we all figured that out a few years ago as we started to build verification environments using IEEE 1800 SystemVerilog. While it did add design features like new ways to interface code, it also had verification features like classes, dynamic data types, and randomization that have no analog (pardon the pun) in the IEEE 1364 Verilog language. But the syntax was a reasonable extension, many more designs needed advanced verification, and we had the Open Verification Methodology (OVM) followed by the standardized Accellera Systems Initiative Universal Verification Methodology (UVM) so thousands of engineers got trained on object-oriented programming. Architectures were created, templates were followed, and the verification IP components were built. Then they were integrated and the simulation speed took a nose dive. Yikes, why did that happen?

Keywords

SystemVerilog, OVM, UVM, object-oriented, performance, scalability

1. SystemVerilog != Verilog

While SystemVerilog and Verilog share much of the same syntax they can't be coded using the same rule sets, especially the testbench aspect of SystemVerilog. Most engineers working with SystemVerilog today were trained on Verilog and learned to code for the static, hardware environments targeted by that language. SystemVerilog testbenches, such as those coded to the UVM standard, are dynamic in terms of both code and data. Furthermore, the testbenches have to manipulate large amounts of data calculated and driven into the design and then process and compare the data retrieved from it.

All of this processing is done with dynamic datatypes, classes, flow-control, and other language features that have no Verilog equivalent. The result can be a testbench that has both more code than the design and accesses more memory. These large SystemVerilog environments present verification engineers and former hardware engineers with numerous innocuous coding opportunities that can result in either reasonable or very poor performance.

2. Performance Programming for Hardware Engineers

For any engineer transitioning to SystemVerilog coding, the most important question to ask is "will it run fast when it scales?" This is the guiding principle that will lead to efficient initial coding and to the rapid debug of performance issues identified during performance profiling.

2.1 Loop Invariants

Loop invariants are values that don't change within the loop but are executed every time through the loop. The potential gain from removing loop invariants depends on the complexity of the invariant calculation,

the body of the loop, and the number of cycles in the loop. For example, if the loop has two relatively simple calculations inside, one of which is invariant, but the loop count is low the gain from moving the invariant outside of the loop will be minimal. However, if the loop executes millions of times, the gain can become material.

A less obvious example of an invariant is the loop end value. In the two code examples below, the second for loop executes much more quickly because the final value for the loop is pre-calculated into the variable `l_end`. Certainly, in this small example the performance difference is insignificant, but consider the scaling. If this loop calculated the security encryption for millions of packets in a monitor to compare that to the data coming through a packet processor, the performance gain could be very large. This example also applies when you use a built in array method like `.size()` on a dynamic array. If the loop check is `myqueue.size()` then the simulator has to dereference `myqueue` and calculate the size every time through the loop. Setting a local variable to the size of the queue, assuming that the loop itself doesn't change the size of the queue, will save simulation time provided that the queue is large and/or the loop is long.

```
int i, a[256], b[256];
int length=4, count=6, l_end;
for (i=0; i < length*count; i++)
     a[i] = b[i];


l_end = length * count;
for (i=0; i < l_end; i++)
     a[i] = b[i]
```

A more subtle form of loop invariant is the dereferencing mentioned in the previous paragraph. In the hardware world, a hierarchical reference can be pre-calculated because the references are static at run time. In a SystemVerilog testbench, the references usually traverse both class instance hierarchies and dynamic types all of which can change during the simulation run. Therefore, the simulator has to walk through all of the references to get to the data. In the code below, a single data value in one class is accessed every time through the loop and assigned to an array in another class. The second loop shows the same functionality written more efficiently.

```
int i, size, key;
Dynamic_array_handle dec_arr;
Dynamic_array_handle enc_arr;
for (i=0; i < comms.proto.pkt.xmt.enc.size(); i++)
   comms.proto.pkt.rcv.dec[i] = decrypt(comms.proto.pkt.xmt.enc[i],
                                        comms.proto.pkt.xmt.key);

size = comms.proto.pkt.xmt.enc.size();
key = comms.proto.pkt.xmt.key;
dec_arr = comms.proto.pkt.rcv.dec;
enc_arr = comms.proto.pkt.xmt.enc;
for (i = 0; i < size; i++);
   dec_arr[i] = decrypt(enc_arr[i], key);
```

2.2 Short-Circuits For Speed

A common compiler optimization is to execute a branch when the minimum number of terms enables short-circuiting the branching condition. For example, the first `if` statement below will cause the branch as soon as the first term is true and the second branch as soon as it is false.

```
if (term1 || term2 || term3)

if (term1 && term2 && term3)
```

From a performance perspective, there is little difference in these statements unless the terms are actually complex method calls or the statements are inside a loop. If the terms are individually complex, then the term that is most likely to cause the branching to execute should be the first in the list. For example, the reader can easily see the performance error in the following code. It is functionally correct, but will run more slowly, especially inside a loop.

```
if (rarely_happens() || sometimes_happens() || nearly_everytime())
      code_to_execute
```

Turning the concept of invariant around is guarding a complex calculation that is not often used. In the code example below, the randomize() method is called outside of the condition in which it was used causing many expensive, but unnecessary calls. Note that this example assumes that `live` is not a rand variable.

```
size = millions_of_pkts.size();
for (i = 0; i < size; i++) begin
    data = millions_of_pckts[i].randomize();
    live = millions_of_pktsp[i].live
    if (live == TRUE)
        inject(data);
end
```

2.3 Respect the Macro

Even as we typed the words, we could hear the shouts – macros are pure evil! That is simply not the case. For sure, excessive use can create problems including challenging debug and code expansion (consuming memory) but judicial use can dramatically speed looping calculations. In the example below, the `reverse()` function was called on all 1,000,000 data points. Each call to `reverse()` required the creation of a stack frame that could affect cache hits (the simulation executes fastest when loop code and data working set is all within the cache memory) and dramatically slow simulation. The macro version, as innocent as it looks in capital letters, runs much more quickly. However, when the drive to use macros for performance causes the coder to directly access data in classes rather than use method APIs, reuse can be diminished trading a performance improvement in a simulation run for overall project-level productivity. Other object-oriented languages have the concept of the `inline` directive that instructs the compiler to insert the code directly inline instead of in a function call; unfortunately, this construct is not available in SystemVerilog yet so users are at the mercy of compilers (and macros).

```
function int reverse(int a) begin
  int i, b = 0, mask = 1;

  for (i = 0; i < 32; i++)
    b |= (a & (mask>>i)) << (32-i);
  begin end
endfunction

left_side[x] = right_side(big_endian[x]);

`define REVERSE (b, a) begin \
  int i, mask = 1; \
  for (i = 0; i < 32; i++) \
    b |= (a & (mask>>i)) << (32-i); \
  begin end \
end

`REVERSE(left_side[x], right_side[x]);
```

2.4 Respect the Static Classes

Static allocation is the object oriented extension to the general programming concept of macros. Where macros replace the hierarchical execution flow by in-lining code, the static classes replace the dynamic creation of classes. In situations where the same set of classes is newed and de-allocated (garbage collected) repeatedly, the simulator repeatedly cycles through memory management. If the classes are statically defined, the overall memory footprint of the simulation remains consistent, but the speed of execution will increase.

As with macros, static allocation can be too much of a good thing. If there is a need to use and dispose of classes quickly, it may be better to implement an object pool of "used" classes that can be parked when they aren't being used. The pool would then be checked first for available classes that could be reset before asking for the allocation of a new class thus saving a de-allocation/reallocation cycle.

It is important to note that, if a pooling strategy is being used, the constructor of the object is not being called for each

3. Light and Quick

Memory is probably the biggest source of mysterious slow-downs. In hardware, the biggest memory issue is modeling physical memory and its footprint is typically quite obvious. However, in the SystemVerilog testbench it can be hidden. Given the small space in this paper, the examples in this section are descriptive scenarios rather blocks of code.

3.1 Performance Depends on Knowledge of Complete Inheritance Hierarchy

Undocumented classes can be a performance nightmare. When you derive a subclass you inherit all of the data and methods in the class hierarchy. But what is in there? When you create instances of your new subclass

Consider this. Your project team has a third generation facial algorithm that distributes the computation from four pipelined processors to an array of 1024 parallel processors. The memory bus arbitration logic has been modified to be hierarchical; your task is to verify the latency and functionality of the new arbiter. Upon reviewing the verification IP for the processor's memory interface from the previous generation, you observe that it has an API for all of the memory access modes. You then derive a new class for the third generation and add an array to hold the transaction history to measure the latency. You then create an instance of the verification IP for every processor, set the history array size to 32 and the simulation memory explodes. What happened?

What you didn't know is that the base class had an array to support the verification of a DMA mode that was no longer used. That array used your history-sizing class object, but interpreted that integer as kilobytes of DMA rather than single transactions.

While contrived, this description points to poor class development and management. Data objects should only be accessible through a method API to avoid unanticipated access. Interfaces should also be used to clearly define APIs so that redundant structures are not added in derived classes. By carefully studying the base classes and planning the derived classes, problems associated with hidden data can be avoided.

3.2 Lost Handles Make Performance Fall

SystemVerilog classes are allocated on demand and freed through garbage collection. The allocation process occurs when a the `new()` method for a class is called but the garbage collection occurs under two conditions: either when the number of references to the class handle falls to zero or a simulation engine tracing algorithm detects object graphs that are self-referencing but lack direct user references. If the lifecycle of the classes, and other dynamic objects, is not managed the heap memory continues to grow.

Consider an intelligent packet processing protocol. The verification of that protocol may include both a check that the individual packet was transmitted correctly and that the performance of the output interface adjusts to the type of traffic that passes through it. For this example, assume that a queue is used to feed the input, but rather than popping each packet as it is processed the next packet is accessed by moving an index pointer. In this case, the protocol verification IP will execute properly, but the heap memory will grow quickly because the input queue will continue to grow with references to packet class handles that are never garbage collected. Dynamic arrays and associative arrays are also affected this way with associative arrays being the most typical case of unexpected *memory leakage*.

While this bug may seem obvious, lost handles can be insidious. If an array counter range is improperly set, handles may not be cleared properly. It is recommended that additional checks be added to the code to detect overflows. Global arrays are especially insidious because multiple threads may operate on them with unexpected side effects. Engineers do generally plan for the creation of these SystemVerilog objects, but the testbench development must also plan for their destruction.

3.3 It Can be Better for Data and Threads to Live Forever

A corollary to section 3.2 is memory thrashing. There is an overhead to creating, destroying, and garbage collecting data objects. Aside from the requests to the operating system for each, efficient execution does depend on how easily the OS can find the appropriately sized memory block. Garbage collection can help

maintain larger memory blocks, but the more that the objects are created and destroyed, the more work for these overhead routines. An alternative is park objects in a holding (recycling) data structure rather than destroying them. In simulations where millions of packets are created and destroyed while the working set remains in the low thousands, this recycling data structure provides significant performance improvement. Of course, the actual improvement depends on the number of objects in the working set, the number that would have been created/destroyed, and the intrinsic size of each object.

One more point on this subject is that the discussion here also applies to parallel threads created in the testbench environment. Multi-threading in the testbench is an elegant means to tightly bind interfaces to the data and methods that act on them. Like data objects, these threads are created and destroyed during the simulation run which does involve performance overhead. In very large cases, an additional interface to manage threads may be needed to manage performance as described for data objects earlier in this section.

3.4 Beware of Unforeseen Library Overhead

Libraries like the Accellera Systems Initiative Universal Verification Methodology (UVM), provide functionality to create complex verification environments quickly. From a performance perspective, one does need to keep in mind the implications of the layered approach recommended by the library. For example, it may be simply obvious to have a thread attached to the monitor/driver that is triggered on every clock to transfer data and the upper layers triggered on higher-abstraction events. For data transfers that are essentially complete on each clock transfer, this is an efficient and elegant model.

The challenge is in the scaling. For example, if the data packet is large but only streams through the interface in small pieces, the threads associated with the interface have to operate on every clock. While that may not seem like a lot of code, the issue is that the simple thread may pull a much more complex class hierarchy through inheritance. The alternative is to pull the data in burst or DMA modes in much the same way that a memory subsystem has multiple access methods using triggers other than the clock at the driver/monitor level.

So when are these novel uses of the standard library interfaces needed? That is truly an issue for the developer and integrator. Simple approaches are the best to start with because they are the easiest to maintain. However, teams with an eye toward performance will continuously benchmark their environments looking for bottlenecks associated with scaling and then seek more efficient algorithms.

3.5 Randomization

To this point, the guidelines in this paper are general object oriented software engineer practices which explains why neither reference has any SystemVerilog code. [1][2] The randomization engine in SystemVerilog introduces a divergence from other software languages but requires similar handling for performance.

The subject of building an efficient constraint set is broad, but there are simple coding elements that connect to the subject of this paper. All of the discussion in section 2 applies here.

For example, pre-calculating range values before passing them into the constraints will improve performance as shown in the example below:

```
Rand byte data[];
Rand byte data_mode;

constraint valid_data {

      data.size() ==

        foreach( data[i] ) {
         data[i] inside
         {[(data_mode-8)*16:
          (data_mode+8)*16]};

        }

}
```

Another potential performance impact is combinatorial constraints that appear sequential. These can be faster to solve sequentially, but the SystemVerilog LRM requires that the "the random values are selected to give a uniform value distribution over legal value combinations" which forces the simulator's functionality and can result is slower than expected execution.

```
rand enum bit { MEMORY, VIDEO}  target;
rand bit [63:0] addr;

constraint valid_address {

   if( pkt_type == MEMORY) addr
      inside {[0:'hffff_ffff]};

   if( pkt_type == VIDEO ) addr
     inside {['h1_0000_0000:
              'h1_0000_ffff]};

}
```

The above constraint will bias toward packet types of MEMORY because the address range is much larger.

4. Summary

Why a given SystemVerilog testbench runs slow can be a mystery without the information described in this paper. In addition to these best practices, it is recommended that engineers working with any of the dynamic languages in EDA – IEEE 1800 SystemVerilog, IEEE 1647 ***e***, and IEEE 1666 SystemC – also become familiar with their simulation vendor's profiling tools.

A profiler, like the one provided with the Cadence Incisive Enterprise Simulators, is an invaluable tool for measuring the performance of the simulation environment. Verification engineers with a hardware background may view a profiler as a tool to debug simulator performance issues. While that is one use,

verification engineers with a software background understand that the profiler is a tool to be used every few days to tune their algorithms. Simple and reusable code is by far the best and all algorithms should start that way. However, every system has physical limits and the profiler is the tool you need to optimize the performance of your verification environment to keep it within those limits.

The examples in this paper will be posted in the UVM World contributions area so that the whole community can speed their SystemVerilog simulations.

5. Acknowledgements

The authors would like to thank the IBM Cores team for input to and use of the suggestions in this paper.

6. References

[1] Coding for Performance and Avoiding Garbage Collection in Android

[2] Performance Programming Applied to C++

[3] IEEE 1800-2009 SystemVerilog LRM