DVCon-2019
San Jose, CA
Voted Best Paper
1st Place

Sunburst Design

World Class SystemVerilog & UVM Training

Yikes! Why is My SystemVerilog *Still* So Slooooow?

Cliff Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com
www.sunburst-design.com

John Rose
Cadence Design Systems, Inc.
jlrose@cadence.com
www.cadence.com

Adam Sherer
Cadence Design System, Inc.
asherer@cadence.com
www.cadence.com

ABSTRACT

This paper describes a few notable SystemVerilog coding styles and their impact on simulation performance. Benchmarks were run using the three major SystemVerilog simulation tools and those benchmarks are reported in the paper. Some of the most important coding styles discussed in this paper include UVM string processing and SystemVerilog randomization constraints. Some coding styles showed little or no impact on performance for some tools while the same coding styles showed large simulation performance impact. This paper is an update to a paper originally presented by Adam Sherer and his co-authors at DVCon in 2012. The benchmarking described in this paper is only for coding styles and not for performance differences between vendor tools.

Table of Contents

I. Introduction 4
Benchmarking Different Coding Styles 4
II. UVM is Software 5
III. SystemVerilog Semantics Support Syntax Skills 10
IV. Memory and Garbage Collection – Neither are Free 12
V. It is Best to Leave Sleeping Processes to Lie 14
VI. UVM Best Practices 17
VII. Verification Best Practices 21
VIII. Acknowledgment 25
References 25
Author & Contact Information 25

Table of Figures

Figure 1 - Example coding style performance table 4
Figure 2 - Inefficient get_num function - 3-D foreach-loop 6
Figure 3- Possibly inefficient get_num function - 2-D foreach-loop that sums size() of all queues 7
Figure 4 - Efficient get_num function utilizing separate counter variable "elements" 8
Figure 5 - 3-D foreach loop compared to keeping the elements count 8
Figure 6 - 2-D foreach loop with size() method compared to keeping the elements count 9
Figure 7 - Conditional code example with possible more efficient coding on the bottom 9
Figure 8 - Example using simple logic port declarations 10
Figure 9 - Example using wire-logic port declarations 10
Figure 10 - Benchmark results using logic ports -vs- wire-logic ports 11
Figure 11 - Examples of bit-blasting -vs- full-vector syntax 11
Figure 12 - Benchmark results using generate bit-blasting -vs- full vector operations 12
Figure 13 - Example of using looped and non-looped class object construction 13
Figure 14 - Benchmark results using looped and non-looped class object construction 13
Figure 15 - always_ff executes DPI code before testing txactive 14
Figure 16 - always_ff tests txactive before calling DPI code 14
Figure 17 - DPI code tests active signal before executing code 15
Figure 18 - Benchmark results using looped and non-looped class object construction 15
Figure 19 - Mixed static and dynamic processes with inefficient wake-up 16
Figure 20 - Mixed static and dynamic processes recoded for efficient simulation 17
Figure 21 - Benchmark results using behavioral while-loops -vs- standard FSM coding styles 17
Figure 22 - Conditional messaging in UVM 18
Figure 23 - Benchmark results using unconditional -vs- conditional UVM string processing 19
Figure 24 - Unconditionally sampling transaction and broadcasting on a UVM analysis port 20
Figure 25 - Conditionally sampling transaction and broadcasting on a UVM analysis port 20
Figure 26 - Benchmark results of transaction sampling & broadcasting with no components 20
Figure 27 - Constraint-based sorting of array elements 21
Figure 28 - post_randomize() sorting of array elements 22
Figure 29 - Benchmark results showing penalty for "clever" array-sorting constraint 22
Figure 30 - Coding assertion with local variables -vs- using other assertion capabilities 23
Figure 31 - Merged sampling -vs- separate sampling of covergroups 24
Figure 32 - Benchmark results of covergroup sampling in separate & merged always blocks 25

I. Introduction

In the 2012 edition of this topic we observed that thousands of engineers were cranking out SystemVerilog code but often found the simulation of that code ran slower than expected. We offered a number of testbench-focused recommendations for optimizing that code. Since then, millions more lines of code have been generated, simulators have become faster, but the question engineers raise is often the same with one new word: “Why is my SystemVerilog *still* so slow?”

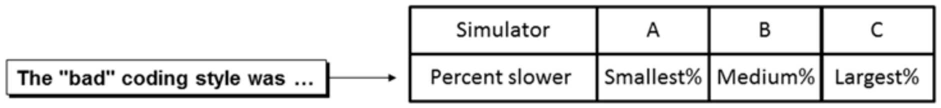
Generally speaking, all SystemVerilog simulators are faster in 2019 than they were in 2012. UVM testbench code, randomization, assertions, and design execution have all become faster as the underlying engines become more efficient. While more efficient engines are necessary, it is not sufficient to address the compute requirements of modern verification environments. That leaves engineers with only a few options they can use to improve simulation efficiency: add compute power, get more performance from their simulator vendors, construct more effective verification environments, and improve the code they feed into the first three. Ouch. Cloud and datacenter expansion are answers for compute power access but that requires financial investment. Squeezing the simulator vendor will at most yield modest gains. More effective verification environments can have larger benefits ranging from new technologies like formal analysis to refactoring algorithms. It is this last point – refactoring then can raise a “yikes” from engineers because it sounds like a lot of work for the same output. But as we examine this topic, we can discover approaches that can speed execution multiple times by training engineers to use some new best-practices.

Benchmarking Different Coding Styles

Note from Cliff Cummings: It is one thing to claim that certain coding styles are more/less efficient and another to back the claims with actual data. My Cadence co-authors have made claims about specific coding styles and I wanted to verify those claims; hence, I have run benchmarks as described below.

The recommended coding styles in this paper were all submitted by my Cadence co-authors and may have shown greater improvements in their 2012 paper, but in this paper, I found it surprising that a few of the recommended coding styles actually made the Cadence simulations slightly worse, which indicates that the Cadence simulator itself has overcome some deficiencies that might have been present in 2012. Bottom line is that the reader should not assume that all improvements were best for Cadence, and not even my Cadence co-authors know the benchmark results for the Cadence simulator.

The coding styles described by my Cadence co-authors were tested using the three major SystemVerilog simulators and are reported in this paper. My Cadence co-authors have not seen the raw benchmark data and were not given access to competitor simulators. Where I have included performance data in the paper, I have not identified which simulator gave which performance numbers. I always listed the relative performances from least improvement measured to most improvement measured, which means that each simulator was sometimes "Simulator A," "Simulator B" and "Simulator C" as shown in Figure 1.


The "bad" coding style was ...

Simulator	A	B	C
Percent slower	Smallest%	Medium%	Largest%

Figure 1 - Example coding style performance table

Each simulator had at least 3 benchmarks where they had the best benchmark times, at least 1 benchmark where they had the worst benchmark times, and at least 3 benchmarks where recommended coding styles caused the greatest improvement.

The benchmark examples were run five times on each simulator and the run times were averaged and then compared to the other styles used in the benchmark. Each benchmark directory has a CNT_file that defines the loop count used in each benchmark. Count values were anywhere from 1,000 loops for larger, memory intensive benchmarks to 100,000,000 for smaller benchmarks that would complete quickly. To test each benchmark setup, I would frequently use a starting count of 100 to make sure the simulations ran without errors. The benchmark scripts are simple (if not stupid) Makefiles, which were easy to assemble and test.

You will not be told which simulator was A, B or C for each benchmark. You will not be shown the actual simulation times for each benchmark. The intent of this paper is to alert engineers of coding styles that may or may not be inefficient for a specific simulator and coding styles that are definitely very inefficient for all simulators.

It is NOT the intent of this paper to prove that one simulator is faster or better than another, and we really do NOT want you to judge a simulator based on a single-feature, coding-style benchmark!!

If readers want to know the relative efficiency/inefficiency for specific coding styles as executed on their preferred simulator, the reader can download the benchmark files and scripts and run the simulations themselves. The benchmark code can be found on the web page http://www.sunburst-design.com/benchmark_code/DVCon2019/

II. UVM is Software

Like it or not, UVM *is* software. As such, engineers working with UVM need to become familiar with known-good software practices. As stated in the introduction, simulators are continuously optimized to improve performance. To do so, vendors typically look at coding patterns then tune the simulator to recognize these patterns for faster execution. For commonly used code, like the UVM reference library, this manifests itself in faster simulation for most projects. For project-specific coding styles or application-specific coding styles, the simulation improvements may completely miss some projects. With that said, there are inefficient coding styles that are more difficult for the simulators to recognize and need to be adjusted with software-knowledgeable coding practices.

Frequent function/task calls add overhead and may mask algorithmic order issues. Each function/task has to set-up the call, push data references and/or complete data copies to the call stack and process any specified return. When they are called in a loop, the performance cost is paid each time through the loop. For simple calls, the compiler may in-line the function/task to avoid the stack frame manipulation, but complex calls are typically not in-lined because they bloat the simulation code, which can lead to cache misses creating an entirely different performance issue. Moreover, the body of the function/task may itself contain loops and/or additional calls. These can increase the order of the algorithm especially as the application scales.

The example in Figure 2 calculates the number of elements in an MDA (Multi-Dimensional Array) of queues using a 3-dimensional **`foreach`**-loop by iterating over the array and counting elements. The algorithm is slow because it counts every element every time. It would be more efficient to use the queue's **`size()`** method as shown in Figure 3, but the most efficient approach is shown in Figure 4 where a separate count (**`elements`**) variable is maintained. While the runtime difference between the separate counter and the built-in function may be small for a small MDA, it may be hard to predict how the code will be used so the best practice would be to code for efficiency.

```
class container;
  // 2-dimensional dynamic array of queues of trans_obj handles
  trans_obj mda_q [][][$];
  int rows, cols; // default values are 0

  function void set_queues(int row, int col);
    rows = row;
    cols = col;
    mda_q = new[row];   // ILLEGAL: mda_q = new[row][col];
    // foreach loop used to initialize column sizes
    foreach(mda_q[row]) mda_q[row] = new[col];
  endfunction

  function void rand_init(int row, int col, int q_depth);
    trans_obj tr = new();
    int       q_rand_depth;

    for (int i=0; i<(row-1); i++) begin
      for (int j=0; j<(col-1); j++) begin
```

```
        // randomize q-depth for each queue
          q_rand_depth = $urandom_range(q_depth-1);
          for (int k=0; k<(q_rand_depth-1); k++) begin
          void'(tr.randomize());
          add(tr, i, j); // add to queue at [i][j]
          end
        end
      end
    endfunction

    //------------------------------------------------
    // returns total number of trans_obj handles from
    // 2-dimensional dynamic array of queued handles
    //------------------------------------------------
    function int get_num();
      foreach(mda_q[i,j,k]) get_num++;
    endfunction

    function void add (trans_obj obj, int row, int col);
      // checks for legality not shown
      mda_q[row][col].push_back(obj);
    endfunction

    // If number of trans_obj handles is not 0, remove
    // and return the first queued trans_obj handle
    function trans_obj get_first(int row, int col);
      if (mda_q[row][col].size()) begin
        get_first = mda_q[row][col].pop_front();
      end
    endfunction
  endclass
```

Figure 2 - Inefficient get_num function - 3-D foreach-loop

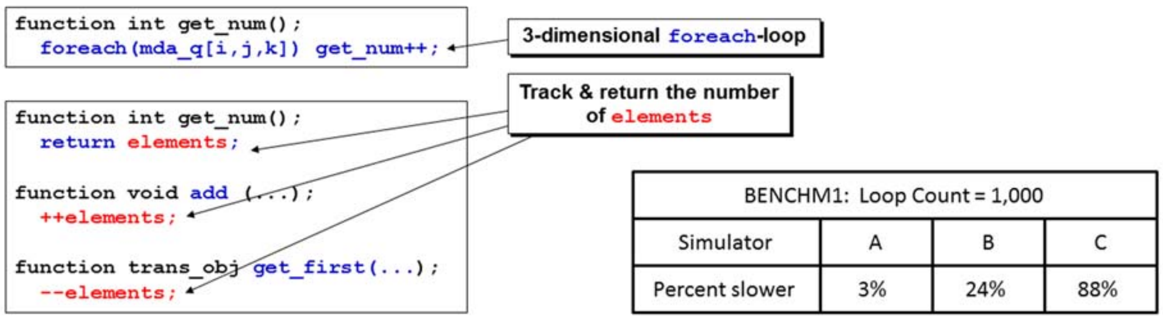
```
  class container;
    // 2-dimensional dynamic array of queues of trans_obj handles
    trans_obj mda_q [][][$];
    int rows, cols; // default values are 0


    function void set_queues(int row, int col);
      rows = row;
      cols = col;
      mda_q = new[row];   // ILLEGAL: mda_q = new[row][col];
      // foreach loop used to initialize column sizes
      foreach(mda_q[row]) mda_q[row] = new[col];

    endfunction

    function void rand_init(int row, int col, int q_depth);
      trans_obj tr = new();
      int       q_rand_depth;

      for (int i=0; i<(row-1); i++) begin
        for (int j=0; j<(col-1); j++) begin
        // randomize q-depth for each queue
          q_rand_depth = $urandom_range(q_depth-1);
          for (int k=0; k<(q_rand_depth-1); k++) begin
          void'(tr.randomize());
          add(tr, i, j); // add to queue at [i][j]
          end
        end
      end
    endfunction

    //------------------------------------------------
    // returns total number of trans_obj handles from
    // 2-dimensional dynamic array of queued handles
    //------------------------------------------------
    function int get_num();
      foreach(mda_q[i,j]) get_num += mda_q[i][j].size();
    endfunction

    function void add (trans_obj obj, int row, int col);
      // checks for legality not shown
      mda_q[row][col].push_back(obj);

    endfunction

    // If number of trans_obj handles is not 0, remove
    // and return the first queued trans_obj handle
    function trans_obj get_first(int row, int col);
      if (mda_q[row][col].size()) begin
        get_first = mda_q[row][col].pop_front();

      end
    endfunction
  endclass
```

Figure 3- Possibly inefficient get_num function - 2-D foreach-loop that sums size() of all queues

```
class container;
  // 2-dimensional dynamic array of queues of trans_obj handles
  trans_obj mda_q [][][$];
  int rows, cols; // default values are 0
  int elements;   // default values is  0

  function void set_queues(int row, int col);
    rows = row;
    cols = col;
    mda_q = new[row];   // ILLEGAL: mda_q = new[row][col];
    // foreach loop used to initialize column sizes
    foreach(mda_q[row]) mda_q[row] = new[col];
    elements = 0;
  endfunction

  function void rand_init(int row, int col, int q_depth);
    trans_obj tr = new();
    int       q_rand_depth;

    for (int i=0; i<(row-1); i++) begin
      for (int j=0; j<(col-1); j++) begin
        // randomize q-depth for each queue
        q_rand_depth = $urandom_range(q_depth-1);
        for (int k=0; k<(q_rand_depth-1); k++) begin
          void'(tr.randomize());
          add(tr, i, j); // add to queue at [i][j]
        end
      end
    end
  endfunction

  //----------------------------------------------
  // returns total number of trans_obj handles from
  // 2-dimensional dynamic array of queued handles
  //----------------------------------------------
  function int get_num();
    return elements;
  endfunction

  function void add (trans_obj obj, int row, int col);
    // checks for legality not shown
    mda_q[row][col].push_back(obj);
    ++elements;
  endfunction

  // If number of trans_obj handles is not 0, remove
  // and return the first queued trans_obj handle
  function trans_obj get_first(int row, int col);
    if(mda_q[row][col].size()) begin
      get_first = mda_q[row][col].pop_front();
      --elements;
    end
  endfunction
endclass
```

Figure 4 - Efficient get_num function utilizing separate counter variable "elements"

Benchmark measurement - How slow is the 3-D `foreach` loop compared to keeping the elements count?


3-dimensional foreach-loop
Track & return the number of elements

```
function int get_num();
  foreach(mda_q[i,j,k]) get_num++;
```

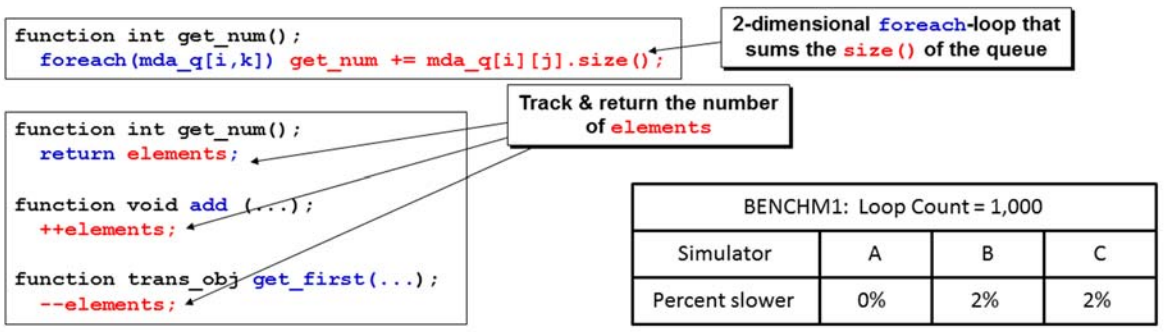
```
function int get_num();
  return elements;

function void add (...);
  ++elements;

function trans_obj get_first(...);
  --elements;
```

BENCHM1: Loop Count = 1,000			
Simulator	A	B	C
Percent slower	3%	24%	88%

Figure 5 - 3-D foreach loop compared to keeping the elements count

Depending on the simulator used, deep `foreach` nesting can have a negative performance impact.

Benchmark measurement - How slow is the 2-D `foreach` loop with `size()` method compared to keeping the elements count?



```
function int get_num();
  foreach(mda_q[i,k]) get_num += mda_q[i][j].size();
```

```
function int get_num();
  return elements;

function void add (...);
  ++elements;

function trans_obj get_first(...);
  --elements;
```

2-dimensional foreach-loop that sums the size() of the queue
Track & return the number of elements

BENCHM1: Loop Count = 1,000			
Simulator	A	B	C
Percent slower	0%	2%	2%

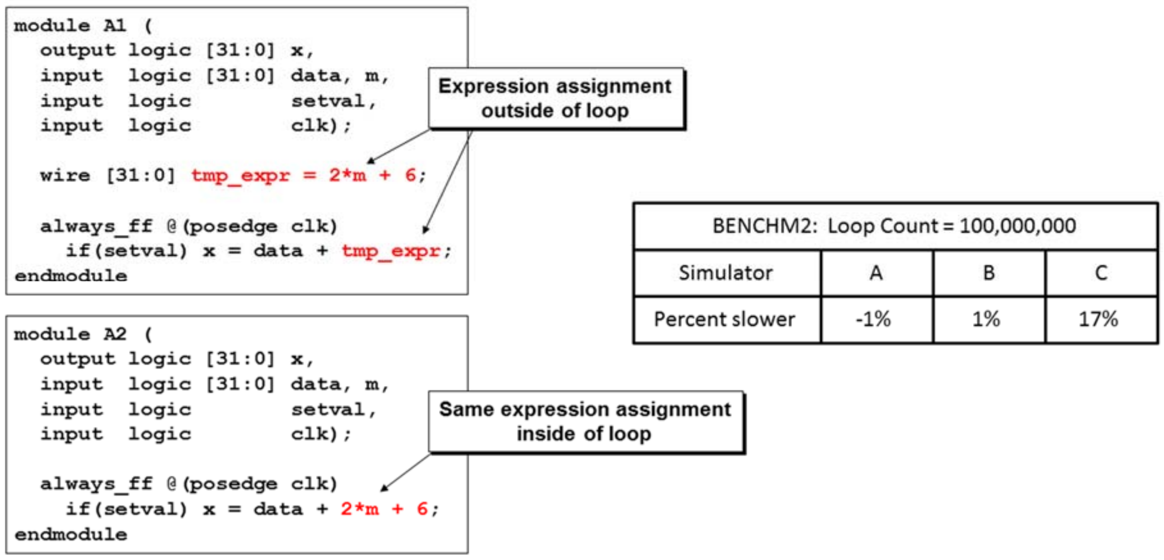
Figure 6 - 2-D foreach loop with size() method compared to keeping the elements count

This benchmark did not show a huge difference in performance as noted across the different simulators.

Benchmark Directory: BENCHM1

There are other general coding approaches, some of which were identified in the 2012 paper [1]. For example, minimize string manipulation wherever possible. Also removing loop invariants – code inside a loop that is not affected by the loop iterations – will speed simulation especially if the loop count is high. This may not be apparent to the loop author if the loop limit is variable as the limit may be much higher in scaled usage. A complement to this point is moving code that is only used in conditional statements, such as **`if`** or **`case`**, inside the conditional especially if the conditional is unlikely to be true or the calculation is only needed in one branch. If the code is in the testbench, then the compiler will likely make this adjustment for you. But if the code is in the DUT, or has any delay condition, the compiler may not be able to make the optimization because another process may update the rhs (right-hand side) of the expression.

Benchmark measurement - How slow are the temp assignments outside of the loop compared to making the assignments directly inside of the loop?



```
module A1 (
  output logic [31:0] x,
  input  logic [31:0] data, m,
  input  logic        setval,
  input  logic        clk);

  wire [31:0] tmp_expr = 2*m + 6;

  always_ff @(posedge clk)
    if(setval) x = data + tmp_expr;
endmodule
```

```
module A2 (
  output logic [31:0] x,
  input  logic [31:0] data, m,
  input  logic        setval,
  input  logic        clk);

  always_ff @(posedge clk)
    if(setval) x = data + 2*m + 6;
endmodule
```

Expression assignment outside of loop
Same expression assignment inside of loop

BENCHM2: Loop Count = 100,000,000			
Simulator	A	B	C
Percent slower	-1%	1%	17%

Figure 7 - Conditional code example with possible more efficient coding on the bottom

Not a huge difference in performance was noted but the result is simulator-dependent.

Benchmark Directory: **BENCHM2**

III. SystemVerilog Semantics Support Syntax Skills

SystemVerilog syntax knowledge is enough to get you coding, but semantics will help you do it efficiently. The semantics cover information beyond the syntax including defaults, simulator function specified by the standard, simple coding masking time-expensive execution, and more. Performance issues associated with semantics are rarely manifest in simulator profiles making them difficult to discover unless you can recognize coding issues associated with syntax semantics. This does not discount the value of profilers, but it does imply that good coding and profiling work together to achieve performance.

Let's take a look at the **`logic`** type. Introduced in SystemVerilog, the **`logic`** type can have either **`wire`** or variable storage and that storage type is determined from context by the simulator if it is not explicitly declared. This matters to simulation because wires can be collapsed to be the same object for higher simulation speed whereas variables cannot. Since the semantic for **`logic`** type is to default to variable storage in all cases except for the inputs or inouts of a design unit, you may have a properly executing simulation that mysteriously runs more slowly than you would expect. Some simulators provide some profiling and optimization support for this issue, but it is a best practice to code as explicitly as possible to avoid side-effects such as slower performance. Figure 8 shows a coding example where **`A6.y`**, **`A6.X1.y`**, and **`A6.X1.T1.y`** are distinct and where the highlighted declarations might not allow them to be collapsed to a single object. Note that **`wire`** is implicit on the **`input`** port so the declaration is not needed to assure that the **`a`** signals are collapsed together. However, it is not implicit on the **`output`** and might need to be declared in order for the simulator to collapse the **`y`** signals together.

```
module A6 (output logic [7:0] y, input logic [7:0] a, b);
  A6_1 X1 (.y(y), .a(a), .b(b));
endmodule

module A6_1 (output logic [7:0] y, input logic [7:0] a, b);
  A6_2 T1 (.y(y), .a(a), .b(b));
endmodule

module A6_2 (output logic [7:0] y, input logic [7:0] a, b);
  logic [7:0] tmp;

  assign y = tmp;

  always_comb tmp = a & b;
endmodule
```

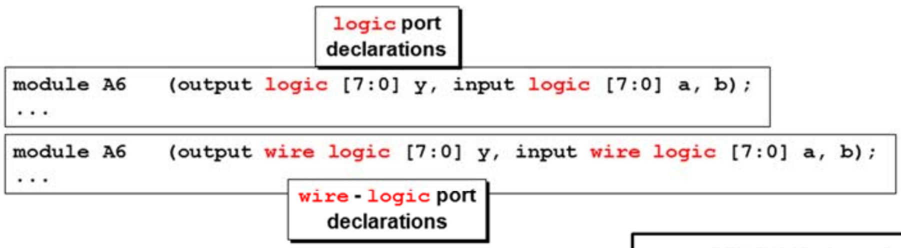
Figure 8 - Example using simple logic port declarations

It may be easier for designers to simply add **`wire`** to each port declaration to enable faster simulation speed as shown in Figure 9.

```
module A6 (output wire logic [7:0] y, input wire logic [7:0] a, b);
  A6_1 X1 (.y(y), .a(a), .b(b));
endmodule

module A6_1 (output wire logic [7:0] y, input wire logic [7:0] a, b);
  A6_2 T1 (.y(y), .a(a), .b(b));
endmodule

module A6_2 (output wire logic [7:0] y, input wire logic [7:0] a, b);
  logic [7:0] tmp;

  assign y = tmp;

  always_comb tmp = a & b;
endmodule
```

Figure 9 - Example using wire-logic port declarations

Benchmark measurement - How slow are `logic` -vs- `wire-logic` ports in simulation?



```
module A6   (output logic [7:0] y, input logic [7:0] a, b);
...
module A6   (output wire logic [7:0] y, input wire logic [7:0] a, b);
...
```

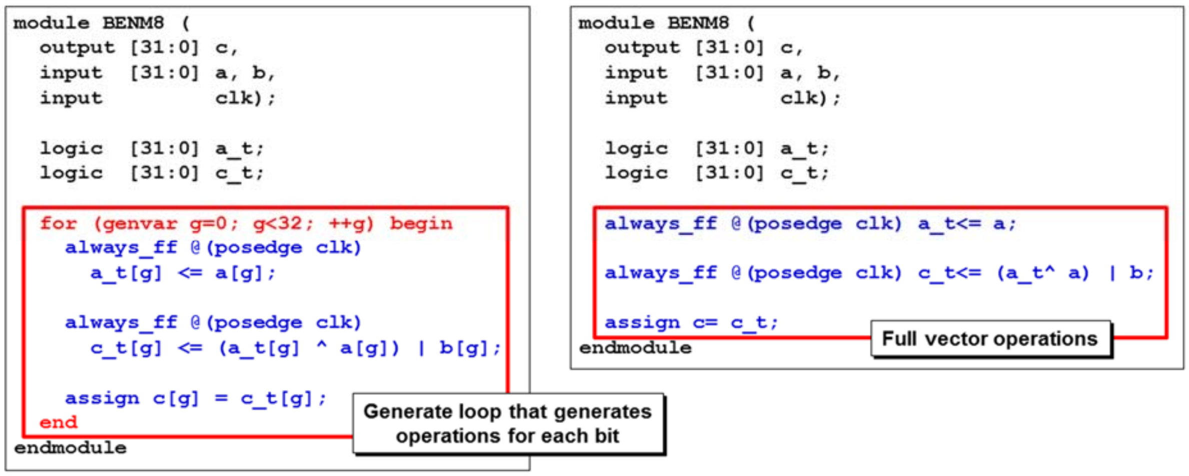
BENCHM3: Loop Count = 100,000,000			
Simulator	A	B	C
Percent slower	-3%	0%	7%

Figure 10 - Benchmark results using logic ports -vs- wire-logic ports

One simulator had a minor penalty for logic ports, but `wire` - `logic` ports are cumbersome, confusing and in general not worth the extra coding effort.

Benchmark Directory: **BENCHM3**

Another important semantic is that the simulator will typically operate faster on a full vector than individual bits. This coding style is known as bit-blasting and the syntax for it is straight forward; looping over the bits of a vector for example. In Figure 11, the inefficient example uses a `generate` statement that creates a static hierarchy. While the simulator may be able to optimize a simple example within a local process like this one, complex examples with optimization across a hierarchy are often a source of hidden performance issues.

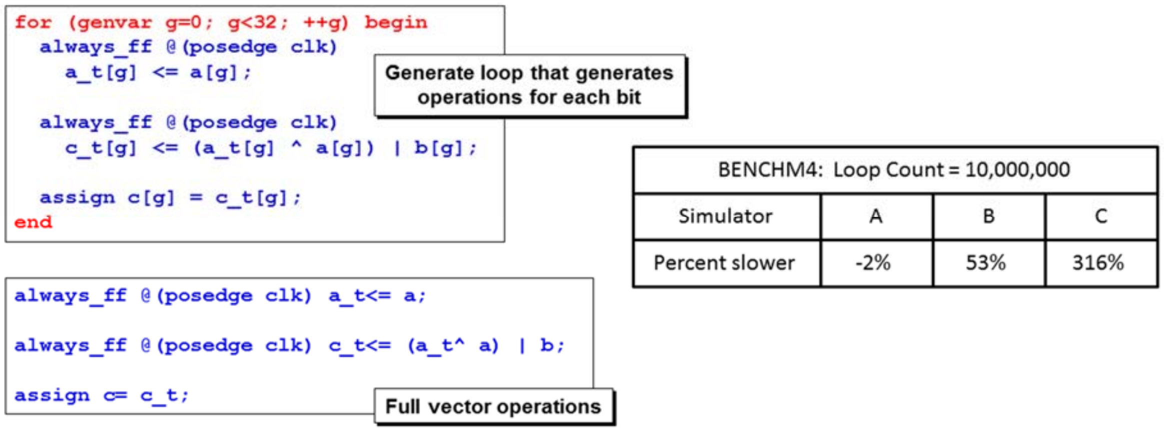


```
module BENM8 (
  output [31:0] c,
  input  [31:0] a, b,
  input         clk);

  logic  [31:0] a_t;
  logic  [31:0] c_t;

  for (genvar g=0; g<32; ++g) begin
    always_ff @(posedge clk)
      a_t[g] <= a[g];

    always_ff @(posedge clk)
      c_t[g] <= (a_t[g] ^ a[g]) | b[g];

    assign c[g] = c_t[g];
  end
endmodule
```

```
module BENM8 (
  output [31:0] c,
  input  [31:0] a, b,
  input         clk);

  logic  [31:0] a_t;
  logic  [31:0] c_t;

  always_ff @(posedge clk) a_t<= a;

  always_ff @(posedge clk) c_t<= (a_t^ a) | b;

  assign c= c_t;
endmodule
```

Figure 11 - Examples of bit-blasting -vs- full-vector syntax

Benchmark measurement - How much slower are inefficient bit-blasting operations?



```
for (genvar g=0; g<32; ++g) begin
  always_ff @(posedge clk)
    a_t[g] <= a[g];

  always_ff @(posedge clk)
    c_t[g] <= (a_t[g] ^ a[g]) | b[g];

  assign c[g] = c_t[g];
end
```

```
always_ff @(posedge clk) a_t<= a;

always_ff @(posedge clk) c_t<= (a_t^ a) | b;

assign c= c_t;
```

BENCHM4: Loop Count = 10,000,000			
Simulator	A	B	C
Percent slower	-2%	53%	316%

Figure 12 - Benchmark results using generate bit-blasting -vs- full vector operations

Inefficient bit-vector operations can have a severe performance penalty but the penalty was very simulator dependent for this example.

Benchmark Directory: **BENCHM4**

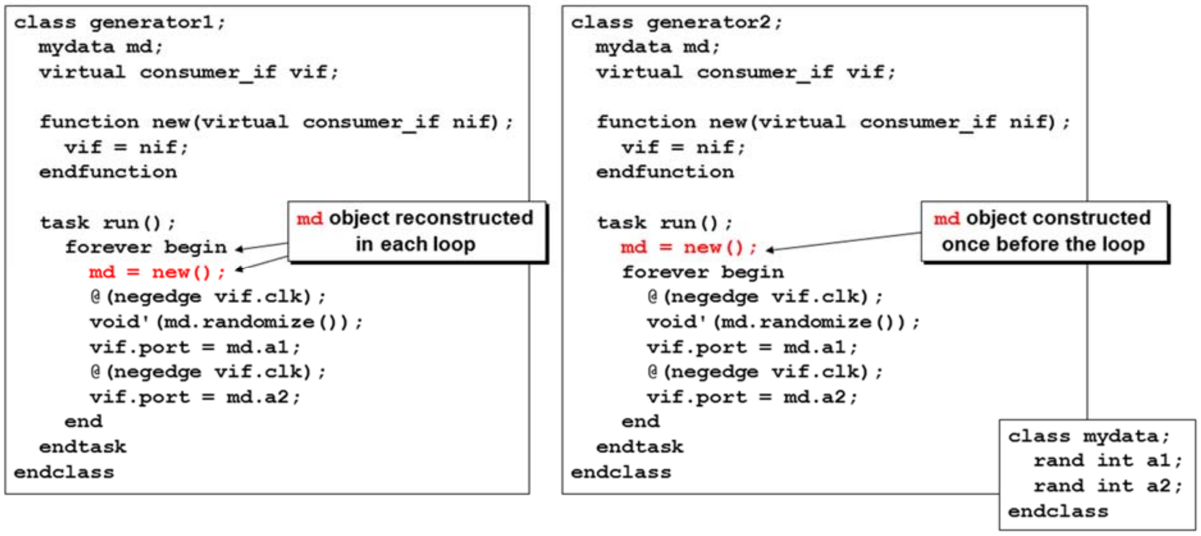
A third example of performance due to semantics is pass by reference versus pass by value. Without the general concept of a pointer and/or without the impact from algorithmic scaling, SystemVerilog coders might ignore the cost of passing by value. Engineers should see the `ref` construct in their code wherever a function only needs read access to any large data objects, such as `struct` with hundreds of fields or QDAs (Queues, Dynamic arrays, Associative arrays) with hundreds of elements, but does not write back to it. Just keep in mind that all parameters in an argument list that follow the `ref` construct will pass by reference unless you explicitly use `input`, `output` or `inout`.

The semantics of dynamic data structures (QDAs) are also sources of common performance issues that are generally true of SystemVerilog and most languages that have these types. An easy one to recognize is the use of static arrays instead of dynamic arrays wherever possible. Even if there is a small amount of variability to the length of the array, it is better to specify a slightly larger static array rather than take on the overhead of the dynamic array (memory footprint and garbage collection time). Another common performance mistake is to use dynamic arrays where a queue is better and vice versa. Since dynamic arrays are best for look-up and random insertion/deletion operations and queues are best for front or back operations with automatic resizing, the simulators have different internal representations to optimize each group of operations. Comparing access between queues and associative arrays, an arbitrary indexing for an object in a queue is O(1) but it is O(logn) for associative arrays.

IV. Memory and Garbage Collection – Neither are Free

We focused on the general semantics of dynamic types in the previous section, but the memory and garbage collection aspects of those types warrant a separate section. Inefficient memory can lead to significant cache misses, heap management overhead, and garbage collection overhead, all of which can be difficult to discover through profiling.

Copying dynamic objects, especially deep copies, should be minimized. One approach mentioned earlier is to use the `ref` construct in `function` calls, but another critical approach is to leave deep copies to the consumer of the objects wherever possible. Another approach is to reuse objects instead of continuously creating new objects when an object is required. This may be done with a single object or with a pool of objects. This is certainly a situation where the coder needs to understand how the code will be used as it is far too easy to create objects with simple `new()`methods. While the example in Figure 13 is focused on the single object example for brevity in this paper, one can envision a class with separate methods to create and destroy objects. The create method would check a dynamic object containing previously used objects and only `new()`-create one if the pool is empty. Similarly, the destroy method should push the object into the pool rather than simply dereferencing it for garbage collection. And, yes, semantics here imply that a queue is the proper dynamic object for the pool.



```
class generator1;
  mydata md;
  virtual consumer_if vif;

  function new(virtual consumer_if nif);
    vif = nif;
  endfunction

  task run();
    forever begin
      md = new();
      @(negedge vif.clk);
      void'(md.randomize());
      vif.port = md.a1;
      @(negedge vif.clk);
      vif.port = md.a2;
    end
  endtask
endclass
```

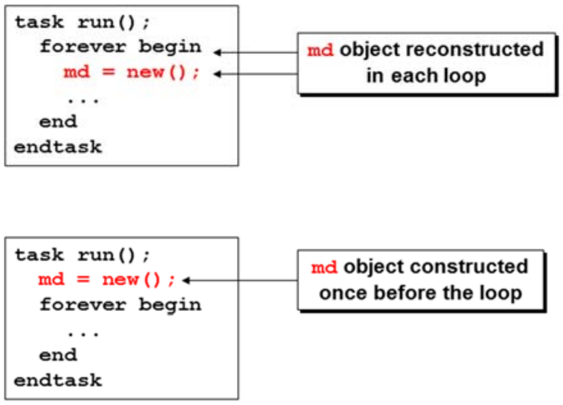
```
class generator2;
  mydata md;
  virtual consumer_if vif;

  function new(virtual consumer_if nif);
    vif = nif;
  endfunction

  task run();
    md = new();
    forever begin
      @(negedge vif.clk);
      void'(md.randomize());
      vif.port = md.a1;
      @(negedge vif.clk);
      vif.port = md.a2;
    end
  endtask
endclass
```

```
class mydata;
  rand int a1;
  rand int a2;
endclass
```

md object reconstructed in each loop
md object constructed once before the loop

Figure 13 - Example of using looped and non-looped class object construction

Benchmark measurement - How much slower is looped construction (and destruction) or objects -vs- single construction and reuse?



```
task run();
  forever begin
    md = new();
    ...
  end
endtask
```

```
task run();
  md = new();
  forever begin
    ...
  end
endtask
```

md object reconstructed in each loop
md object constructed once before the loop

BENCHM5: Loop Count = 10,000,000			
Simulator	A	B	C
Percent slower	14%	26%	82%

Figure 14 - Benchmark results using looped and non-looped class object construction

It should be obvious that class object construction and destruction has a price. Use good judgement when constructing class object.

Benchmark Directory: **BENCHM5**

Implicit heap management is another way to avoid hidden performance issues. Classes are heap objects and carry a fair amount of overhead. Wherever possible **`struct`**[s] should be used instead – either inside the class or instead of the class. For example, if the main purpose of the class is to be a container of heterogeneous data types, then a **`struct`** is a better choice. A scoreboard is a good example. It is more efficient to code a **`struct`** as data element within the class that manipulates the scoreboard instead of accessing it in a separate class because that separate class will require heap management and potentially engage garbage collection but the simple **`struct`** will not.

Putting interface-heavy functionality into the interface rather than in classes is also more simulation efficient with the added benefit of being more reusable, because the functionality is associated with the interface itself, and interfaces can be synthesizable but classes are not.

V. It is Best to Leave Sleeping Processes to Lie

SystemVerilog simulators are event driven – the more events they run at a given time point, the slower they go. The implication, to paraphrase an old adage, is to let sleeping processes lie and not wake them unnecessarily. Sometimes the wake-up may occur in unexpected circumstance.

A very common process in SystemVerilog is the **always** block with a single sensitive signal, such as the clock. This static process is highly optimized in all simulators, but side-effects from dynamic tasks or functions such as DPI (or any external) functions, virtual class tasks/functions, and virtual interface tasks/functions may disable the optimization. Some of these may be handled by a given simulator, but these side-effects can be arbitrarily complex, so the optimization cannot be maintained in all cases. In Figure 15, the DPI call was first executed outside the conditional, while in Figure 16 the DPI call is only executed if the **txactive** signal is high. The DPI code shown in Figure 17 is passed the **txactive** signal and become the **active** signal that is again tested inside of the C-code. Because of this, the simulator will need to run the **always** process on every **posedge clk** because it cannot see any side-effect coming from the DPI function. By moving the DPI call inside the conditional, the simulator might optimize the process wake up to **posedge clk** and **txactive** reducing the number of times the process executes.

```
import "DPI-C" function void dpi_tic(logic active, int count);

module BENM9A (input logic txactive, clk);
  int   counter;  // default value is 0

  initial $display("%m");

  always_ff @(posedge clk) begin
    //move DPI code into condition if it is conditional
    dpi_tic(txactive, counter);
    if (txactive)
      counter <= counter+1;
  end
endmodule
```

Figure 15 - always_ff executes DPI code before testing txactive

```
import "DPI-C" function void dpi_tic(logic active, int count);

module BENM9B (input logic txactive, clk);
  int   counter;  // default value is 0

  initial $display("%m");

  always_ff @(posedge clk)
    if (txactive) begin
      //move DPI code into condition if it is conditional
      dpi_tic(txactive, counter);
      counter <= counter+1;
    end
endmodule
```

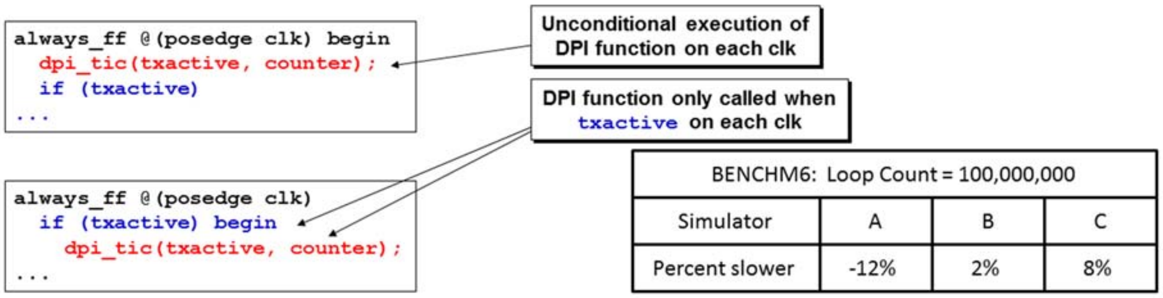
Figure 16 - always_ff tests txactive before calling DPI code

```
// DPI code
#include <stdio.h>
#include <svdpi.h>

void dpi_tic(int active, int count)
```

```
{
  if(active)
  {
    if((count%100000) == 0) {
      printf("DPI tic %d\n", count);
    }
  }
}
```

Figure 17 - DPI code tests active signal before executing code

Benchmark measurement - How much slower is unconditional execution of the DPI function in this example?



```
always_ff @(posedge clk) begin
  dpi_tic(txactive, counter);
  if (txactive)
...
```

```
always_ff @(posedge clk)
  if (txactive) begin
    dpi_tic(txactive, counter);
...
```

Unconditional execution of DPI function on each clk
DPI function only called when txactive on each clk

BENCHM6: Loop Count = 100,000,000			
Simulator	A	B	C
Percent slower	-12%	2%	8%

Figure 18 - Benchmark results using looped and non-looped class object construction

In this DPI-call example, there was not a huge difference and one simulator actually ran faster executing the DPI function while another ran slower. This might be due to DPI optimizations by one simulation vendor as opposed to actually executing the DPI code unconditionally. Even though the benchmark simulation did not show the expected performance difference, the results might be different when executing tasks, functions, virtual methods and nested combinations of the various methods.

Benchmark Directory: **BENCHM6**

It is possible to combine both static and dynamic process wake-up issues masking wake-up performance issues. For example, code can be written such that execution begins by triggering a static process, but if that leads to code triggered based on execution flow, the subsequent processes can be dynamic. Figure 19 contains an example of a state machine coded with **while** statements. This style will lead to the intended hardware behavior, but the simulator will wake-up the statement machine on every **posedge clk** regardless of the state variable. While the simulator may be able to optimize this simple coding example, the coding can be arbitrarily complex, so it cannot be optimized in all cases. However, recoding using a recommended FSM coding style, as shown in Figure 20, will lead to both proper synthesized hardware and efficient simulation.

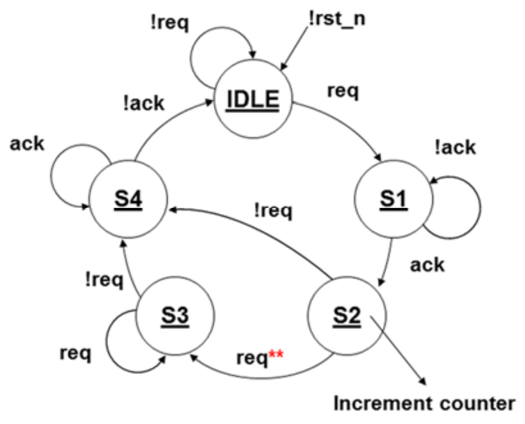
```
module BEN11A (
  output logic [31:0] counter,
  input  logic        req, ack,
  input  logic        clk, rst_n);

  enum logic [2:0] {
                    IDLE,
                    S1,
                    S2,
                    S3,
                    S4,
                    XXX} state;

  initial @(negedge rst_n) begin
    forever begin: fsm
      state   = IDLE;
      counter = '0;
      @(posedge rst_n);
      @(posedge clk); // Stay in IDLE until the next clk

      forever begin
        while (!req) @(posedge clk); // Stay in IDLE
                                  state = S1;
                     @(posedge clk); // S1
        while (!ack) @(posedge clk); // Stay in S1
                                  state = S2;
          counter = counter + 1;     // S2
                     @(posedge clk); // S2 for just one clk
        if (req) begin
                                state = S3;
                     @(posedge clk); // S3
          while (req)@(posedge clk); // Stay in S3
      end
                                state = S4;
                   @(posedge clk); // S4
        while ( ack) @(posedge clk); // Stay in S4
                                  state = IDLE;
                     @(posedge clk); // IDLE
      end
    end
  end

  always @(negedge rst_n) begin
    disable fsm;
  end
endmodule
```

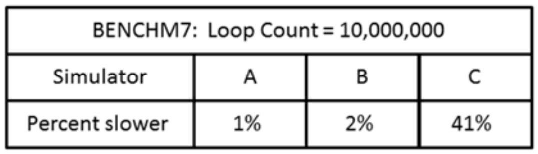
Figure 19 - Mixed static and dynamic processes with inefficient wake-up

```
module BEN11B (
  output logic [31:0] counter,
  input  logic        req, ack,
  input  logic        clk, rst_n);

  enum logic [2:0] {
                  IDLE,
                  S1,
                  S2,
                  S3,
                  S4,
                  XXX} state, next;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always_comb begin
    next = XXX;
    case(state)
      IDLE: if( req) next = S1;
            else     next = IDLE;
      S1:   if( ack) next = S2;
            else     next = S1;
      S2:   if(!req) next = S4;
            else     next = S3;
      S3:   if(!req) next = S4;
            else     next = S3;
      S4:   if(!ack) next = IDLE;
            else     next = S4;
      default:       next = XXX;
    endcase
  end

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) counter <= '0;
    else begin
      case (next)
        IDLE, S1, S3, S4: ; // No counter change
        S2     : counter <= counter + 1;
        default: counter <= XXX;
      endcase
    end
endmodule
```

Figure 20 - Mixed static and dynamic processes recoded for efficient simulation

Benchmark measurement - Surprisingly, the standard FSM coding style was SLOWER that the `while`-loop style.





BENCHM7: Loop Count = 10,000,000			
Simulator	A	B	C
Percent slower	1%	2%	41%

Figure 21 - Benchmark results using behavioral while-loops -vs- standard FSM coding styles

Despite the fact that this benchmark showed that the standard FSM coding style might be slower than the while - loop behavioral implementation, correct FSM coding styles are more important that this benchmarked simulation performance.

Benchmark Directory: **BENCHM7**

VI. UVM Best Practices

Not surprising, all of the principles discussed so far apply to UVM. Wherever possible, deep-copy, pass by value, process wake-up associated with UVM should be minimized.

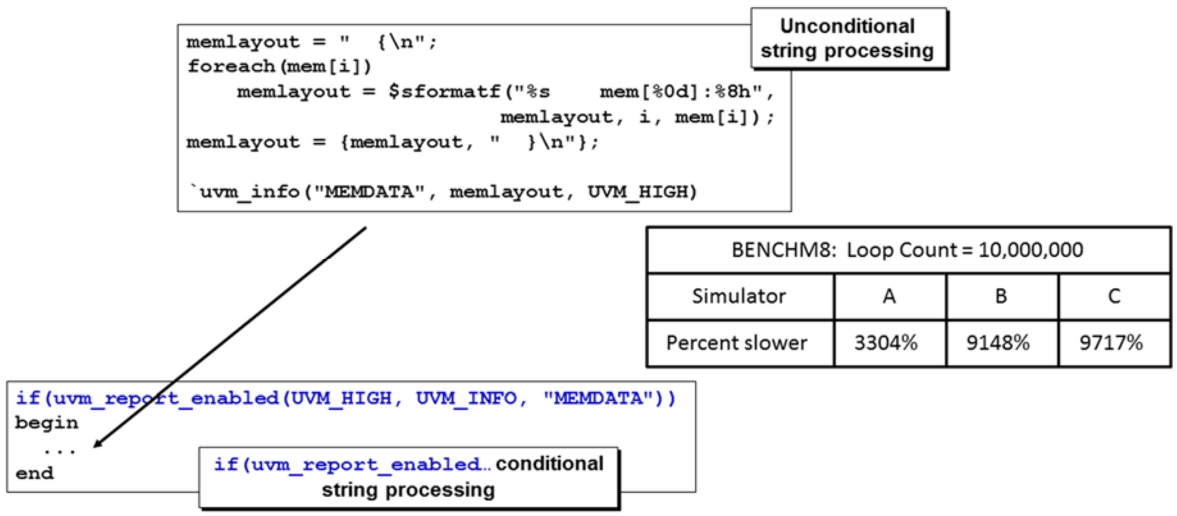
A common approach in UVM is to code message writing to interesting parts of the verification environment. A simple optimization is to guard the messaging using the `uvm_report_enabled()` function. In Figure 22, the messaging is triggered if the verbosity level is set at or above `UVM_HIGH`. And if the messaging needs to be written frequently, the UVM tree printer or even the line printer should be used to get meaningful information while keeping the messaging overhead to a minimum.

```
`include "CNT_file"
class mem_print_test;
  int mem [47:0];
```

```
        virtual clk_if vif;

        function new(virtual clk_if nif);
          vif = nif;
        endfunction

        task run();
          set_data(32'h0000_0000);
          repeat (`CNT/10) get_data();
          set_data(32'h1111_1111);
          repeat (`CNT/10) get_data();
          set_data(32'h2222_2222);
          repeat (`CNT/10) get_data();
          set_data(32'h3333_3333);
          repeat (`CNT/10) get_data();
          set_data(32'h4444_4444);
          repeat (`CNT/10) get_data();
          set_data(32'h5555_5555);
          repeat (`CNT/10) get_data();
          set_data(32'h6666_6666);
          repeat (`CNT/10) get_data();
          set_data(32'h7777_7777);
          repeat (`CNT/10) get_data();
          set_data(32'h8888_8888);
          repeat (`CNT/10) get_data();
          set_data(32'h9999_9999);
          repeat (`CNT/10) get_data();
        endtask

        function void set_data(int data, bit random='0);
          if (random)
            for (int i=0; i<48; i++) mem[i] = $urandom;
          else
            mem = '{default:data};
        endfunction

        function void get_data();
          string memlayout;
          `ifdef FAST
          // Only do expensive string processing for >= UVM_HIGH verbosity
          if(uvm_report_enabled(UVM_HIGH, UVM_INFO, "MEMDATA")) begin
          `endif
            // Format the memory layout into a string
            memlayout = "  {\n";
            foreach(mem[i])
                memlayout = $sformatf("%s    mem[%0d]:%8h",
                                       memlayout, i, mem[i]);
            memlayout = {memlayout, "  }\n"};
          `ifdef FAST
          end
          `endif
          `uvm_info("MEMDATA", memlayout, UVM_HIGH)
         endfunction
      endclass
```

Figure 22 - Conditional messaging in UVM

Benchmark measurement - How much slower is this UVM simulation using unconditional string processing?



```
memlayout = "  {\n";
foreach(mem[i])
    memlayout = $sformatf("%s    mem[%0d]:%8h",
                          memlayout, i, mem[i]);
memlayout = {memlayout, "  }\n"};

`uvm_info("MEMDATA", memlayout, UVM_HIGH)
```

Unconditional string processing

BENCHM8: Loop Count = 10,000,000			
Simulator	A	B	C
Percent slower	3304%	9148%	9717%

```
if(uvm_report_enabled(UVM_HIGH, UVM_INFO, "MEMDATA"))
begin
  ...
end
```

if(uvm_report_enabled.. conditional string processing

Figure 23 - Benchmark results using unconditional -vs- conditional UVM string processing

The unconditional array string processing even when the processed string was not printed was huge, exacting a penalty of 3,000-10,000 time slower than conditional string processing. Care should be taken as it relates to UVM string processing.

Benchmark Directory: **BENCHM8**

Another source on unnecessary execution is associated with TLM analysis ports. Since these are often used as a callback mechanism, monitors, collectors, or other components package data and then write the data to all of the consumers that are attached to the analysis port. As such, the monitor will often do all the transaction sampling and broadcasting, but there is no requirement that an analysis port be connected to any other components so the sampled transaction may not be used. In fact, the connection can be environment dependent.

Figure 24 shows the unconditional monitoring and broadcasting of transactions on the analysis port, while Figure 25 uses a conditional test to see if anything is connected to the analysis port before sampling and broadcasting a transaction.

```
class tb_monitor extends uvm_monitor;
  `uvm_component_utils(tb_monitor)

  virtual dut_if vif;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  uvm_analysis_port #(trans1) ap = new("ap", this);

  // Unconditionally broadcast UVM analysis port transactions
  task run_phase(uvm_phase phase);
    forever collect();
  endtask

  task collect();
    trans1 tr = trans1::type_id::create("tr");
    get_txn_from_interface(tr);
    ap.write(tr);
  endtask
```

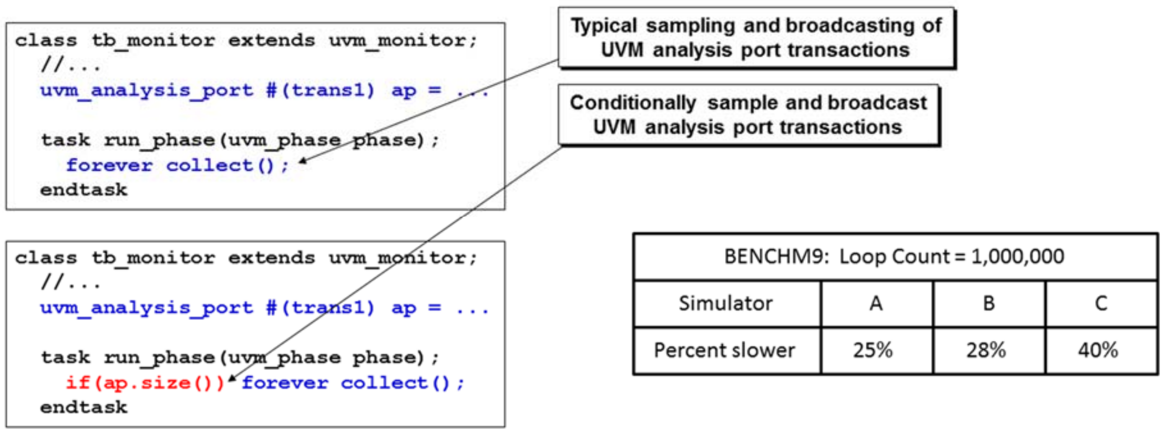
```
    task get_txn_from_interface(trans1 tr);
      tr.data1 = vif.data1;
      tr.data2 = vif.data2;
      @vif.cb1;
    endtask
  endclass
```

Figure 24 - Unconditionally sampling transaction and broadcasting on a UVM analysis port

```
  class tb_monitor extends uvm_monitor;
    `uvm_component_utils(tb_monitor)

    virtual dut_if vif;

    function new(string name, uvm_component parent);
      super.new(name, parent);
    endfunction

    uvm_analysis_port #(trans1) ap = new("ap", this);

    // Conditionally broadcast UVM analysis port transactions
    task run_phase(uvm_phase phase);
      if(ap.size()) forever collect();
    endtask

    task collect();
      trans1 tr = trans1::type_id::create("tr");
      get_txn_from_interface(tr);
      ap.write(tr);
    endtask

    task get_txn_from_interface(trans1 tr);
      tr.data1 = vif.data1;
      tr.data2 = vif.data2;
      @vif.cb1;
    endtask
  endclass
```

Figure 25 - Conditionally sampling transaction and broadcasting on a UVM analysis port

Benchmark measurement - How much slower is unconditional UVM transaction sampling analysis port broadcasting -vs- NOT sampling and broadcasting if the analysis port is unconnected to other components?


Typical sampling and broadcasting of UVM analysis port transactions
Conditionally sample and broadcast UVM analysis port transactions

```
class tb_monitor extends uvm_monitor;
  //...
  uvm_analysis_port #(trans1) ap = ...

  task run_phase(uvm_phase phase);
    forever collect();
  endtask
```

```
class tb_monitor extends uvm_monitor;
  //...
  uvm_analysis_port #(trans1) ap = ...

  task run_phase(uvm_phase phase);
    if(ap.size()) forever collect();
  endtask
```

BENCHM9: Loop Count = 1,000,000			
Simulator	A	B	C
Percent slower	25%	28%	40%

Figure 26 - Benchmark results of transaction sampling & broadcasting with no components

Turning off unused analysis port path sampling and broadcasting can significantly improve simulation performance. Verification IP (VIP) developers should take note and turn off transaction sampling and broadcasting if the VIP agent is not connected to any other components.

Benchmark Directory: **BENCHM9**

There are some additional best practices that can help reduce the risk of unexpected overhead. If objections need to be frequently raised and lowered for objects deep in the hierarchy, the overhead can become prohibitive, so the use of global objection handling may be warranted. Configurations can also lead to unexpected overhead if the complexity of the types causes the expansion of the fields to hundreds or thousands of elements. In this case a configuration container should be used. A final point of guidance is to never use wildcards for field names unless absolutely necessary because of the unanticipated algorithmic order issues. A simple name lookup is of log(n) order, but the wildcards invoke the regular expression execution causing additional overhead.

VII. Verification Best Practices

This section probably warrants a paper unto itself. As such, we will focus on a few critical recommendations associated with randomization, assertions, and coverage execution. The authors agree that this just scratches the surface on this topic, but these simple recommendations can be broadly applied so they fit within the context of this paper.

SystemVerilog provides multiple ways to randomize variables so it's important to understand the advantages of each to maximize performance. **`$urandom`** adds thread stability to the older Verilog standard $random and remains the fastest way to do randomization of single, independent variables which are randomized often. More complex constraint-based randomization is often used in UVM environments and is subject to a lot of optimization work in each simulator. However, there are practices that can help each engine solve constraints faster. Using solve order to break up or simplify constraint solving and using **`pre_randomize/post_randomize`** for sequential solving can speed-up simulation but you do need to be careful to avoid creating invalid solutions. Echoing recommendations made earlier in this paper regarding arrays, coders should be careful to avoid coupling such variables where possible. Figures 15 and 16 bring some of these concepts together. In Figure 27, the loop sets up a constraint on each array element based on its neighbor resulting in a list of 16-256 (randomized) integers with 32-bit variables that have to be solved simultaneously. Modifying the code to use **`post_randomize()`** and an array **`sort()`** method as shown in Figure 28 can improve runtime performance up to 1000x.

```
class txn15;
  rand int          addr;
  rand logic [15:0] payload[$];
  rand bit    [2:0] del;

  constraint size_ct { payload.size() inside { [16:256]}; }

  constraint sort_ct {
    foreach (payload[i]) {
      // i must be greater than 0
      if(i) payload[i] >= payload[i-1];
    }
  }

  function void show_payload();
    `uvm_info("FIG15", $sformatf("payload=%p", payload), UVM_MEDIUM)
  endfunction
endclass
```

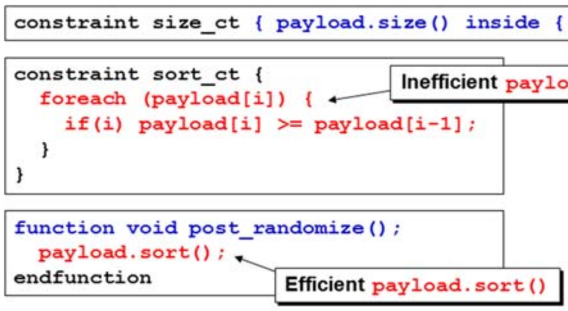
Figure 27 - Constraint-based sorting of array elements

```
class txn16;
  rand int          addr;
```

```
    rand logic [15:0] payload[$];
    rand bit    [2:0] del;

    constraint size_ct { payload.size() inside { [16:256]}; }

    function void post_randomize();
      payload.sort();
    endfunction

    function void show_payload();
      `uvm_info("FIG16", $sformatf("payload=%p", payload), UVM_MEDIUM)
    endfunction
  endclass
```

Figure 28 - post_randomize() sorting of array elements

Benchmark measurement - How much slower is constraint-based array sorting -vs- **`post_randomize()`** use of the array **`sort()`** method?



```
constraint size_ct { payload.size() inside { [16:256]}; }

constraint sort_ct {
  foreach (payload[i]) {
    if(i) payload[i] >= payload[i-1];
  }
}

function void post_randomize();
  payload.sort();
endfunction
```

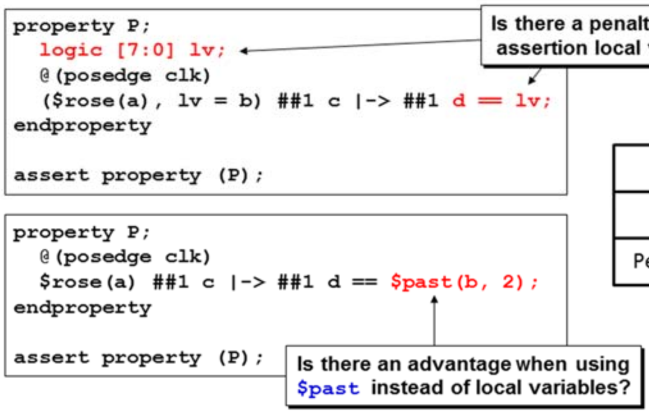
BENCHM10: Loop Count = 100,000			
Simulator	A	B	C
Percent slower	1535%	3621%	6553%

Figure 29 - Benchmark results showing penalty for "clever" array-sorting constraint

As shown in Figure 29, randomization constraints techniques can hinder simulation performance the 1,000's of percent. Be aware of the penalties you might incur from "clever" constraint techniques.

Benchmark Directory: **BENCHM10**

Assertions are also commonly used in UVM environments and subject to a lot of optimization in simulators. As with randomization, a set of best practices can help boost performance. In general, we want to balance the well-known value of assertions with efficient simulation. Minimizing the number of attempts by configuring the enabling condition to trigger only on the first cycle of the enabling condition, using single-cycle assertions wherever possible, and using single-clock assertions – even if that means splitting the assertion into two separate assertions – all result in improved performance. While local variables may be needed to manipulate data inside sequences and properties, they add overhead during simulation.



```
property P;
  logic [7:0] lv;
  @(posedge clk)
  ($rose(a), lv = b) ##1 c |-> ##1 d == lv;
endproperty

assert property (P);
```

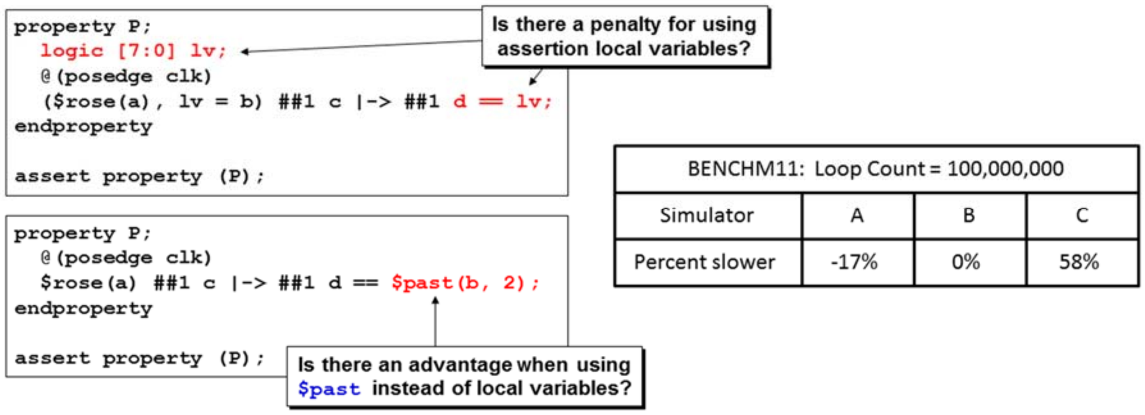
```
property P;
  @(posedge clk)
  $rose(a) ##1 c |-> ##1 d == $past(b, 2);
endproperty

assert property (P);
```

Is there a penalty for using assertion local variables?
Is there an advantage when using $past instead of local variables?

BENCHM11: Loop Count = 100,000,000			
Simulator	A	B	C
Percent slower	-17%	0%	58%

Figure 30The example at the top of Figure 30 shows an assertion coded using a local variable such that when **`a`** goes high, a local value of **`b`** is saved and then used at the end of the assertion. The example at the bottom of Figure 30 simply uses the **`$past()`** assertion capability to state that the **`b`**-value from two cycles ago should be checked at the end of the assertion.

Benchmark measurement - How much slower is using local variables -vs- other assertion techniques?



```
property P;
  logic [7:0] lv;
  @(posedge clk)
  ($rose(a), lv = b) ##1 c |-> ##1 d == lv;
endproperty

assert property (P);
```

```
property P;
  @(posedge clk)
  $rose(a) ##1 c |-> ##1 d == $past(b, 2);
endproperty

assert property (P);
```

Is there a penalty for using assertion local variables?
Is there an advantage when using $past instead of local variables?

BENCHM11: Loop Count = 100,000,000			
Simulator	A	B	C
Percent slower	-17%	0%	58%

Figure 30 - Coding assertion with local variables -vs- using other assertion capabilities

The use of local variables seems to be very-vendor dependent. Some vendors may have optimized local variable while others may have optimized other assertion methods. Ask your vendor for recommendations regarding the use of local variables with their SystemVerilog simulator.

Benchmark Directory: **BENCHM11**

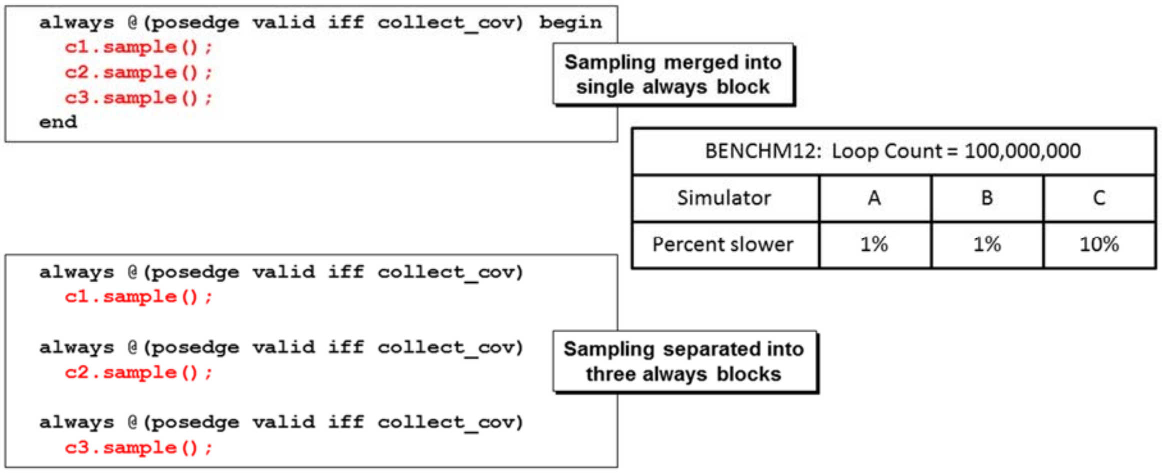
Coverage is a third common element of verification environments. When setting up the coverage bins, choose vectored or auto bins to maximize performance as scalar and fixed-size bins create additional simulation overhead and are more difficult to debug. Per the mantra of this paper, fewer coverage events will deliver faster simulation. This starts by sampling coverage using a specific event trigger rather than a generic event such as a system clock. Coverage sampling events can be further reduced by having **`covergroup`**[s] share common expressions. A third method to reduce sampling events is to merge sample process that use the same event as shown in Figure 18.

```
module top;
  bit [31:0] addr1, addr2;
  bit [15:0] data;
  bit        collect_cov;
  bit        valid;   // triggering event

  logic clk;
  `include "CNT_file"

  covergroup cg1;
    adr1: coverpoint addr2 {
            bins a[4] = {[0:$]};
          }
  endgroup

  covergroup cg2;
    adr2: coverpoint addr2 {
            bins b[4] = {[0:$]};
          }
  endgroup
```

```
    covergroup cg3;
      dat1: coverpoint data {
              bins d[4] = {[0:$]};
            }
    endgroup

    clkgen i1 (.clk(clk));

    cg1 c1 = new();
    cg2 c2 = new();
    cg3 c3 = new();

    initial begin
      forever @(posedge clk) begin
        addr1 = $urandom;
        addr2 = $urandom;
        data  = $urandom;
        {collect_cov, valid}  = $urandom_range(0,3);
      end
    end

    initial begin
      repeat(`CNT) @(posedge clk);
      $finish;
    end

    `ifdef MERGED

    // Sampling merged to a single event
    always @(posedge valid iff collect_cov) begin
      c1.sample();
      c2.sample();
      c3.sample();
    end

    `else

    always @(posedge valid iff collect_cov)
      c1.sample();

    always @(posedge valid iff collect_cov)
      c2.sample();

    always @(posedge valid iff collect_cov)
      c3.sample();

    `endif
  endmodule
```

Figure 31 - Merged sampling -vs- separate sampling of covergroups

Benchmark measurement - How much slower is sampling of **`covergroup`**[s] in separate **`always`** blocks -vs- sampling in the same **`always`** block?



```
always @(posedge valid iff collect_cov) begin
  c1.sample();
  c2.sample();
  c3.sample();
end
```

Sampling merged into single always block

BENCHM12: Loop Count = 100,000,000			
Simulator	A	B	C
Percent slower	1%	1%	10%

```
always @(posedge valid iff collect_cov)
  c1.sample();

always @(posedge valid iff collect_cov)
  c2.sample();

always @(posedge valid iff collect_cov)
  c3.sample();
```

Sampling separated into three always blocks

Figure 32 - Benchmark results of covergroup sampling in separate & merged always blocks

The sampling of **`covergroup`** [s] in separate -vs- merged **`always`** blocks did not show a large performance difference and the results, although small, were simulation vendor dependent.

Benchmark Directory: **BENCHM12**

VIII. Acknowledgment

UVM/SystemVerilog environments have grown in size and complexity since we looked at SystemVerilog performance in 2012. With that growth, the cost of slow execution has also grown. This paper provided code examples and guidelines to make every simulation faster and reduce those "yikes" moments verification engineers have when they run their regressions.

References

[1] "Yikes! Why is my SystemVerilog so Slooooow?" Frank Kampf, Justin Sprague, Adam Sherer, DVCon U.S. Proceedings 2012.

[2] *IEEE Std 1800-2012, IEEE Standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language.* by IEEE, 3 Park Avenue, New York, NY 10016-5997, USA.

[3] *Assertion Writing Guide, Product Version 14.2, January 2015, Chapter 8 - "Maximizing Assertion Performance"*, by Cadence Design Systems, Inc., 2655 Seely Avenue, San Jose, CA 65134, USA.

[4] "App Note Spotlight: Streamline Your SystemVerilog Code, Part I," by John Rose. Blog post by Tyler Sherer 19 March 2018. Cadence Functional Verification Blogs. Retrieve from http://community.cadence.com/cadence_blogs_8/b/fv/posts/app-note-spotlight-streamline-your-systemverilog-code-part-i.

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 37 years of ASIC, FPGA and system design experience and 27 years of SystemVerilog, synthesis and methodology training experience.
Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past 16 years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.
Mr. Cummings participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee from 1994-2012, and has presented more than 50 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.
Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.
Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.
Email address: cliffc@sunburst-design.com

John Rose, Product Engineering Architect, Cadence Design Systems, Inc., has worked in the EDA industry for the past 24 years and prior to that was an ASIC designer using Verilog HDL with Synthesis. Mr. Rose spent three years as a design and verification consultant designing a wide range of chips from image processors to encryption/ decryption chips for peripheral controllers.
Mr. Rose was involved in the development of a C based verification library, TestBuilder, that morphed into a SystemC based verification library, SCV. Mr. Rose was also a key contributor in the development in the verification methodology libraries, URM (Cadence proprietary library), OVM (Cadence and Mentor collaboration) and UVM (Accellera).
Mr. Rose holds a BSEE from The University of Kansas.
Email address: jlrose@cadence.com

Adam Sherer
Adam Sherer drives verification software and hardware sales in the Eastern North America region for Cadence Design Systems. Mr. Sherer has 27 years of experience in verification and software engineering that also has included roles in product management, applications engineering, standards development, and R&D.
Mr. Sherer received his MS EE from the University of Rochester, with research published in the IEEE Transactions on CAD. His BS EE and BA CS were received from SUNY Buffalo. Mr. Sherer also holds a 2017 patent in verification technology.
Email address: asherer@cadence.com

Last Updated: April 2019