Unconstrained UVM SystemVerilog Performance

Wes Queen, Verification Manager
IBM
RTP, NC

Justin Sprague, Principal Applications Engineer
Cadence Design Systems
San Jose, CA

***Abstract* — It's been a decade – we all know how to write SystemVerilog constraints. Just choose the variables to randomize, write the constraint expressions, and let the simulator do its magic. If it runs slow, just call the simulator vendor and ask them to fix it. What more is there? By following the guidelines presented here we can get unconstrained performance from the simulators running our UVM SystemVerilog constraints**

Keywords— SystemVerilog, Randomization, Constraints, UVM, Performance

I. Introduction

Randomization is a fundamental part of UVM SystemVerilog verification environments. Inside these environments, randomization is often used in configuration code as well as during sequence and item generation. As these environments grow in scope and complexity, it is becoming common to see tens of thousands to hundreds of thousands of randomize calls in the course of a single simulation - if not more. As a result, the performance of those randomize calls is becoming an important focus area for verification environment developers.

A. Set some reasonable goals for randomization times

A fundamental step in keeping randomization performance reasonable is to establish some goals for performance of each randomize call. An important step to establishing these goals is to understand the roles each UVM class type takes on in your verification environment.

Configuration classes capture the high level knobs that describe how your design and the UVCs in your environment operate. These classes are typically randomized only once or twice per simulation. As a result, it is reasonable to allow the randomization time for these classes to be measured in times of seconds or more. A randomization time of a few seconds or perhaps even ten or fifteen seconds is minor when compared to the overall simulation run time. With this longer time available for randomization, creating complex constraints with large numbers of variables can be very effective here.

Item classes, such as those classes derived from uvm_transaction or uvm_sequence_item, are used to represent the types of objects we send into or through a device - such as a packet. Similarly these classes can be used to represent the basic bus transactions needed to stimulate a device – such as a PCI Express bus read or bus write. Typically, item classes are generated and randomized very frequently. For these classes it is important to keep the randomization time as small as possible. A single simulation may create anywhere from hundreds to millions of items. As a result, a randomization time of a single second can result in a very long total simulation time. To keep these times low, we want to minimize the work the constraint solver does to the minimal number of constraints and variables needed.

B. Understand the basic randomization process

When creating classes with random variables and constraints, it's important to understand how the randomization process works in SystemVerilog. For a complete understanding, you'll want to spend a few hours reading the chapter in the SystemVerilog Language Reference Manual (LRM) on Constrained Random Value Generation.

For the purpose of this paper, we can shorten that learning time by highlighting a few basic rules:

- All random variables are solved at the same time.
- All constraints are bi-directional.
- If a solution is possible, the randomize call should find it.
- Variables are solved in a way that provides for an even distribution over the entire constraint space.

Keeping these rules in mind when coding can help guide you to simpler classes with easier to solve constraints.

II. Coding techniques for simulation speed

Once reasonable expectations have been established for the different types of objects in the UVM environment, we can begin to look at some guidelines to speed up the different randomize calls.

Before we begin discussing each, it's important to note two things. First, the performance impact of each approach will vary based on the specific structure of your classes and the constraints within them. Second, this paper demonstrates approaches that have been useful with classes containing complex constraints or those randomized many times. For the purposes of illustration, the examples contained here use small classes with a few constraints. These techniques are most applicable when applied to classes with many constraints and variables or to those classes randomized many times.

A. Use solve…before to simplify complex relationships

Recall that constraint expressions are bi-directional. Consider the following code:

```
class meal extends uvm_sequence_item;
  rand day_t weekday;
  rand meal_t lunch;
  constraint example_c {
    (weekday == TUESDAY) ->
                 lunch != PIZZA);
    If (weekday== WEDNESDAY )
                    (lunch == SOUP);
  }
endclass
```

Reading the above code, it would be intuitive to expect that the value of weekday is chosen prior to the value of lunch. However, that's not the case. In SystemVerilog, the values of all variables are chosen simultaneously. This means that the randomization engine is selecting a value for lunch at the same time as it is selecting a value for the day of the week. Solving for all values simultaneously allows the engine to ensure a uniform distribution across all the variables used in the class.

When we write our constraints, we often anticipate an ordering, such as picking the day of the week prior to the picking the selection for lunch. In these cases, solving both variables simultaneously and ensuring a uniform distribution across them is not necessary and can result in unneeded complexity.

Fortunately, we can tell the randomization engine that we want some variables chosen before others. This is done with the solve…before syntax. For example, we can add the constraint:

```
solve weekday before lunch;
```

In the simple example from the above code, this would not make a significant performance difference. However, consider more complex cases such as:

```
constraint mode_c {
   mode inside {SINGLE, DUAL, /*and more*/};
   if (mode == SINGLE) {
     data_width inside {2,4,6};
     // lots of other constraints
   }
   if( mode == DUAL) {
     data_width inside {2,3,4,5,6};
     // lots of other constraints
   }
   if( data_width == 3 ) {
     min_value = 5;
     max_value = 11;
   }
}
```

In the above code, we have a case where solving mode before data_width and solving data_width before min_value or max_value can indeed reduce the complexity. Adding solve…before constraints can guide such an ordering. Those constraints would look like:

```
constraint ordering_c {
   solve mode before data_width;
   solve data_width before
         min_value, max_value;
}
```

Adding these two solve…before constraints introduces an explicit ordering when solving the variables. This ordering is what allows for the speedup.

Before applying solve…before, it is important to be aware that this explicit ordering though does not always help. There are two general cases to watch for. First, this new ordering will change the distribution of values seen in the results. You may find that adding some dist constraints will help resolve this. Second, solve…before constraints can sometimes result in slower randomization times due to the impact of the ordering change on other variables solved for during the randomization step.

A general approach for adding solve…before constraints is: First, look for ordering patterns in your constraints where this approach may help. Second, add a solve..before that pattern. At this point run the simulation and measure the results. If a performance gain is observed, examine the distributions on the variables to ensure they fit the general requirements. If the distribution is not ideal, then apply one or more dist constraints.

B. Understand the impact of arrays with foreach constraints

Foreach constraints are a useful way to apply repetitive constraint expressions to each item in an array. When using foreach, it is important to remember that its usage effectively results in a new constraint for each item in the array. Should the array have ten entries, this is like adding ten new constraints for each line in the foreach. If the array has 100 entries, it is like adding 100 new constraints. This can quickly result in adding more random variables and more constraints to the problem the randomization engine needs to solve. Consider the following code:

```
class packet extends uvm_sequence_item;
  rand bit [31:0] data[];
  constraint data_values {
    data.size() == 100;
    foreach( data[i] ) {
      if(i != 0) {
        data[i] > data[i-1];
      }
    }
  }
endclass
```

Here we see that there is a data array, and that it has 100 entries. Each entry in the array must be greater than the value of the prior item in the array. Not only does this code introduce more variables and more constraints, but it further complicates the problem by creating a relationship between every variable in the array. This pattern results in 100 random variables that must all solved for simultaneously. By creating this relationship, it further slows randomization.

A similar use model for foreach that can lead to performance problems is seen with nested foreach constraints used to determine uniqueness:

```
class config extends uvm_sequence_item;
  rand int table[];
  constraint table_values {
    table.size() == 50;
    foreach( table[i] ) {
      foreach( table[j] ) {
        if( i != j )
          table[i] != table[j];
      }
    }
  }
endclass
```

This code uses nested foreach constraints to force every entry in the table to be different from every other entry – in other words, to create a table of unique entries. This ensures that the randomization engine needs to be solve for all that values in the array simultaneously.

The best way to speed foreach constraints is to make sure that there is no linkage between the arrays entries inside the foreach calls. If that linkage exists, and is unavoidable, consider remodeling the code so that it can be solved procedurally inside a post_randomize function. This will be discussed in the next section.

C. Use pre_randomize() and post_randomize() to assign values procedurally

The pre_randomize() and post_randomize() functions provide the ability to execute code procedurally either before or after the randomization engine runs. These functions can be a simpler alternative to complex constraint code. Consider the code:

```
class item extends uvm_sequence_item;
  rand int data[];
  int var_a, var_b, var_c;
  constraint value_c {
    value.size() == 50;
    foreach( value[i] ) {
      value[i] != (var_a*var_b)/var_c;
    }
  }
endclass
```

In the above code we can use the pre_randomize() function to eliminate the extra calculations of (var_a*var_b)/var_c from the constraint expression. The code can be written as:

```
class item extends uvm_sequence_item;
  rand int data[];
  rand int var_a, var_b, var_c;
  rand int invalid_value;
  function void pre_randomize();
    invalid_value =(var_a*var_b)/var_c;
  endfunction
  constraint value_c {
    value.size() == 1024;
    foreach( value[i] ) {
      value[i] != invalid_value;
    }
  }
endclass
```

We will further discuss the use of complex expressions in constraints in the next section.

Similarly, the post_randomize() function can also be used to eliminate array processing. A common pattern seen in generating item classes is the generation of payload data. Often the values used in the data are the result of simple values with few constraints. In our previous section, we saw a case where a user wanted an array of values, all larger than the prior one. Using post_randomize(), that code could have been written as:

```
class packet extends uvm_sequence_item;
  bit [31:0] data[];

  function void post_randomize();
    data = new[100];
    foreach( data[i] ) begin
      if(i ==0) begin
        data[i] = $urandom();
      end else begin
        do
          data[i] = data[i-1] + $urandom();
        while( data[i] < data[i-1] )
      end
    end
  endfunction
endclass
```

Here, instead of solving the data array during randomization. a simple procedural foreach loop iterates through each entry and generates a simple random value. This can be less work for a randomization engine and run faster. When moving randomization of specific variables like this to a post_randomize() function, make sure to also remove the rand keyword for the variable itself – it is no longer needed.

Note, in the code above, some additional logic was needed to account for overflow when adding the result of data[i-1] and $urandom. Generally, post_randomize() procedural code will be similar to the original constraint code, though it may be necessary to add additional code such as was done here.

D. Reduce complex math in constraint expressions

The process of finding a valid result in the constraint solver can be an iterative process where constraint expressions are evaluated multiple times. In classes that are randomized frequently, this can result in complex expressions being calculated many, many times. Consider the following code:

```
class item extends uvm_sequence_item;
   rand int x_pos[512], y_pos[512];
   int y_offset, var_a, var_b. var_c;
   constraint position_c {
   foreach( x_pos[i] ) {
     x_pos[i]==((12*(y_pos[i] + y_offset)*
                (var_a * var_c)) /var_b) -
                ((var_a * var_b) /var_c);
   }
  }
endclass
```

In the above code, as we both iterate through the foreach constraint block and also iterate through each expression one or more times, we are causing the randomization engine to calculate expressions with a number of complex expressions such as multiplies or divides. By rewriting the expression as well as adding some temporary variables, we can reduce the complexity of the expressions. The reduced complexities in the constraint expressions can result in less work for the constraint solver during each randomize call. A simple rewrite could look like:

```
class item extends uvm_sequence_item;
   rand int x_pos[512], y_pos[512];
   int y_offset, var_a, var_b. var_c;
   int tmp1, tmp2;
   function void pre_randomize();
     tmp1 = 12 * (var_a * var_c)/var_b;
     tmp2 = ((var_a * var_b) /var_c);
    endfunction

    constraint position_c {
     x_pos[i]==((tmp1*(y_pos[i]+y_offset))
                                 - tmp2;
     }
```

In the rewritten code, we were able to introduce two temporary variables that simplified the math in the constraint expression. Now, much of the complex math is computed prior to randomization through the use of a pre_randomize() call. Note, that in this case we were not able to remove all the complex math, but only reduce it.

III. Coding techniques for memory usage

Runtime performance is not the only measurement of compute resources. Memory usage is another important aspect to consider. Simultaneously randomizing large numbers of variables, especially where there are complex relationships between those variables can result in more memory being consumed as the randomization engine accounts for all the variables. Consider the following code:

```
class address_table extends uvm_object;
  rand int unsigned address[];
  int unsigned address_max;
  constraint table_values {
    address.size() == 250;
    foreach( address[i] ) {
      if( i != 0 ) {
        address[i] > address[i-1];
      }
      address[i] < address_max;
      // Other address[i] constraints
    }
  }
endclass
```

In the above code, there are a large number of simultaneous variables to solve for. This was discussed earlier in the paper in greater depth. As the size of the randomization problem grows, the memory required to solve it grows. The best approach to reducing memory is to simplify the problem and reducing the number of variables linked together. However, if that is not possible, consider solving it in post_randomize or perhaps solving it in smaller increments.

IV. CODING TECHNIQUES FOR PRODUCTIVITY

Writing constraints that are easy to maintain and debug is as important as writing constraints that run fast. As many UVM environments have grown, so too have the constraints that are in them. Many users have found that as their constraint code has grown larger, managing that code and debugging problems has become more difficult. There are several techniques we can use to help manage this complexity.

A. Constraint organization for re-use & ease of maintenance

One of the first steps in managing constraint complexity to follow a structured approach for organizing constraint code. Item classes generally have a limited number of random variables and constraints. Configuration classes are often larger, have more variables, and a larger number of constraints. The larger the classes and the randomization problem, the more organization can help.

1) Develop a consistent approach for organization of ranfom variables and constraint code.

The code developed for verification environments will likely by read, and often debugged, by engineers other than the original author. With that in mind, it is important to develop easy to read and maintain constraint code. An easy way to start that process is to follow a consistent format for the organization of random variables and constraints.

The authors have found it useful to organize your class structure so that random variables are grouped together and are separate from the constraints. Another common approach is to place random variables together, just prior to their use in constraint blocks. The most important thing here is to adopt a style and be consistent.

As classes grow larger, external constraint blocks can also help to preserve the readability of the class definition by separating the constraint names from the constraint code.

If the constraint code is subject to churn from project to project, but the variables remain the same, consider placing the constraint code into a separate file that is `included. This helps preserve the consistency of the random variables and supporting functionality but allows for easy migration of constraints as you switch projects.

2) Organize your constraints into multiple constraint blocks

Instead of writing one large constraint block for a class, create multiple smaller constraint blocks with descriptive names. In each constraint block place only those constraints needed to enforce a specific condition, mode, or rule. This will allow for easier, long term maintenance of the class – especially when the maintenance is done by a team of engineers. For example, this could look like:

```
class device_config extends uvm_object;
  rand mode_type mode;
  rand int max_size;
  rand int num_channels;
  rand bit retry_allowed;

  rand channel_config channels[];

  constraint half_mode_defaults {
    if( mode == HALF ) {
      max_size == 1024;
      num_channels = 2;
      retry_allowed = 1;
    }
  }
  constraint full_mode_defaults {
    if( mode == FULL ) {
      max_size == 512;
      num_channels = 4;
      retry_allowed = 0;
    }
  }
  constraint channel_defaults {
    channels.size() == num_channels;
    foreach(channels[i]) {
       channels[i].max_size ==
              local::max_size;
       channels[i].retry_allowed ==
              local::retry_allowed;
     }
  }

endclass
```

Splitting your constraints into well defined, and named, constraint blocks also makes it easier to use inheritance to redefine constraint blocks or to create new constraint blocks to augment the existing ones.

3) Break constraints into multiple classes

Create configuration objects for the different blocks in the design. Each of those configuration objects should contain only the random variables and constraints needed for that block. Create a single device level configuration object that creates each block level configuration. At this level, you can choose to randomize the block level objections concurrently.

If you randomize them concurrently, you could add constraints that span across the different blocks. An example of this approach was seen in the prior section.

You could also randomize the block level objects sequentially. Randomizing them sequentially would involve randomizing some device level parameters as random variables and then randomizing the each block's configuration object individually in post_randomize(). This could look like:

```
class device_config extends uvm_object;
  rand mode_type mode;
  rand int max_size;
  rand int num_channels;
  rand bit retry_allowed;

  channel_config channels[];

  constraint half_mode_defaults {/*…*/}
  constraint full_mode_defaults {/*…*/}

  function void post_randomize();
    channels = new[num_channels];
    foreach( channels[i] ) begin
      channel[i] =
channel_config::type_id::create(/*name*/,
                                          this);
      if( !channels[i].randomize() with {
        channels[i].max_size ==
             local::max_size;
        channels[i].retry_allowed ==
             local::retry_allowed;
      } ) else
      `uvm_error(“device_config”,
                 ”randomize failed”)
    end
  endfunction
endclass
```

Regardless of which approach is chosen it will help in managing the constraint code. Either approach can provide for an easy transition between block, device, and system level testing.

B. Use macros to replace repetitive code

The UVM library uses macros to replace repetitive coding and make it easier for code writers (and code readers). In situations where you have repetitive constraint code, consider doing the same thing. Macros could range in complexity from one or two constraints to an entire constraint block.

Using user defined macros native to a configuration file can also reduce constraint complexity. For instance, when a range of random variables may be either positive or negative, but the results of multiples of those values can't be greater than the sum of the absolute value of those numbers, a macro may be of use to simplify the constraint equation, cutting down the number of evaluations in that constraint block, for instance:

```
constraint macro_example {
  x inside {[-50:50]};
  y inside {[-50:50]};
  max_value inside {[0:100]};

  if(x < 0 && y < 0 ) –x-y <= max_value;
  if(x > 0 && y < 0 )  x-y <= max_value;
  if(x < 0 && y > 0 )  y-x <= max_value;
  if(x > 0 && y > 0 )  x+y <= max_value;
}
```

Could become much simplified if a absolute value (ABS) macro is created.

```
`define ABS(value) (((value) < 0) ? \
      (-(value)) : (value))
```

The macro_example above now becomes:

```
constraint_macro_example {
  x inside {[-50:50]};
  y inside {[-50:50]};
  max_value inside {[ 0:100 ]};
  `ABS(x) + `ABS(y) <= max_value;
}
```

C. Avoid modifying random fields manually after randomization

When developing randomized classes, it's important to think about the engineers that will need to use our code. A common temptation is to randomize a class and then external to the class manually tweak one or two variables to obtain some specific result. This can make it difficult for consumers of our code to understand just what we're attempting to do. A good rule is to limit any updates to random variables to the class itself. Instead of manually changing the value use a “randomize() with {}” call or even understand the condition and create constraints in the class itself to allow the condition to naturally happen. The key here is to let the constraint solver do the work for you and break the constraints into parts that make sense such that there isn't the need or temptation to set the random variable manually.

D. Create standalone environments to prototype and test constraints

Our complex constraint classes are just one part of a large UVM environment. The scale of the environment can make it difficult to quickly prototype new ideas or to run a large number of randomize calls on our class. For larger and more complex classes, it is a good idea to create a small, stand alone environment for just this class. Creating these standalone environments early in the process often only takes a few minutes. For this modest investment, you can save significant time as you introduce new ideas.

Creating a stand-alone environment also allows you to run a large number of randomize calls in succession. By doing this, you can get a good sense for the performance of the randomization in just that class – ignoring other simulation impacts. In an hour of simulation in the full environment, perhaps the a test would create 100,000 packets. In a stand-alone environment a test could still create 100,000 packets. However, in this instance, you would be able to observe only the time required for the randomization to complete. With that information you can then experiment with your code and try various optimizations. While a simulation profiling tool could provide similar information, the simplified environment coupled with the shorter run times can allow more efficient experimentation.

E. Record coverage on the random variables on your class

It can be enlightening to record coverage on the random variables in your classes. As these classes grow, and the constraints become more and more complex, there are often unintended consequences to constraint space. A condition that you may expect to happen very frequently may almost never happen, or perhaps never even happen at all. By adding covergroups to your randomized classes, you're able to see the distribution of how the different random variables are selected. Simply trigger the covergroup's sample method from post_randomize() to get a count from each randomize call.

If you collect coverage on classes for which you have standalone environments, you can quickly create large numbers of items and get a very good understanding of how your class will react in your larger environment

V. SUMMARY

Developing UVM environments with randomization code that both runs fast and is easy to maintain is very important. As times the problem may seem daunting and perhaps even mysterious. By following some straightforward rules, we can open the door to much high performance constraints that are indeed easy to maintain.

To improve simulation performance, the key concepts are:

- Selectively apply solve…before
- Carefully use foreach
- Use pre/post_randomize to reduce complexity
- Reduce the complexity of expressions used in constraints.

To improve productivity, the key concepts are:

- Use consistent practices for organizing randomization code
- Use macros to simplify and replace repetitive code
- Avoid modifying random variables manually
- Develop standalone environments for testing complex constraint code
- Record coverage on random values

As you write your own constraints, we invite you to try these approaches. More importantly though, we encourage you to experiment with your own code and grow in your confidence with randomization and writing constraints

VI. 6 ACKNOWLEDGEMENTS

The authors would like to thank the IBM Cores team for input to and use of the suggestions in this paper and the Cadence randomization team for guidance as we have worked through optimizing constraint code.