SVAs in IP: the Holy Grail or the Holy Snail

Lawrence Said
Markus Pugi

Cisco Systems
Ottawa, Canada

www.cisco.com

ABSTRACT

Adaptation of System Verilog Assertions (SVAs) has grown considerably due to their ease of use for both simple checks and implementing coverage metrics. As ASICs are pushed to use/reuse external and internal IP blocks of RTL, including their SVAs, DV testbenches consequently inherit these free SVAs. This increasing number of SVAs as a testbench grows can have adverse effects on productivity if not properly managed and implemented. This paper details typical SVA syntax performance issues, why they are difficult to diagnose and identify, and methods to find and address them. All of this gathered from recent experience improving a large-scale environment where more wall clock per simulation was spent in SVAs than in RTL & DV code combined. Further topics include implementing a regex mechanism for starting and stopping assertions, how this compares to typical module based start/stop mechanism and its application to vertical integration.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Constrained random verification operates on two fundamental principles:

1. The expected behavior of a device under test (DUT) can be described through a set of well-formed rules that can be verified
2. The set of stimulus required to exercise these rules is made deterministic through the use of coverage.

System Verilog assertions (SVA) augment the verification effort and can be used as both a checking and coverage tool. The use of SVAs has become increasingly prominent in industry and their widespread use is fundamentally attributed to their ease of implementation as a native verification tool capable of covering the state space in a more efficient and complete manner. However, as the number of assertion lines per lines of RTL (assertion density) increases, the impact on simulation performance becomes increasingly relevant. Our paper analyses the effect of various SVA implementations on the run-time performance of a testbench, and provides guidelines to reduce—and in certain cases mitigate—these contributors.

2 Impact Of Assertions

SVAs will impact simulation performance, regardless of their implementation. As the number of events per simulation tick increases, so will the run-time. Unfortunately, the magnitude of this impact is variable and somewhat difficult to predict at elaboration time. In most cases, the run-time performance hit of assertions is a low priority item and only investigated (if at all) near the end of the verification timeline. Although the assertion density of user implemented assertions may remain stable, the total number of assertions in the testbench is constantly growing as industry attempts to increase productivity through reuse of internal/external IP.

By virtue of the correlation between assertion density and wall-clock simulation time, the first step in improving run-time performance is reducing the number of assertions in the design. For instance, instead of creating an assertion to check every bit of a vector for X, bit-wise XOR the vector and evaluate the result in a single assertion. If there is a particular behavior that needs to be verified in a multiply instantiated sub-module, create the assertion in the top level module, instead of creating an assertion for each sub-module.

Aside from these simple examples, it may not always be possible (or practical for debug) to collapse several assertions into a single checking entity. As a second step, adhering to simple assertion coding guidelines can help reduce the first order effects of SVAs on run-time performance.

2.1 Custom Assertions

Assertions implemented by the verification engineer or designer (custom assertions) can significantly impact simulation performance if not implemented in a disciplined manner. The simplest guidelines to follow are:

- Avoid long temporal sequences
- (and to a lesser extent) Avoid local variables in fixed delay temporal sequences (use flops or $past instead)

For a more comprehensive list of guidelines (provided by VCS) see [7] (Reference Verification Methodology User Guide > Design for Verification (DFV) > Assertion coding guidelines > General rules)

2.1.1 Long temporal consequent expression

Take a simple safety assertion example:

```
asrt_safe: assert property (@(posedge clk) a |-> 1'b1 ##`N b);
```

Which states `b` must happen `` `N `` number of cycles after `a` asserts. The simulation run-time is linearly correlated to the temporal length of the sequence on the right side of the implication operator, as demonstrated by Figure 1. Every clock edge, the simulator forks off a thread to evaluate `asrt_safe` and kills that thread `` `N `` number of cycles later (once the assertion is evaluated) [3],[4]. If the antecedent always evaluates true every cycle, then `` `N `` number of threads will be running for the length of the simulation [5]. The worst case is an unbounded temporal sequence which never terminates, potentially forking off a thread every cycle until the end of the simulation (i.e. `` `N `` == sim-time).

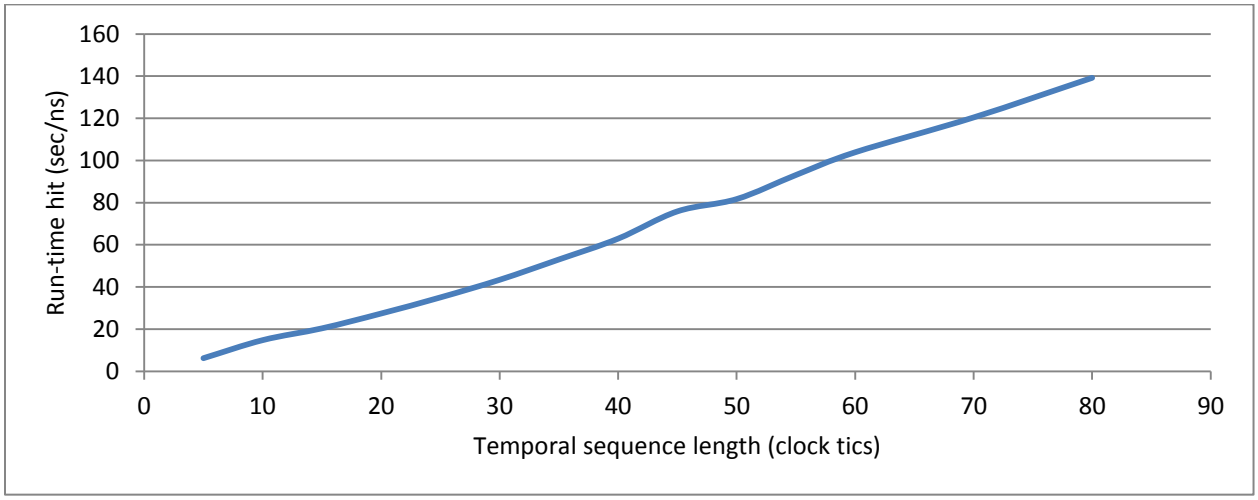

Figure 1: Run-time performance vs temporal sequence length

An equivalent single cycle assertion can be written to achieve the same safety check with significantly less (near-zero) performance impact by flopping `a`, `` `N `` number of times (say `a_ff`) and rewrite as:

```
asrt_safe: assert property (@(posedge clk) a_ff |-> b);
```

One could also opt for the `$past(name [, number_of_ticks])` operator which acts as a shift register. The performance of this system call degrades as the `number_of_ticks` increases [2]. Avoid this system call for `number_of_ticks` greater than a few clock cycles [5].

2.1.2 Unbounded temporal antecedent expressions

Unbounded temporal antecedent expressions like the following should be avoided at all cost

```
asrt_safe: assert property (@(posedge clk) a ##[0:$] b |-> c;
```

These can very quickly impact the performance of the simulation. If an unbounded sequence appears in the left-hand side of an implication, it is not subject to the first match restriction [2]. An unbounded sequence that evaluates true may always evaluate true sometime later in the future. For this reason, antecedent sequence can never end and the simulator will keep a running thread for each positive evaluation of `a`. VCS will usually issue a `[SVA-OPTCOV-LMFA]` warning for these types of assertions as their impact can be easily traced.

The solution is to implement an assertion that *will* be subject to the first match restriction. If `b` were a boolean, then the assertion could be rewritten as:

```
asrt_safe: assert property (@(posedge clk) a ##1 b[->1] |-> c;
```

Which guarantees the assertion will match on the first occurence of `b` after `a` evaluates true.

If b happens to be a sequence, then the goto operator will not work. Instead the `first_match` operator must be used.

```
asrt_safe: assert property (@(posedge clk) a ##1 first_match (##[1:$] b) |-> c;
```

The first_match operator is much less efficient than the goto operator [5], but is still preferred over the original unbounded temporal expression.

2.1.3 Bounded (long) temporal antecedent expressions

The bounded temporal antecedent expression is a subset of the unbounded one presented in 2.1.2. However, the bounded variant can have as significant of an impact depending on the length of the temporal sequence. Similar to what was shown in 2.1.1, simulation time is linearly correlated to the length of the antecedent temporal sequence in an implication property.

However, given that the antecedent sequence is not subject to the first match restriction (see 2.1.2), two sequences whose effective temporal lengths are the same could potentially impact performance differently. This is demonstrated pictorially in Figure 2, where the CPU run-time of two assertions is tracked versus simulation time. The difference between the two assertions is the ranged delay in the antecedent sequence. The difference in the performance impact (slope of the trendline) is directly attributed to the first match restriction. As the temporal sequence length increases, the greater the number of threads, and the greater the difference between the slopes of the two trendlines, as shown in Figure 3 (simulation performance inversly proportional to the trendline slope).

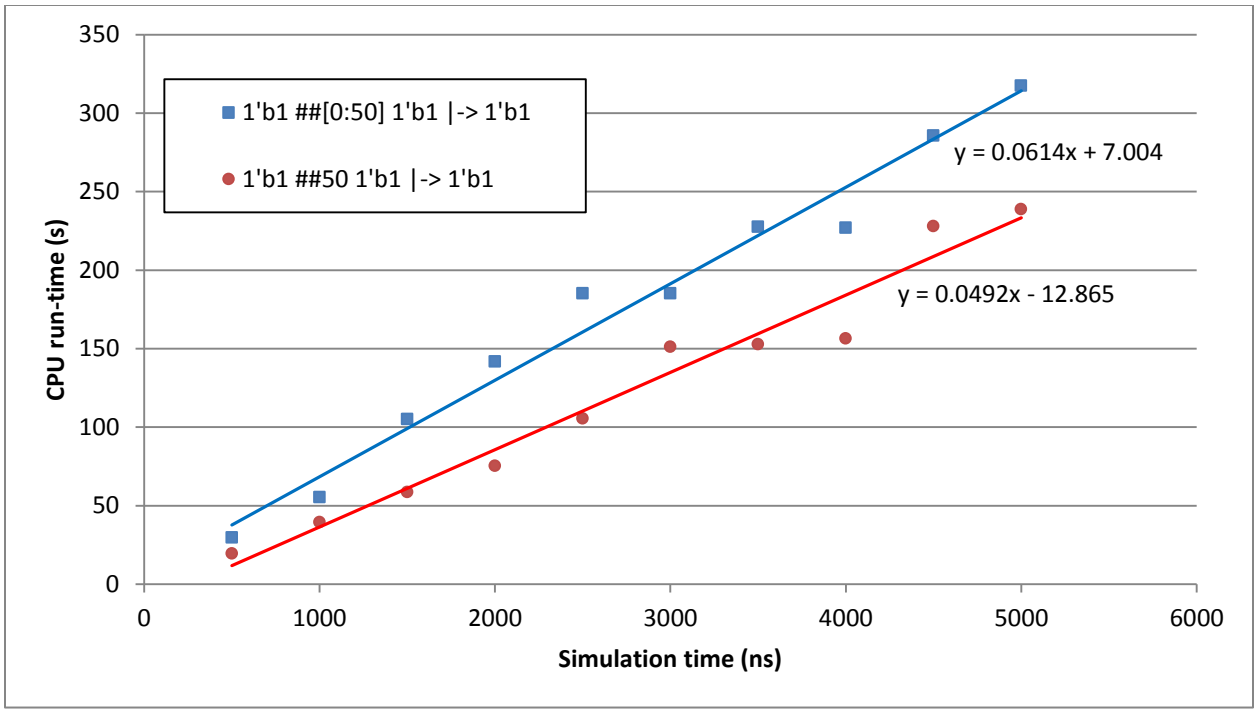

Figure 2: CPU run-time versus sim length with 50 cycle antecedent temporal sequence

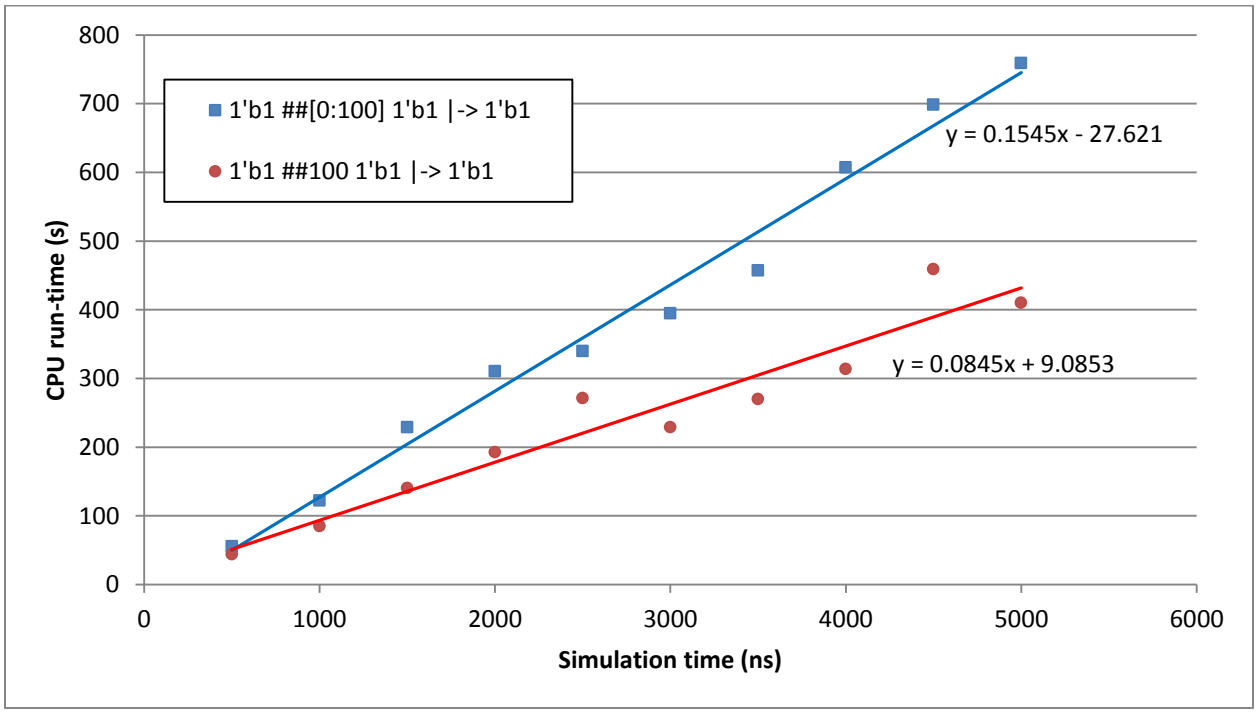

Figure 3: CPU run-time versus sim length with 100 cycle antecedent temporal sequence

2.2 Vendor Assertions

Vendors provide a library of assertions that can be instantiated anywhere in the design. These are tested assertions that are guaranteed to perform the checks they describe. For this reason, DV

engineers will opt to use these in their verification flow to eliminate the risk of error inherent in coding custom assertions [5].

2.2.1 Comparative analysis of vendor assertions

There are currently 53 vendor assertions (safety and fairness) provided by VCS. We present here a comparative analysis of the performance impact of each assertion (both temporal and immediate). Temporal sequences were kept as short as possible to still provide a passing assertion. The experiment was run for 50,000ns and with 8000 instances of the same module, for each vendor assertion module. The experiment was then repeated with coverage disabled (by setting all `coverage_level_*` parameters to 0):

```
<assertion module> #(...(),
                     .coverage_level_1(0),
                     .coverage_level_2(0),
                     .coverage_level_3(0)
                     ) asrt_inst_name(...);
```

The results of the experiment are displayed pictorially in Figure 4. Note that the results are displayed as run time (in CPU time) rater than as a performance ratio (simulation time / CPU time). This is done intentionally for illutrative purposes. Of all the assertions run, only two are not plotted: `assert_dual_clk_fifo` and `assert_valid_id`. The run-time of these assertions was an order of magnitude higher than the rest and were excluded from the plot. Section 2.2.2 will elaborate on the impact of enabling coverage collection when instantiating these assertion modules.

When using vendor assertions *with* default coverage level parameters, be cognisent of the following worst performing ones (for reference, the best, *measurable* performance is with assert_no_underflow at 5841.1 ns/s):

Assertion name	**Performance (ns/s)**
assert_dual_clk_fifo	3.3
assert_valid_id	17.3
assert_req_requires	48.7
assert_memory_sync	49.2
assert_multiport_fifo	50.2
assert_fifo	67.5
assert_req_ack_unique	76.6

Table 1: Worst performing vendor assertions with coverage enabled

When using vendor assertions *without* coverage enabled, be cognisent of the following worst performing ones (for reference, the best, *measurable* performance is with assert_no_underflow at 2902 ns/s):

Assertion name	**Performance (ns/s)**
assert_valid_id	36.9
assert_req_ack_unique	103.3
assert_multiport_fifo	160.8
assert_fifo	343.9

assert_stack	354.8
assert_dual_clk_fifo	365.4
assert_memory_sync	370.9

Table 2: Worst performing vendor assertions with coverage disabled

A detailed analysis of the performance characteristics of each assertion is beyond the scope of this paper. However, a general conclusion can be made that termporal assertions (especially those that hold a significant amount of state information) will exhibit poorer performance than zero-time instantaneous assertions.

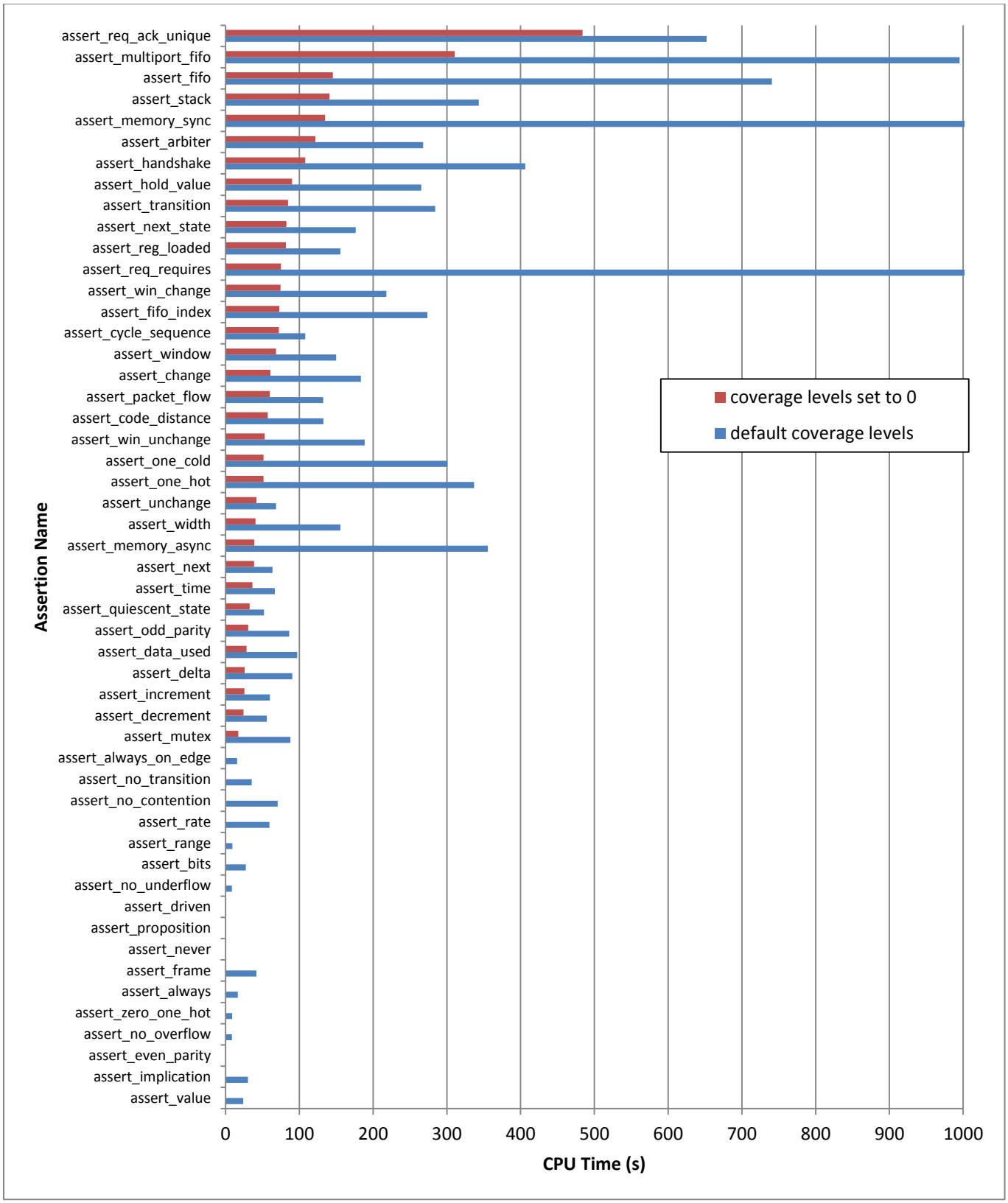

Figure 4: CPU run-time of VCS assertions with default and zero coverage levels

2.2.2 Know what's under the hood

Many of these custom assertions come with built-in coverage assertions that are enabled by default. A simple `assert_always` covers the evaluated expression. In most cases this is not needed and will impact simulation [1]. In fact, from the collected data, a performance hit is incurred when using certain vendor assertions without disabling coverage collection as illutrated in Figure 5 (**note:** ommited assertions were those not exhibiting any significant performance loss). The chart below indicates the performance impact of instantiatiating vendor assertions without specifying the `coverage_level_*` (i.e. instantiating with default coverage parameters). VCS vendor assertions come with three levels of coverage, two of which (levels 2 and 3) are disabled by default.

From the graph, simple assertions such as `assert_always` and `assert_value` will run approximately 40 to 60 times longer respectively with the default coverage parameters. If instantiating any assertions shown in Figure 5, review the cover properties carefully and decide whether the benefits outweigh the simulation run-time cost. When in doubt, disable the coverage parameter:

```
<assertion module> #(...(),
                     .coverage_level_1(0),
                     .coverage_level_2(0),
                     .coverage_level_3(0)
                     ) asrt_inst_name(...);
```

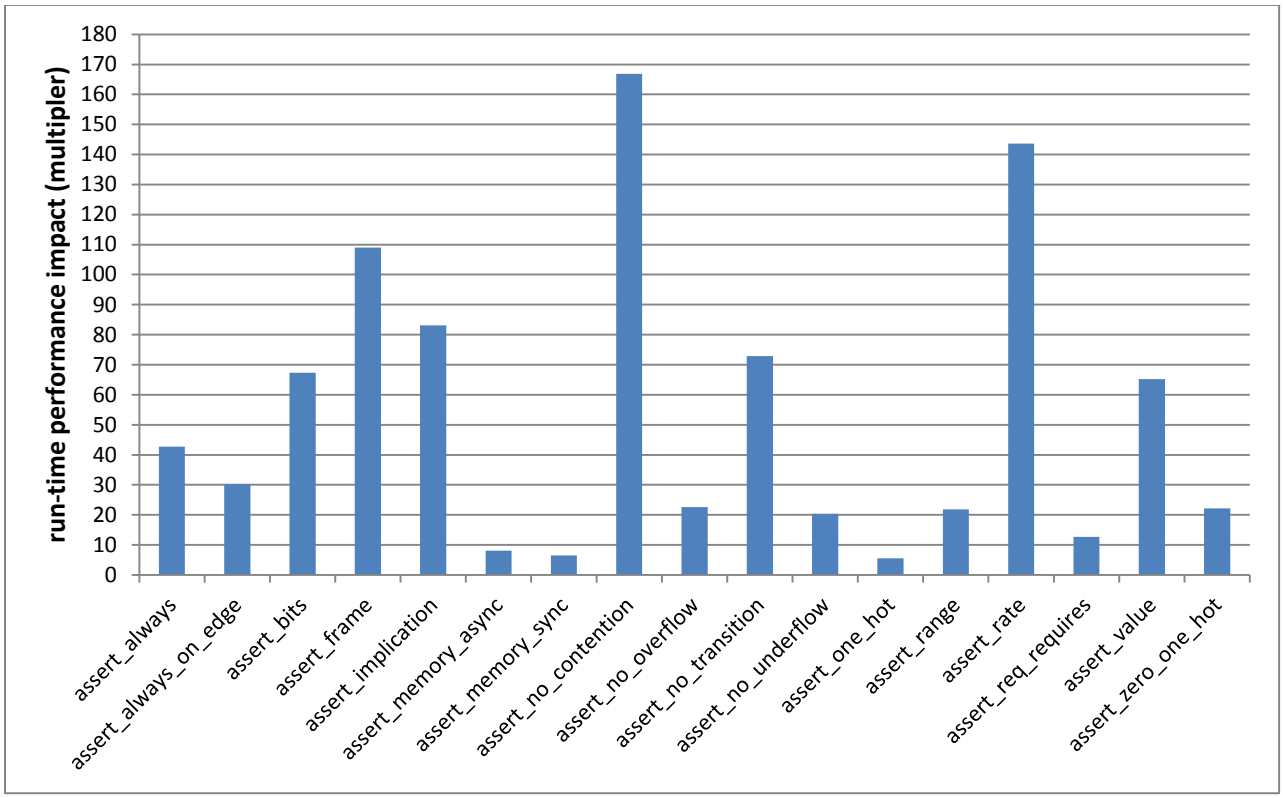

Figure 5: Performance impact of instantiating VCS assertions with default cover parameters

3 Identifying The Symptoms And Diagnosing The Problem

There are limited tools available for identifying SVA run-time hit. Currently, the *modus operandi* is to run with/without assertions in the design and track the run-time impact with the profiler. Once the symptoms are identified (i.e. it becomes known that assertions are slowing down simulation) the next step becomes identifying the culprits. Unfortunately, it may be the case that a single ill-coded assertion will not significantly impact performance, but that several thousand of them (instantiated across hundreds of modules) could lead to a appreciable decline in run-time performance. Moreover, the profiler provides a report at the module level; if custom assertions are coded inline with the RTL, only the module in which they were instantiated would appear in the profiler, leaving it up to the user to determine which of those inlined SVAs (if not all) are the main contributors.

Recent improvements to the profiler have enabled it to identify the runtime performance of the vendor assertion library, as shown in Figure 6 below.

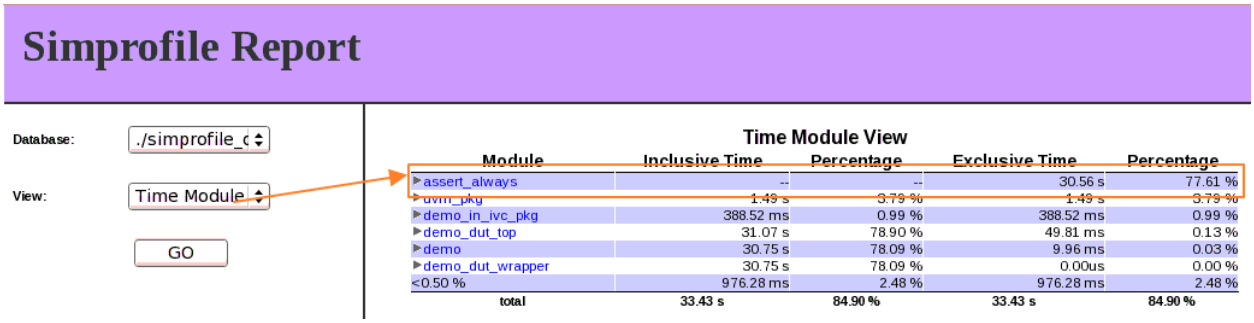

Figure 6: Screenshot of module view simprofile report

However, once a performance inhibiting SVA module has been identified, one could investigate if VCS switches are causing the performance impact. For example, the `-assert enable_diag` directive prevents certain optimizations in the compiler and can lead to appreciable performance degredation. Otherwise manual detective work is necessary to see if any performance improvement is even possible. This is due to the fact that it is not shown in the profiler that the performance impact is likely due the the sheer number of instantiations of a given SVA module. To reduce the burden of SVA performance contribution at this point is tedius. A proactive approach to reduce the number of assertions upfront is preferred, or perhaps group SVAs into SVA categories of "bring-up/debug" and "tapeout necessary" that can be compiled in/out with defines.

Often this is not possible if the SVAs are inherited and therefore re-used from a block of IP.

A re-active approach to SVA performance identifying and isolating is to:

1. print a list of all assertion modules in a DUT
2. sort the list
3. manually identify which blocks have the highest assertion density
4. Non-IP blocks: investigate if assertions can be reduced/collapsed, or `coverage_level_*` parameters modified to disabled unnecessary coverage SVAs as discussed in section 2.2.1 and 2.2.2.

5. IP blocks: investigate if a given block has unnecessary assertions that can be runtime disabled

4 Controlling Assertions In The Design

When inheriting assertions from a piece of reusable IP, it may not be possible to apply the above guidelines to improve run-time performance. In some cases, the verification engineer can analyse and opt to disable (in part or completely) the inherited assertions if the benefits outweigh the cost. SystemVerilog provides methods for controlling assertions through various system calls that either require the full hierarchical path name of the assertion or that globally control groups of assertions for a given category. This method does not scale well as the number of assertion grows and cannot be vertically integrated easily into another testbench with different hierarchical paths.

We propose a mechanism that can selectively disable a subset of assertions within the design through regular expression pattern matching. This would allow users, for example, to control assertions instantiated in specific sub-modules in the design without having to specify complete hierarchy.

4.1 Implementation

At elaboration time, the complete list of assertions in the design is gathered and assimilated into a string database (strDB) maintained by a SVA controller. This can be done by utilizing VPI Extensions to SystemVerilog [6], to implement:

1. using `vpi_scan()` function to recursively traverse the entire DUT module hierarchy
2. use `vpi_iterate(vpiAssertion, child)` to extract the assertion modules
3. use `vpi_get_str(vpiFullName,)` to extract the assertion name as a string

Further details can be found in Section 39 of IEEE System Verilog Standards 1800-2012 [4]. Each assertion's path is unique in it's string name returned by the VPI and can then be added to an array of strings to create a string database. The SVA controller can now provide an additional layer atop the SystemVerilog system calls to interface with the user through a list of custom APIs. Through the use of its string database, SystemVerilog system calls (`$asserton`, `$assertoff`) and the UVM provided DPI-C function for regular expression matching (`uvm_re_match`), the controller provides an API to enable/disable SVAs through regex pattern matching.

Regex pattern matching becomes quite relevant and useful when performing vertical reuse. For example, if a given environment had custom code to enable/disable a set of assertions manually:

```
function void sva_stop_module(string a_module_name);
   bit [4095:0] l_path_array;
   $cast(l_path_array, a_module_name);
   $assertkill(0,l_path_array);
endfunction
```

The full path to the assertion would need to be provided:

```
sva_stop_module("dut_top.dut_inst.block_inst.dummy_assert");
```

However, the full path to the module hierarchy is not reusable either vertically or horizontally unless the the path to the block remains unchanged. One could handle this via a set of defines:

```
`define BLK_PATH dut_top.dut_inst.block_inst
`define TO_STR(x) `"x`"
sva_stop_module(sformatf("%s.dummy_assert",`TO_STR(`BLK_PATH)));
```

In which a helper (`TO_STR) define is necessary to pass a define into a string. The extra overhead above is no longer necessary and can be achieved with a simple regex function call:

```
sva_stop_module_regex("*block_inst.dummy_assert");
```

Where the custom `sva_stop_module_regex` simply performs:

1. a regex pattern match on the string database (strDB)
2. calls `sva_stop_module` using the match from the database

This provides a mechanism to start/stop the assertion using regular SystemVerilog system calls without specifying the full path name.

5 Conclusion

A simulation's CPU run-time performance will degrade proportionally to the number of events processed every simulation tick. Assertions and cover properties are not exempt from this rule, and can in fact significantly impact performance as discussed in this paper. Writing custom assertions with discipline and being cognisent of the performance characteristics of vendor assertions is key to reducing the memory footprint and runtime contribution of assertions in the design. As the number of inherited assertions from reusable IP increases, methods of detecting performance degredation become increasingly relevant as the assertion density increases in today's testbenches.

Adherence to simple guidelines and efficiently managing inherited assertions (especially from reusable IP) can help the DV engineer achieve the full potential of SVAs without having to suffer the drawbacks of a slower testbench.

6 References

[1] Thinh Ngo, *A Method to Dynamically Disable/Enable SystemVerilog Assertions*, Silicon Valley 2015

[2] Haque, Michelson, Khan, *The Art Of Verification with SystemVerilog Assertions*, Fremont California 2006

[3] Stuart Sutherland, *Who Put Assertions In My RTL Code? And Why? How RTL Design Engineers Can Benefit from the Use of SystemVerilog Assertions*, Silicon Valley 2015

[4] Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language, Std 1800-2012, IEEE, NY, http://standards.ieee.org/findstds/standard/1800-2012.html

[5] J. Bergeron, E. Cerny, A. Hunter, A. Nightingale, *Verification Methodology Manual for SystemVerilog*, Spriner 2006

[6] VPI Extensions to SystemVerilog, http://www.eda.org/sv-cc/hm/att-1630/01-12-5_SV_vpi_annotated.pdf

[7] VCS-MX Online Documentation,
"$VCS_HOME/doc/UserGuide/userguide_html/USERGUIDE_HTML/index.html"