A linear approach to complex constraint structures

Elihai Maicas

Tchiya Dayan

Intel

Haifa, Israel

www.intel.com

ABSTRACT

Constrained-Random Verification is the accepted method for verification in ASIC development teams. It is efficient, and enables the verification engineers to reach coverage goals quickly, and to rapidly reveal exceptional corner-case issues. The essence of this technique requires constructing a set of constraint blocks in order to control the DUT stimuli. Sometimes, creating these constraints in the conventional way severely degrades the simulation memory and runtime performance. This paper presents general guidelines, and unconventional techniques that can optimize the constraints coding process. The presented approach provides similar statistical attributes obtained from the known straightforward approach, yet with minimal performance impact.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

In this paper we will describe in detail a case where a set of constraints was written to characterize a frame data structure that in general, is a type of three dimensional array. At first, the relations between the frame elements were taken from the spec, and translated to System Verilog syntax, without any consideration to simulator performance. When progressing from simple tests to more complex ones, among other scenario modifications, the frame size needed to be enlarge. At some point, we witnessed a degradation in simulation performance, and after some analysis, we found that the frame randomization time increased dramatically. We then started looking for ways to decrease the latter.

2. Background

Intel's 3D camera (RealSense™) is a complex system-on-board with various components: IR camera, color camera, IR laser projector etc. One of these main components is the Imaging ASIC. The Imaging ASIC essentially converts raw data from the sensors to high rate color and depth frames, which would be delivered to the host, for further processing by a designated application.

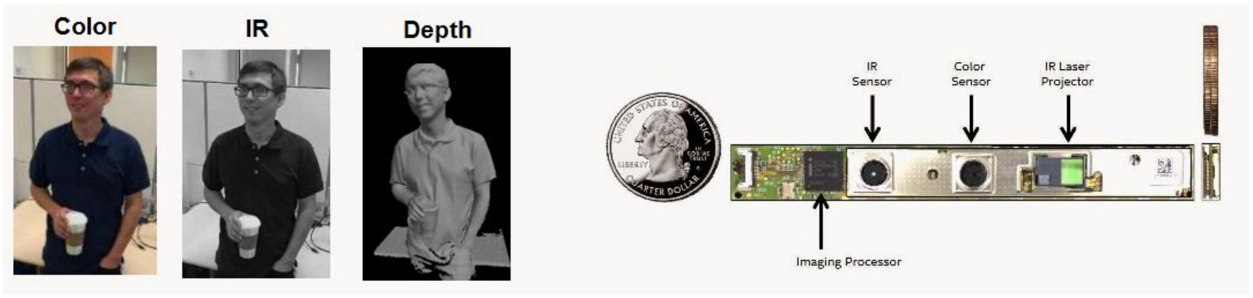


Figure 1. Intel® RealSense™ 3D Camera

During the verification process, each sub-unit is isolated and thoroughly verified prior to integration into the subsystem. One of those sub-units requires this rather complicated data structure:

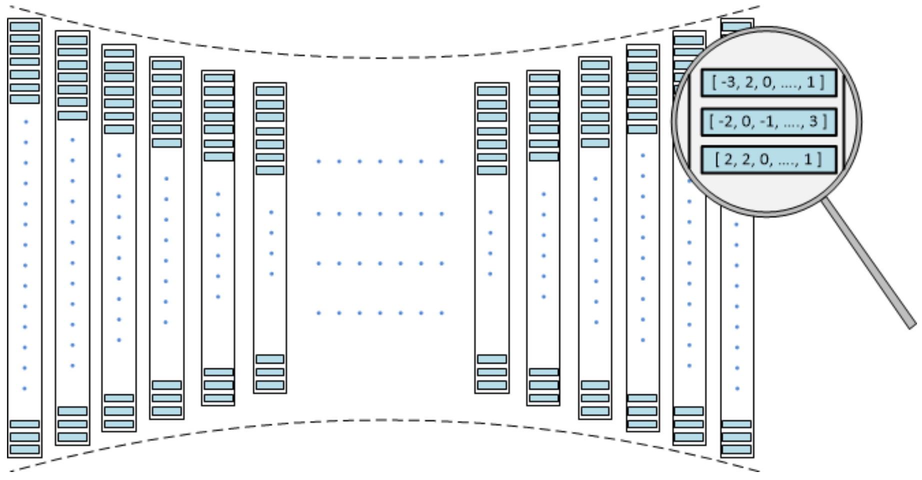


Figure 2. Target randomized frame structure

Figure 2 shows a frame composed of an array of columns (up to 1920). Each column is an array of "samples" (up to 5000; light blue rectangles), and finally, each sample is an fixed array of 16 attributes.

The relations of the data structure elements is as followed:

- The number of columns is constant
- The number of samples varies per column
- Each of the 16 attributes in a sample is an integer in the range [-4:3]
- The sum of all the attributes in each sample is a specific integer in the range [-2:2], which depends on the number of samples in its column

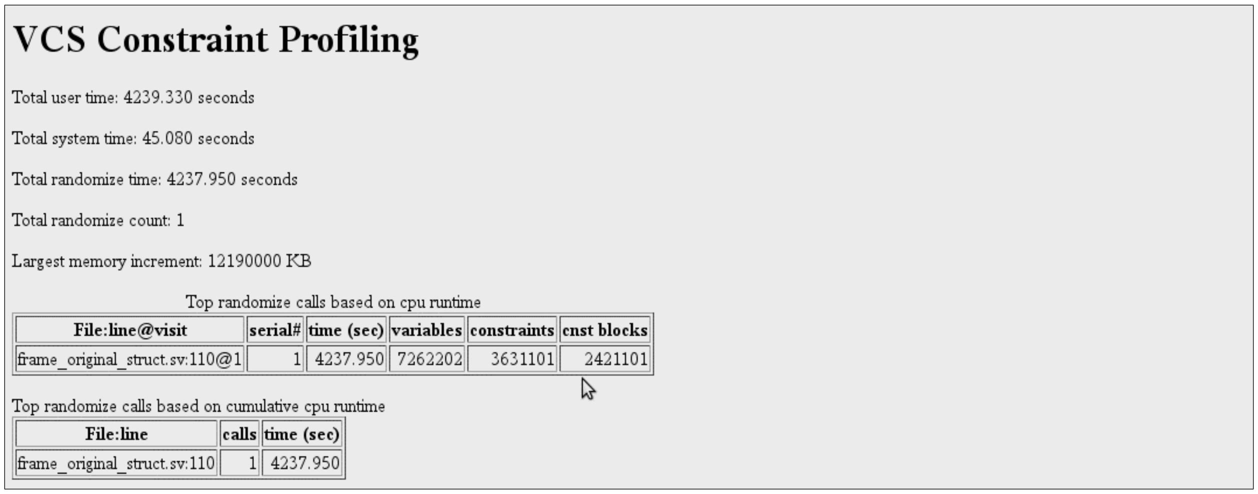
3. The Problem

As stated above, initially the constraints describing the data structure were written in the conventional way, that is, via straightforward encoding in System Verilog. Simulating a small number of frame size gave good coverage results, with reasonable test time. Increasing those values towards the full sizes caused severe degradation of simulation performance, both in test time and memory consumption. As an example, a single full 1920x5000 (1920 columns, each with 5000 samples) sized frame test took 17(!!) hours.

3.1 Problem Analysis

As the saying goes: "Understanding the Problem is Half of the Solution." Analyzing the VCS simulation profiler reports, revealed that the majority of the test time is consumed by the randomization of the data structure in Figure 2, which had a huge number of constraints.

Following this understanding, we used the VCS constraints profiler during debug and analysis (obtained by using the runtime switch `+ntb_solver_debug=profile`), to get a brief summary of the desired profiler results.



VCS Constraint Profiling

Total user time: 4239.330 seconds

Total system time: 45.080 seconds

Total randomize time: 4237.950 seconds

Total randomize count: 1

Largest memory increment: 12190000 KB

Top randomize calls based on cpu runtime

File:line@visit	serial#	time (sec)	variables	constraints	cnst blocks
frame_original_struct.sv:110@1	1	4237.950	7262202	3631101	2421101

Top randomize calls based on cumulative cpu runtime

File:line	calls	time (sec)
frame_original_struct.sv:110	1	4237.950

Figure 3. VCS Constraint Profiling – report snippet

In this paper we will demonstrate the measurements and analysis of a simpler testcase: 1100x1100 frame (1100 columns, each with 1100 samples).

Figure 3 shows that in the demonstrated testcase there are more than <u>3.5 million constraints</u> to be solved, and that a single randomization of such frame takes more than <u>70 minutes</u>.

Analyzing the constraint solver profiling reports of multiple tests with various frame sizes, we found that there is an exponential relation between the number of constraints to the time and memory consumption of the simulation:

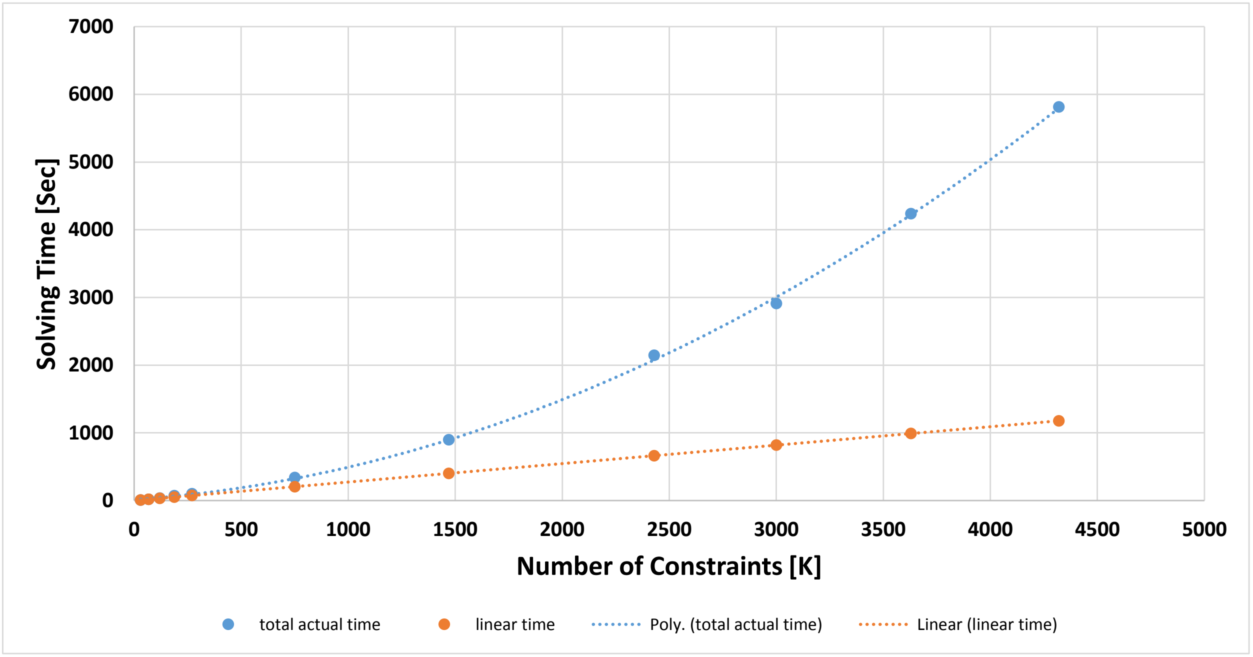


Figure 4. Constraints solving time vs. Number of constraints

4. The Solution

Our main goal was to limit the number of constraints to be solved simultaneously by the constraint solver, while maintaining the correctness of the solution. A few techniques that were applied on the testcase described (1100x1100 frame) and their impact on solving time are described hereafter.

4.1 Technique 1: Revise the variable type of the attributes

Each of the 16 attributes is an integer number in the range [-4:3]. Initially the constraint was written as follows:

```
rand shortint attr[16];

constraint attr_span_c {
  foreach (attr[ii]) {
    attr[ii] inside {[-4:3]};
  }
}
```

Figure 5. Attribute variable type - original

We replaced the shortint variable with a bit signed packed array, which has an intrinsic limitation on its span. A 3-bit signed packed array values span the range [-4:3]. This representation is based on the two's complement method. There is no need for further constraints on the attributes:

```
rand bit signed [2:0] attr[16];
```

Figure 6. Attribute variable type - revised

This change improved the solving time by 70%, as described in the table below:

Table 1. Attribute variable type – profiler comparison

Attr. var type	Number of Constraints	Avg. solving time [sec]
Shortint	3,631,101	4,237 (1.17 hr.)
Signed packed array	1,211,101	1,270 (21.1 min.)

4.2 Technique 2: Replace the sum() constraint with post_randomize sequential loop

The sum of all attributes is a specific integer in the range [-2:2]. The array sum() method can be used in a constraint block, but adds a large number of constraints:

```
// define new data type, to be used in casting
typedef bit signed [7:0] signed_8_bit_t;

rand bit signed [2:0] attr[16];

constraint attr_sum_c {
  attr.sum() with (signed_8_bit_t'(item)) == 1;
}
```

Figure 7. Array sum - original

In order to avoid the large number of constraints that the sum() method adds, we removed it and imitated its effect with a simple sequential loop, placed in the post_randomize method of the sample object class. Specifically, we let the constraint solver assign 16 random attributes in the randomization stage, and then iterated through the array and adjusted its values, so its sum will match the desired value. For instance, if the initial sum came out to be -2, and the target sum is 1, all that is left is to add one to three attributes, and we are done:

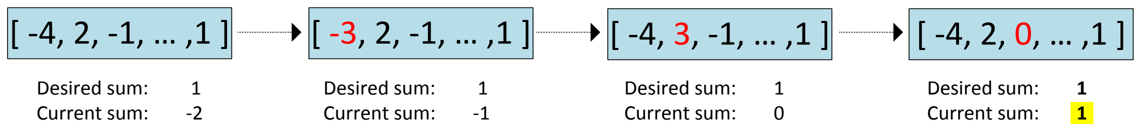


Figure 8. Adjusting attributes array on post_randomize - example

The code snippet in Figure 9 demonstrates the sequential code used for the adjustment of the attributes array values. Due to required correction symmetry (increase or decrease the array sum), only one part was inserted :

```
function void sample::post_randomize();
  signed_8_bit_t init_sum, trgt_sum, diff_sum, cntr;
  shortint       idx;

  // calc init sum
  init_sum = attr.sum() with (signed_8_bit_t'(item));

  // set helper vals and diff_sum
  // ...

  // adjust array values to fit target sum
  if (diff_sum > 8'sb0) begin
    while (1) begin
      if (attr[idx]>-3'sh4) begin
        attr[idx] -= 3'sb1;
        cntr      += 8'sb1;
      end
      if (cntr == diff_sum) break;
      idx = (idx+7)%16;
    end

  // symmetric operation for (diff_sum < 8'sb0)

endfunction
```

Figure 9. Array sum - revised

When comparing the statistical attributes of the numbers generated from the two methods, we found that they are identical. In addition, we achieved a significant improvement in the solving time, 99.3% from the previous step and 99.8% so far:

Table 2. Array sum limitation – profiler comparison

Sum Method	Number of Constraints	Avg. solving time [sec]
Constraint	1,211,101	1270 (21.1 min.)
Post randomize	1,101	9

4.3 Technique 3: Separate orthogonal parts of the frame to be randomized individually

Typically the full structure of the frame is randomized, sent to the driver as an item, and then driven to the design-under-test:

```
// create the frame
m_frame = new;

// randomize prior sending to the driver
m_frame.randomize();

// send frame to the driver
```

Figure 10. Randomizing the frame - original

In most cases, there is no correlation between the columns, so the driver was revised to process each column at the time, and the columns were randomized on-the-fly:

```
// create the frame
m_frame = new;

foreach (m_frame.column_arr[ii]) begin
  m_frame.column_arr[ii].randomize();
  // send column to the driver
end
```

Figure 11. Randomizing the frame - revised

This final step eliminated almost all of the constraints, and resulted in minimal frame randomization time:

Table 3. Array column randomization – profiler comparison

Column Rand	Number of Constraints	Avg. solving time [sec]
Bulk	1,101	9
On the fly	1	2

5. Conclusions

We have demonstrated a few techniques to optimize large constraint structures, by using different type of variables, and porting complex calculations to the post randomize phase. In the presented case, the final improvement was above 99% (from 1.17 hr. to 2 sec.) in frame randomization time.

This performance boost was a result of transferring the code from a declarative constraint structure (used by the solver engine) to standard procedural code. The procedural code execution time was insignificant (full testcase runtime was 17 hours, and after optimization was reduced to 90 seconds, split by the randomization time (45 sec) and the procedural code).

Although this is a specific case scenario, methods described in this paper can be used in various similar scenarios.

6. Refferences

[1] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," New York, 2013.

[2] Synopsys, Inc., *VCS Mx / VCS MXi User Guide*, J-2014.12 ed., 2014.

[3] ReaLSense SDK Design Guidelines, version 2