Dramatically Increase the Performance of SystemC Simulations

Dr. Greg Tumbush, AMI Semiconductor, Colorado Springs, CO
Mark Hupp, AMI Semiconductor, Colorado Springs, CO

1. Introduction

The Starkey Labs Colorado IC Design Center (now the AMI Semiconductor Colorado Springs Design Center) is in the business of designing and manufacturing ICs for the medical marketplace. The new digital hearing aids we design are fully modeled using SystemC. These large models are leveraged for architectural exploration, early firmware development, and co-verification with the RTL design. It was stated at the 2005 OCSI Symposium panel [4] that a panelist found system simulations with SystemC are already too slow for his large systems and unusable for next generation systems. Clearly, simulation speed of SystemC is already an issue and will only become more important. We believe that with proper performance evaluation and profiling SystemC simulations can be orders of magnitude faster.

This paper will delve into increasing the simulation speed of a "standalone" model, that is, a model compiled using g++ and not using a commercial simulator. We use a 2-part approach to performance optimization. The first part we call *gross profiling*. Gross profiling uses specific test vectors and is used to minimize system level overhead. A good example of system level overhead is VCD file dumping. Using the gross profiling methodology allowed us to attain a 32x simulation speedup. After gross profiling is completed the effects of SystemC datatypes and operators become apparent. The second part of performance improvement is examining SystemC datatypes and operators. Using this methodology resulted in approximately a 2x speedup.

In the first section we will look at previous work done in this area. We will then take a brief look at profiling tools and share our experience. The two benchmark systems we used will be discussed. Then, the gross profiling method will be explained along with benchmark results. The effects of coding style will be explained, again with benchmark results. Importantly, code will be provided to avoid the problems found with certain SystemC datatypes and operators.

2. Previous Work

An excellent presentation examining the effects of SystemC datatypes, SC_THREADs versus SC_METHODS, scheduling, etc. can be found in [1]. This presentation found that the single largest performance improvement can be found by not using SystemC datatypes. However, by not using SystemC data types not all the Verilog-like data manipulation operators, such as bit-selection, range, and concatenation, are available. In [2] a fast bit-accurate datatype is proposed that will allow easy data manipulation. A result of this work is available as the Mentor Algorithmic C Data Types[3]. However, these datatypes are written to be synthesizable which will reduce simulation speed as well as the level of available abstraction.

3. Profiling Tools

The profiler provided with Microsoft Visual Studio provided good usable output but would crash whenever SC_THREADS were used. The Linux profiler, gprof, works with SC_THREADS and the output, while cryptic, did lead us to several substantial performance improvements.

4. Benchmark Systems

Two benchmark systems were used. The first represents an entire simulation environment for a 260K gate ASIC destined for a hearing aid application. In addition to numerous functional models, it contains an embedded processor, bus functional models (BFMs) to drive/monitor the interfaces, and a command line interface. Many different BFMs were created. A unique BFM was a simple model of analog blocks. These were added to test register control of the analog portion of the ASIC. For debugging without the overhead of a simulator and the cost of a simulator license the simulation environment dumps to a VCD trace file. This benchmark is used to evaluate gross profiling and was compiled using the SystemC 2.0.1 library.

The second benchmark represents a simulation environment for the most complex block in the ASIC. The block is approximately 38.5K equivalent gates. In addition to the functional model, the simulation environment contains a few BFMs. This benchmark is used to examine the results of SystemC datatypes and operators. The system model was determined to be too large for all the code changes required to examine this effect. The second benchmark was compiled using the SystemC 2.1 library.

5. Testing Methodology

All tests were run on a standalone PC running Microsoft Windows XP Pro, version 2002, Service Pack 2. The

profiling platform is a 3.2 GHz Pentium 4 with 1 MB of RAM. In addition, the tests were run on a Linux server running Red Hat Linux 7.3 . The server is a dual 2.0 GHz Xeon with 6 GB of RAM. For the PC the test was run five times and the wall clock time of each test noted. There was very little discrepancy between the five tests. The benchmark was compiled in release mode (i.e. not debug mode) on the Windows machine and with no optimization and -O3 optimization enabled on the Linux machine. The Linux machine was tested using the */usr/bin/time* utility. This utility reports the real, user, and system time. The real time is the elapsed real time between invocation and termination. The user time is the user CPU time. The system time is the system CPU time. When comparing the standalone Windows machine to the Linux server we will use wall clock time for the Windows machine and the user + sys time for the Linux machine. All times given are in seconds.

6. Gross Profiling

The gross profiling methodology is used to minimize system level overhead. The test applied should put the system in an idle state (not reset). In this case the profiling tool should report very little activity. If it does not, the reasons need to be examined. Knowledge of the system is very important because the profiler output can be quite cryptic. Keep in mind that more than functional models are being simulated. Every piece of code that makes up the system simulator is running. BFMs that were written as throw-away code have a bad habit of appearing in the release code. Breakpoints are also useful for determining code that is running excessively during idle mode.

In the case of the large benchmark, the Linux profiling utility *gprof* reported that the command line interface was polling for user input every clock cycle. This polling frequency is not required and a poll for user input every 1,000 clock cycles was found to be sufficient. Again using *gprof,* many system calls to methods with "vcd" in their name such as vcd_trace_file, vcd_bool_trace, etc. were noticed. Viewing of VCD files is typically not required unless a bug is found. An option in the make file was added to leave VCD dumping off by default. Lastly we searched the code for BFMs, typically at the testbench level, that do not always need to run. For the vast majority of tests the analog blocks are not used. A make file option to exclude the analog blocks by default was added. The instantiation of the analog model (using "new") and the connection of the modules ports were excluded using ifdefs. To summarize we optimized the simulation using the following steps:

1. Use a profiler to look for functions that are called excessively
2. Turn off VCD generation
3. Exclude blocks that are not required for a particular simulation

The results we obtained are found in Table 1, Table 2, and Table 3. From these results you can see that reducing the number of times we polled on the command line by a factor of 1,000 resulted in a reduction in runtime of 25% for the Windows machine, 4% for the unoptimized Linux machine, and 6% for the optimized Linux machine. Supressing VCD output resulted in a further reduction of 91% for the Windows machine, 90% for the unoptimized Linux machine, and 91% for the optimized Linux machine. Excluding the analog blocks resulted in a further reduction of 53% for the Windows machine, 50% for the unoptimized Linux machine, and 46% for the optimized Linux machine. Overall the total reduction in runtime was 96.9% for the Windows machine, 95.1% for the unoptimized Linux machine, and 95.6% for the optimized Linux machine. This represents a speed-up of 32.6, 20.5, and 22.7 respectively.

CMD Line	VCD	Analog	Windows Wall Clock Time
Yes	Yes	Yes	173.0
No	Yes	Yes	130.0
No	No	Yes	11.3
No	No	No	5.3

Table 1: Gross Profiling results – Windows

CMD Line	VCD	Analog	real	user	sys
Yes	Yes	Yes	195.1	189.3	5.32
No	Yes	Yes	186.7	182.9	2.96
No	No	Yes	20.2	18.7	0.07
No	No	No	9.50	9.47	0.02

Table 2: Gross Profiling results – Linux, no optimization

CMD Line	VCD	Analog	real	user	sys
Yes	Yes	Yes	178.0	173.0	5.00
No	Yes	Yes	167.3	165.6	1.67
No	No	Yes	14.4	14.4	0.04
No	No	No	7.9	7.8	0.04

Table 3: Gross Profiling results – Linux, -O3 optimization

7. Overhead of SystemC Datatypes and Operators

Now that gross profiling is complete, the effects of SystemC datatypes and operators become significant and apparent. It is well known that use of SystemC data types will cause significant overhead in simulation. We have

anecdotal evidence that use of the concatenation operator causes a large overhead and to a lesser extent the use of bit-selection and range operations. Obviously each variable needs to be a SystemC type. In a later section we show how these operations can be performed without the variables being SystemC types.

To test SystemC datatypes and operators the second benchmark is used. The original block uses no SystemC data types. We then measure the overhead of simply making the variables SystemC types. We then benchmark all the permutations of these three constructs. In contrast to gross profiling, we want to use a test that fully exercises the block.

An example of the SystemC bit-selection operator appears in Figure 1. This operation will select bit three from variable B and assign it to A.

```
A=B[3];
```

Figure 1: SystemC Bit-Selection Operator

An example of the SystemC range operator appears in Figure 2. This operation will select the lower nibble from variable B and assign it to variable A.

```
A=B.range(3,0);
```

Figure 2: SystemC Range Operator

An example of the SystemC concatenation operator appears in Figure 3. This operation will concatenate variables B, C, and D and assign it to variable A.

```
A=(B,C,D);
```

Figure 3: SystemC Concatention Operator

As can be seen in Table 4, Table 5, and Table 6 the use of SystemC datatypes results in a performance degradation of 78% in Windows, 150% in unoptimized Linux, and 90% in optimized Linux. Surprisingly, there is no discernable trend to report on the use of bit-selection, range, or concatenation. For the Windows test, the slowest combination is bit-selection only while the unoptimized Linux test the combination of bit-selection and concatenation resulted in the slowest test. For the optimized Linux test, range selection was found to be the slowest. Also interesting to note is that some combination of operators run faster than the use of no operators! The only hard fact that can be gleaned from these results is to avoid the use of SystemC types. In the next section we provide alternative methods for the bit-selection, range, and concatenation operators.

Using *gprof* we did note that with the SystemC 2.0.1 library each concatenation operator produced numerous "new" and "delete" operations. This does not seem to be the case with the SystemC 2.1 library.

SystemC Vars	Bit-select	Range	Concat	Wall Time
No	No	No	No	208.9
Yes	No	No	No	372.8
Yes	No	No	Yes	350.9
Yes	No	Yes	No	366.8
Yes	No	Yes	Yes	363.9
Yes	Yes	No	No	416.9
Yes	Yes	No	Yes	397.4
Yes	Yes	Yes	No	391.1
Yes	Yes	Yes	Yes	400.1

Table 4: Performance of SystemC Operators – Windows

SysC Vars	Bit-select	Range	Concat	real	user	sys
No	No	No	No	557.6	551.5	0.77
Yes	No	No	No	1390.1	1381.9	0.98
Yes	No	No	Yes	1417.6	1411.2	0.13
Yes	No	Yes	No	1428.8	1422.8	0.17
Yes	No	Yes	Yes	1380.3	1373.9	0.35
Yes	Yes	No	No	1452.5	1447.0	0.11
Yes	Yes	No	Yes	1519.6	1513.3	0.54
Yes	Yes	Yes	No	1503.7	1497.5	0.25
Yes	Yes	Yes	Yes	1512.7	1504.9	0.4

Table 5: Performance of SystemC Operators - Linux, no optimization

SysC Vars	Bit-select	Range	Concat	real	user	sys
No	No	No	No	288.7	282.2	0.48
Yes	No	No	No	543.2	537.8	0.05
Yes	No	No	Yes	504.1	499.0	0.13
Yes	No	Yes	No	612.5	607.1	0.14
Yes	No	Yes	Yes	481.2	475.9	0.22
Yes	Yes	No	No	530.2	525.0	0.11
Yes	Yes	No	Yes	522.1	517.1	0.24
Yes	Yes	Yes	No	472.0	464.9	0.55
Yes	Yes	Yes	Yes	541.0	535.3	0.21

Table 6: Performance of SystemC Operators - Linux, -O3 optimization

8. Methods to Replace the Bit-Selection, Range, and Concatenation Operators

In this section we will provide alternative methods for the bit-selection, range, and concatenation operators so that

native C types can be used. We have found in the creation of our models that these three operators are the dominant reason designers require SystemC datatypes. In the interest of space no consistency checking of types and bit-widths are included in these examples.

The bit-selection operator, as seen in Figure 1, can be replaced by the method in Figure 4. The replacement for Figure 1 using this method is *A=return_bit(B, 3).* A bit insertion method is in Figure 5. If we want to insert a binary one in bit position three of variable B we could use: *B = insert_bit(B, true, 3).* This is equivalent to *B[3] = 1* using the SystemC bit-selection operator.

The range operator, as seen in Figure 2, can be replaced by the method in Figure 6. The replacement for Figure 2 using this method is *A=return_range(B, 3, 0).* To insert a range use the method in Figure 7. If we wanted to insert 0xF into A[4:1] we could use: *A=insert_range(A, 0xF, 1, 4).* This is equivalent to A.range(4,1) = 0xF using the SystemC range operator.

The concatenation operator in Figure 3 can be replaced by multiple calls to method *insert_range*. For example, suppose variable B, C, and D in Figure 3 are nibbles. Variable A can be computed by the code in Figure 8.

```
template<class TYPE>
inline bool return_bit(TYPE value, int
position)
{
  return (bool)((value >> position) &
  1);
}
```

Figure 4: Bit-Selection for Native C Types

```
template<class TYPE>
inline TYPE insert_bit(TYPE value,
bool bit_value, int position)
{
  TYPE MASK = 1;
  if (bit_value)
    return value | (MASK <<
    position);
  else
    return value & ~(MASK <<
    position);
}
```

Figure 5: Bit-insertion for Native C Types

```
template<class TYPE>
inline int return_range(TYPE value,
int high_range, int low_range)
{
  return (value >> low_range) & ((1 <<
  (high_range-low_range+1)) - 1)
}
```

Figure 6: Range selection for Native C Types

```
template<class TYPE>
inline TYPE insert_range(TYPE full_in,
TYPE range_in, int startbit, int
bitwidth)
{
  TYPE mask;
  TYPE mask_out;
  mask = (1 << bitwidth) - 1;
  mask_out = ~(mask << startbit);
  return (full_in & mask_out) |
  ((range_in & mask) << startbit);
}
```

Figure 7: Range Insertion for Native C Types

```
A = insert_range(A, B, 8, 4);
A = insert_range(A, C, 4, 4);
A = insert_range(A, D, 0, 4);
```

Figure 8: Concatenation for Native C Types

9. Summary

This paper gives the user a basis for dramatically increasing the performance of SystemC simulations. By only including blocks that are required for a particular simulation, turning off VCD dumping, and looking for inefficiencies in BFMs and testbench level code, we were able to achieve an approximately 32X speedup. Then, by avoiding the use of SystemC types whenever possible we were able to achieve an additional 1.5X speedup. We also illustrated replacements for the SystemC bit-selection, range, and concatenation operators that function with native C types.

10. References

1. Adam Donlin. *Optimizing Models of an FPGA Embedded System.* 2nd North American SystemC Users Group, www.nascug.org

2. Andres Takach, Simon Waters, and Peter Gutberlet, *Fast bit-accurate C++ datatypes for functional system verification and synthesis*, 2004 Forum of Design Languages.

3. Mentor Graphics, Inc., *Algorithmic C Data Types,* http://www.mentor.com/products/c-based_design/index.cfm

4. 2005 OSCI Technology Symposium: "Isn't it Time You Moved Up to SystemC", *Panel: The Business of ESL*