Code Optimization for Enhancing SystemC Simulation Time

Homa Alemzadeh, Soheil Aminzadeh, Reihaneh Saberi, Zainalabedin Navabi

CAD Research Laboratory, Department of Electrical and Computer Engineering
School of Engineering, University of Tehran
{homa, soheil, saberi, navabi}@cad.ece.ut.ac.ir

Abstract

The main contribution of this paper is suggesting a number of techniques to enhance SystemC simulation time. Simulation speed is very important, especially at the early stages of the system design. On the one hand, these techniques guide SystemC developers, and on the other, they can be used in automatic code translators. The experimental results show a significant improvement in the simulation time of SystemC codes.

1. Introduction

With the increasing complexity of digital systems, and the reduced time to market, Electronic System Level (ESL) design is regarded as the main design methodology for implementing large digital systems.

Most of the new complex designs have software parts as well as hardware modules. This implies the necessity of developing hardware and software of a system in parallel, which helps designers with hardware/software co-design, co-verification, and co-simulation.

The heart of the ESL design methodology is a high-level language that is used for specification of both software parts and hardware elements of a complete system. SystemC is one of the most popular System Level Design Languages (SLDL) which is actually a C++ class library with certain characteristics for hardware description. The major facilities of SystemC are implementing main hardware-oriented parameters, close correspondence with RT-Level descriptions, and its high-level interface with C++ [1].

Regarding SystemC as the future choice for modeling hardware in ESL design, there is a need for translating previously designed modules from traditional HDLs into SystemC. A reliable automatic converter makes time-to-market shorter by skipping this time-consuming manual transformation and avoiding errors that frequently happen in this process by filling the gap in between [2].

In this paper we present a number of optimization techniques for describing hardware with SystemC which aim at maximizing the efficiency of system design by improving the simulation time. These techniques can be mainly used in the process of automatic conversion from VHDL to SystemC descriptions. In particular we explore a number of possible alternatives for converting VHDL constructs to SystemC and evaluate the simulation time of the design in case of each conversion. We also take advantage of the most popular techniques proposed for C/C++ code optimization in our conversions. The experimental results show a significant enhancement in simulation speed of SystemC codes. The most related work on the conversion of Verilog HDL to SystemC is [2]. In [3] some guidelines for optimizing the conversion of Verilog HDL constructs to SystemC are introduced and [4] proposes techniques for optimizing SystemC performance. To the best of our knowledge, this work is the first attempt for optimized VHDL into SystemC conversion with considering HDL constructs as well as the C/C++ code optimization techniques.

The rest of this paper is organized as follows, Section 2 describes the importance of efficient VHDL to SystemC conversion. Section 3 introduces some guidelines for optimized conversion of VHDL constructs to SystemC. Section 4 lists a number of common C/C++ code optimization techniques which are used in our conversions. A number of experimental results are presented in section 5, and finally last section concludes the paper.

2. VHDL to SystemC Conversion

In recent years time-to-market constraint has led designers to use pre-designed and pre-verified intellectual properties in new projects. Soft and firm IP-cores are mostly available as traditional HDL

descriptions such as Verilog and VHDL. Also many designers prefer to develop their new designs in conventional HDLs with which they are most familiar and use the common reliable simulation and synthesis tools developed based on these languages. On the other hand, the need for co-design, co-simulation, and co-verification of hardware and software in new complex system designs, makes translating these HDL codes to languages such as SystemC inevitable.

In order to have an effective system design, an efficient simulation model for precise and high-speed system exploration is needed. Faster simulation speed enables design analysis and system partitioning in the earlier steps of design and finally lead to manufacturing more efficient systems. We have applied proposed techniques to VSC converter of UT SystemC Studio [5], an environment for RT level language translation, simulation, and synthesis. VSC and TVS converters of SystemC Studio automatically translate between VHDL/Verilog synthesizable codes and RT level SystemC.

3.   SystemC Optimizations

In this section we investigate some of the most important constructs of VHDL hardware description language and their alternative equivalent structures in SystemC. By evaluating and comparing the simulation time in each case we find which conversions lead to the best simulation time.

3.1. Conditional Statements

Conditional statements in VHDL can be converted to *switch-case*, *if-else* or *conditional signal assignment* in SystemC. Our experiments show that the SystemC *conditional signal assignment*s have a better simulation time than SystemC *switch-case* statements, and *switch-case* statements simulate faster than *if-else* statements. The results depict that all of these SystemC constructs have a better simulation time that their equivalent *if* and *case* statement in VHDL.

3.2.   Component Instantiation

There are two methods for hierarchical module instantiation [6]. The first one is constructor initialization list and second one is using pointer and dynamic memory allocation. As in VHDL and Verilog, in each case two alternative ways for binding the ports are possible: Binding by name and Binding by position, therefore there are four types of component instantiations, which we refer to them as *Method 1.a*

and *Method 1.b, Method 2.a and Method 2.b* respectively.

Our experimental results show that using pointers and dynamic memory allocation is better than initializing using constructor initialization list.

3.3. SC_METHODs instead of SC_THREADs

There are three kinds of processes in SystemC: *SC_METHOD*, *SC_THREAD* and *SC_CTHREAD*. *SC_METHOD* processes in SystemC implement function-wise concurrency while *SC_THREAD* processes implement true threading.

SC_THREAD with its own individual thread stack and local variables is slower than *SC_METHOD*. Usage of threads slows down the simulation performance in most cases due to context switching overheads. On the other hand supporting *wait* statement is the advantage of *SC_THREAD*. Therefore as long as there is no delay or dynamic event needs, *SC_METHOD* is the best choice that makes a significant enhancement in simulation time [4].

The most important usage of *wait* statement is in test bench generation. On the other hand efficient test bench simulation is highly valuable. This part introduces a novel approach for using *SC_METHOD*s instead of *SC_THREAD*s in test benches. The proposed methodology is simple and straightforward. We adapt the method presented in [4] and use event sensitivity list for test bench *SC_METHOD* and replace all "*wait (Δt, unit-of-time)*" statements with "*next_trigger(Δt, unit-of-time)*", notifying the proper *event* in zero time.

Test vector values are saved in a number of arrays. Each *SC_METHOD* can access the contents of these arrays through static integer indexes. This approach can be applied for generation of different kinds of signal values, such as arbitrary single values in random time intervals, random value in specific time intervals, and periodic repeated patterns. Figure 1 shows a general test bench which is implemented by a simple *SC_THREAD* and generates two series of arbitrary and periodic values for signal *a*.

```
a = (sc_lv<4>)("1010");
wait(20,SC_NS);                    Arbitrary
a = (sc_lv<4>)("0110");             Value
wait(11,SC_NS);

for (int i = 0; i < 100; i++)
{
   a = (sc_lv<4>)("0010");
   wait(14,SC_NS);                  Periodic
   a = (sc_lv<4>)("0111");          Repeated
   wait(8,SC_NS);                    Value
}
```
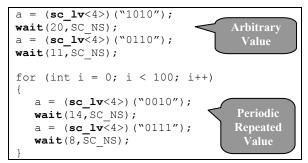
Figure 1 – A *SC_THREAD* Testbench (Partial)

Figure 2 shows application of our methodology to code of Figure 1. The *SC_THREAD* of Figure 1 is converted to an *SC_METHOD* which is sensitive to event *e1*. Two global arrays *A_LUT* and *T_LUT* are declared and initialized for saving the value and time of test vectors to be applied to variable *a*. Indexes *i, t, pIndex,* and *k* are declared as static. These indexes are used to controlling the flow of applying test vectors to variable *a* in each call of *SC_METHOD* after notifying *e1*.
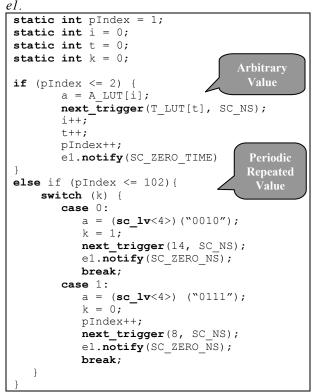
```
static int pIndex = 1;
static int i = 0;
static int t = 0;
static int k = 0;

if (pIndex <= 2) {                    Arbitrary
        a = A_LUT[i];                  Value
        next_trigger(T_LUT[t], SC_NS);
        i++;
        t++;
        pIndex++;
        e1.notify(SC_ZERO_TIME)        Periodic
}                                      Repeated
else if (pIndex <= 102){               Value
    switch (k) {
        case 0:
            a = (sc_lv<4>)("0010");
            k = 1;
            next_trigger(14, SC_NS);
            e1.notify(SC_ZERO_NS);
            break;
        case 1:
            a = (sc_lv<4>) ("0111");
            k = 0;
            pIndex++;
            next_trigger(8, SC_NS);
            e1.notify(SC_ZERO_NS);
            break;
    }
}
```

Figure 2 - Equivalent *SC_THREAD* Testbench

The results of automatic generation of *SC_METHOD* test benches for different types of values can be used in an automatic test bench generation tool.

4. C/C++ Code Optimizations

In this part, we concentrate on different C/C++ code optimizations which using them cause a program to run faster. These techniques mostly include manual optimizations in C/C++ codes, which usually are not handled by compilers. We mainly use the optimization techniques presented in [7-9] and some other online documents in this area. This section lists a number of these techniques.

Function call is causes of one of the most inefficiency in simulation time. Therefore full or partial function inlining can improve simulator speed. Partial

inlining means, inlining of simple conditions which may cause the immediate return in the case of function call [9]. Local variables are more efficient than global one. C/C++ professional developers proposed to use local variables, declare them in the innermost scope, and initialize them as soon as possible. They also advised avoiding type casting by selecting the best type of variables [7, 8]. Loop unrolling [8], using simpler termination conditions in loops, and breaking nested conditional chains in binary fashion are other useful guidelines for C/C++ development techniques.

5. Experimental Results

We applied optimization techniques explored in the previous sections to some VHDL benchmarks and their equivalent SystemC codes. The results of applying C/C++ optimization techniques are shown in Table 1, and the simulation results for applying VHDL to SystemC construct optimizations are depicted in Table 2.

C/C++ optimization	Improvement Percentage
Function Inlining	17.7%
Data Type Optimization	4.9%
Loop Unrolling	1.3%
Switch instead of *if*	29.5%
Break Binary Fashion	13.3%
Simplifying Termination Condition	4.5%

Table 1- Simulation Performance Improvement by Applying C/C++ Optimization Techniques

We also apply all of these techniques to an industrial VHDL design. Our benchmark was the VHDL description of SAYEH Processor [10]. The SystemC optimization methods and C/C++ optimization techniques were applied to VSC converter of UT SystemC Studio [5]. We automatically converted VHDL code of SAYEH to its SystemC version by VSC and evaluated the simulation time with and without optimizations. In order to see the real effect of the optimizations, we wrote a sort program in SAYEH assembly language and ran it on this processor. This program reads ten numbers from memory, shuffled them, sorts them and writes them back into memory for a number of times. Since in this program most of the SAYEH instructions are used, most of the processes in the code will be used and the processor will be busy running them, thus, we can see the effect of our optimizations better.

The simulation speed of SystemC description of SAYEH processor will improve about 8% by replacing *SC_THREAD*s with *SC_METHOD*s. We also compare the simulation speed of SAYEH arithmetic unit with

both four-value logic and two-value logic. The results show that simulation time is improved about 2.5% by applying this technique. We calculated the simulation time for the initial SystemC code and the optimized version of it after applying most of above optimizations. As expected, our optimization techniques resulted in a better simulation time and we gained about 7% improvement in the simulation speed.

6. Conclusions

Because of the complexity of today's designs, it is important that the designers have a unified environment for developing and simulating an entire system. SystemC provides this environment. In this paper we proposed a number of techniques to improve the simulation speed of converted SystemC codes from VHDL. We considered different conversions for main VHDL constructs and also used the famous C++ code optimizations in the conversions. The results of this paper are applied to VSC converter of UT SystemC Studio to automatically generate efficient SystemC codes. Although some of these techniques lead to expansion of code, and applying some of them lead to generating a non-readable SystemC codes, but in automatic code translation, the advantages of considering these techniques is more than their disadvantages.

7. References

[1] S. Mirkhani, Z.Navabi, *System Level Design Languages*, The VLSI Handbook, Chapter 86, CRC Press, 2nd Edition, Dec. 2006.

[2] L. Mahmoudi Ayough, A. Haj Abutalebi, O. F. Nadjarbashi and S. Hessabi, "Verilog2SC: A Methodology for Converting Verilog HDL to SystemC," Proc. of the 11th International HDL Conference (HDL Con 2002), pp. 211-217, San Jose, California, USA, March 2002.

[3] L. Mahmoud Ayough, A. Haj Abutalebi, O. F. Nadjarbashi and S. Hessabi, "Reusing Verilog IP Cores in SystemC Environment by V2SC", Ascend Design Automation, San Jose, USA. [Online Document], Available On: http://www.us.design-euse.com/articles/article12918.html

[4] K.P.CHAMATH AYATILLEKE, "Optimizing SystemC performance", M.S. thesis, Institute for System Level Integration, Livingston, Scotland, UK, 2002.

[5] A. M. Gharehbaghi, R. Saberi, H. Alemzadeh and Z. Navabi, "SystemC Studio: Translation for TLM Combined Simulation and Synthesis" Tool, University Booth of University of Tehran, *11th Design, Automation and Test in Europe Conference (DATE'08)*, Munich, Germany, March 2008.

[6] D. C. Black, J. Donovan, *SystemC: From the Ground Up*, Kluwer Academic Publishers, 2004.

[7] Writing Efficient C and C++ Code Optimization, Koushik Ghosh, [Online Document], Available on: http://www.codeproject.com/cpp/C___Code_Optimization.asp "Optimizing C and C++ Code", [Online Document], Available on: http://www.eventhelix.com/RealtimeMantra/Basics/OptimizingCAndCPPCode.htm

[8] C++ Optimization Strategies and Techniques, Pete Isensee, [Online Document], Available on: http://www.tantalon.com/pete/cppopt/main.htm.

[9] Advanced Compiler Optimization Techniques, Wind River, [Online Document], Available on: http://www.techonline.com/community/related_content/20437?print.

[10] Z. Navabi, *Verilog Digital System Design*, 2nd Edition, McGraw-Hill Press, 2006.

SYSTEMC OPTIMIZATION TECHNIQUE	PERCENTAGE OF IMPROVEMENT *(NORMALIZE VERSUS VHDL OR WORST CASE SYSTEMC)*			
Conditional Assignment	*If-else*	*Switch-case*	*Conditional Signal Assignment*	-
	1.38	6.3	13.9	-
Component Instantiation	*Model_1.a*	*Model_1.b*	*Model_2.a*	*Model_2.b*
	12.3	16.6	36.06	20.5
SC_THREAD versus SC_METHOD	*SC_THREAD*	*SC_METHOD*	-	-
	1	8.08	-	-
4-value-logic versus 2-value-logic	*4-value-logic (SC_THREAD)*	*4-value-logic (SC_METHOD)*	*2-value-logic (SC_THREAD)*	*2-value-logic (SC_METHOD)*
	1	4.6	2.59	5.1
Multiple Processes versus Single-Process	*Multiple Processes*	*Single Process*	-	-
	22.9	19.4	-	-

Table 2 - Simulation Speed Improvement by SystemC Optimization Techniques