Center for Embedded and Cyber-Physical Systems
University of California, Irvine

A Light Weight SystemC Library for Faster Compilation

Farah Arabi, Tim Schmidt, Rainer Dőmer

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{farabi,schmidtt,doemer}@uci.edu

CECS Technical Report 16 - 07
October 20, 2016

Contents

1 Abstract **2**

2 Background **2**

3 Introduction **2**

4 Source Code Transformations **2**

4.1 Basic functions . . . 2

4.1.1 Functions with Plain Old Data Return Types . . . 3

4.1.2 Functions with Class Derived Return Types . . . 3

4.2 Operator Functions . . . 4

4.3 Template Functions . . . 4

4.4 Inline Functions . . . 5

4.5 Functions with modifiers . . . 5

5 Validation and Testing **6**

5.1 Translator modifications . . . 6

5.1.1 Translator User-Defined Transformations . . . 7

5.2 Boost . . . 7

5.3 S2C Testbench Design . . . 8

5.4 Script file . . . 8

6 Experiments and Results **8**

6.1 Compilation Time Results . . . 9

6.2 AST Size Results . . . 9

6.3 Evaluation . . . 12

7 Conclusion **12**

1 Abstract

The Light Weight SystemC Library is a summer undergraduate project that was completed to aid a larger project known as Parallel SystemC Simulation on Many-Core Architectures [1]. In order to reduce the compile-time of the parallel compiler [2], the light weight SystemC library was achieved through multiple transformations made to the header and source files of the original SystemC library [3]. After completing the transformations, 17 designs were tested with the new light weight SystemC library. Our results show that the light weight SystemC library is able to decrease the number of nodes traversed by the SystemC compiler by %14.31-%23.36 for the 17 test designs. As a consequence, our experimental results also show that the new library is able to decrease the compilation time by %14.41-%19.14 for the same 17 designs.

2 Background

The optimization of the SystemC library into a light weight SystemC library is only a small part of the project *Parallel SystemC Simulation on Many-Core Architectures*. The goal of the *Parallel SystemC Simulation on Many-Core Architectures* is to develop a SystemC based simulator that consists of a dedicated SystemC compiler and a parallel SystemC simulator to implement *Out-of-Order Parallel Discrete Event Simulation* (OoO PDES) [5] for SystemC. OoO PDES can execute threads in parallel and out-of-order which results in a faster simulation speed while maintaining the classic SystemC modeling semantics. The light weight SystemC library developed in this summer project and described in this report serves to shorten the time needed to analyze and compile a parallel C++ model.

3 Introduction

In this report, we discuss the process of attaining an optimized light weight SystemC library. First, we discuss the necessary transformations that had to be made to optimize the SystemC library. Second, we describe the testing technique along with the ROSE compiler [6] that provided us with tangible results. Lastly, we analyze these results and validate that the developed light weight SystemC library indeed shortens compilation time.

4 Source Code Transformations

The transformation into a light weight SystemC library involves some changes in the header files of the library and their respective source files. In the original SystemC library, which is implemented in the C++ language, the header files include numerous function definitions. These functions exist in four main different types and are referred to as basic, operator, template, and inline functions. Different types of functions entail having somewhat different transformational procedures. However, there are a few steps in the transformations that are common to most of the different types of functions.

In general, we first move all the function definitions from the header files to their corresponding source files. This reduces the amount of code the parallel compiler has to process and thus is expected to lead to faster processing time. Then, we use the scope resolution operator to specify the necessary scopes to access a specific function in the source file. As for the header file, the function body is deleted and what remains is only the function declaration.

Some exceptions to this general procedure exist and are discussed in detail in the sections that follow.

4.1 Basic functions

We will first describe how basic functions are transformed.

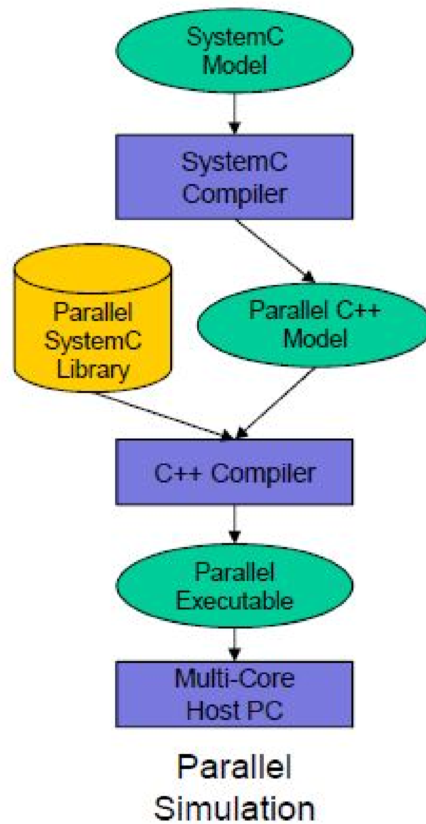

Figure 1: Parallel SystemC Simulation [1]

4.1.1 Functions with Plain Old Data Return Types

Table 1 shows an example of the transformation of a basic function with a plain old data return type. In the upper left cell of Table 1, we can see that the function double to_double( ) const {...} is defined in class sc_time and namespace sc_core. In order to transform this function, it is first copied into the source file where the two function scopes are specified right after the return type. The first is the namespace known as sc_core and the second is the class the function belongs to known as sc_time.

After that, the function definition in the header file is reduced to a function declaration as portrayed in line 100 in the upper right cell of Table 1. With that, the transformation is complete.

4.1.2 Functions with Class Derived Return Types

Class derived return types are three types: pointer (sc_object∗ wait(...)), value (sc_object wait(...)), and reference (sc_object& wait(...)). In order to transform this type of function, we first copy the definition into the source file. Next, the scopes are specified for both the return type and the function. The class derived return types usually require specifying only one scope, the namespace, right before the return type. The positioning of the scopes for the actual function is just like that of the plain old data type functions. Basically,

the corresponding namespace and class are specified after the return type and right before the function name.

	Before	After

sc_time.h	36 namespace sc_core {
	...
	73 class sc_time
	74 {
	75 public:
	...
	105 double to_double() const
	106 {
	107 return sc_dt::uint64_to_double(m_value);
	108 }
	...
	201 };
	...
	250 } //namespace sc_core
		36 namespace sc_core {
		...
		73 class sc_time
		74 {
		75 public:
		...
		100 double to_double() const;
		...
		142 };
		...
		217 } //namespace sc_core

sc_time.cpp	Empty	74 double sc_core::sc_time::to_double() const 76 { 77 return sc_dt::uint64_to_double(m_value); 78 }

Table 1: The transformation of a basic function with a plain old data return type

4.2 Operator Functions

Operator functions are transformed the same way as basic functions with one minor exception. The scopes being specified for the function are to be placed right before the word "operator" and not the actual operator. Table 2 shows an example of the transformation of an operator function.

	Before	After

sc_bit.h	68 namespace sc_dt
	69 {
	...
	86 class sc_bit
	87 {
	...
	150 operator bool () const
	151 { return m_val; }
	...
	186 };
	...
	423 } //namespace sc_core
		68 namespace sc_dt
		69 {
		...
		86 class sc_bit
		87 {
		...
		190 operator bool () const;
		...
		235 };
		...
		345 } //namespace sc_core

sc_bit.cpp	Empty	110 sc_dt::sc_bit::operator bool () const 111 { return m_val; }

Table 2: The transformation of an operator function

4.3 Template Functions

Templates are a C++ feature that allows a function or class to work for various data types without rewriting it each time it is used. In the header files, the template is written right before the class.

When we copy a function definition found in a template class into the source file, we also add a template right before each function definition. This can be seen in line 67 of the new source file in Table 3 (bottom right corner). Similar to a basic function, we also need to specify the function scopes in the source file. Looking at line 36 of the original header file in Table 3, we can see that the first scope is sc_core, the namespace. The second scope in line 56 is sc_event_expr, the template class that the function belongs to. In the new source file, we notice a modification to the class scope because of the presence of the <> operators

and the typename T resulting in sc_event_expr<T>.

	Before	After

sc_event.h	36 namespace sc_core {
	...
	55 template< typename T >
	56 class sc_event_expr
	57 {
	...
	102 push_back(type const & el) const
	103 {
	104 sc_assert(m_expr);
	105 m_expr->push_back(el);
	106 }
	...
	193 };
	...
	500 } //namespace sc_core
		36 namespace sc_core {
		...
		55 template< typename T >
		56 class sc_event_expr
		57 {
		...
		74 push_back(type const & el) const;
		...
		85 };
		...
		385 } //namespace sc_core

sc_event.cpp	Empty	67 template< typename T > 68 void sc_core::sc_event_expr<T>:: 69 push_back(type const & el) const 70 { 71 sc_assert(m_expr); 72 m_expr->push_back(el); 73 }

Table 3: The transformation of a template function

4.4 Inline Functions

An inline function is used by the compiler as an optimization technique to reduce the run time of the generated executable. The compiler replaces the function call statement with the function code itself and then compiles it.

To transform an inline function, the function definition in the header file is first copied into the source file. In the source file, the word “inline” is deleted and the class and namespace or only the namespace is added right before the function name.

As for the header file, there are two cases. First, if only an inline function definition exists in the header file, then the definition is reduced to a declaration and the word “inline” is removed. Second, if an inline function declaration exists in addition to a function definition, then the function definition is completely removed from the header file.

An example of the second case can be seen in Table 4. In the original header file in Table 4, line 241 has the inline function declaration and lines 306-313 have the inline function definition. As one can see in line 206 of the new header file, the function declaration inside the class is mainly preserved except for the word “inline” that is removed. In addition, the function definition is absent in the new header file.

4.5 Functions with modifiers

Functions with modifiers can actually be any type of the earlier discussed types of functions (basic, operator, template and inline). What differentiates them from the rest is the first single word or modifier preceding the function return type. The modifiers that are found in the header files include static, virtual, extern, friend and explicit. In order to transform a function with a modifier, we ignore the modifier and identify which of the 4 types of functions it is and then we transform it accordingly. Once we complete the changes, we just have to ensure that the modifier remains in the function declaration in the header file only. The modifiers are not to be placed in the source files.

	Before	After

sc_prim_channel.h	34 namespace sc_core{
	...
	194 class sc_prim_channel
	195 {
	...
	241 inline void request_update();
	...
	272 };
	...
	306 inline
	307 void
	308 sc_prim_channel::request_update()
	309 {
	310 if (! m_update_next_p){
	311 m_registry->request_update(*this);
	312 }
	313 }
	...
	352 } //namespace sc_core
		34 namespace sc_core{
		...
		194 class sc_prim_channel
		195 {
		...
		206 void request_update();
		...
		249 };
		...
		254 } //namespace sc_core

sc_prim_channel.cpp	Empty	146 void
		147 sc_core::
		148 sc_prim_channel::request_update()
		149 {
		150 if (! m_update_next_p){
		151 m_registry->request_update(*this);
		152 }
		153 }

Table 4: The transformation of an inline function with a function declaration and definition

5 Validation and Testing

After completing the transformation of the SystemC library, we tested a set of benchmark designs with our old and new SystemC libraries to check for a decrease in the number of traversed AST nodes and compilation time. To do so, we used the ROSE compiler [6], specifically a ROSE translator. The ROSE compiler is an open source compiler infrastructure that is used to build source-to-source program transformations and analysis tools for large-scale language applications such as C++.

Listing 1: Source code for a ROSE-based identity translator

```
1 #include <rose.h>
2 int main (int argc, char** argv)
3 {
4   // Build the AST used by ROSE (frontend)
5   SgProject* project = frontend(argc, argv);
6
7   // Generate source code from AST (backend)
8   return backend(project);
9 }
```

5.1 Translator modifications

A translator is a ROSE tool that is simply used in place of one's default compiler for example, g++ or gcc. The translator source file translator.cpp consists of three parts: frontend, transformation, and backend.

First, the frontend, a function provided by the ROSE infrastructure, creates the abstract syntax tree (AST) from the input.cpp file. An abstract syntax tree is basically a flow chart or tree representation of the syntactic structure of the source code in a given file. Second, the user inserts some user-defined transformations. Third, the backend, also a function provided by the ROSE infrastructure, generates a source file from the AST.

To test out the translator, we first implemented a basic identity translator. The source code for the identity translator is provided in Listing 1. The identity translator, just like any other translator, has a frontend and

backend. However, it does not perform any transformations, for it is conserving the identity of the source code in the input file as its name suggests.

After the identity translator successfully compiled the input source file, we moved on to implementing two user defined transformations. The transformations included a traversal to count nodes and a clock to time the compilation.

5.1.1 Translator User-Defined Transformations

- Traversals to count the nodes:
  The first addition to our translator was the implementation of a counter to count the total number of traversed nodes in an AST. We were not able to produce a visual representation of the AST, for the number of nodes generated exceeded the maximum number of nodes that could be displayed in the interactive viewer. On the other hand, we were still able to access the number of produced nodes.

  We implemented a visit function in the VisitorTraversal class to override the protected member function of AstSimpleProcessing, a built-in class that was made public to VisitorTraversal. The implemented visit function contains a counter that increments every time a new node is visited.

- Clock to time the compilation:
  The second addition was a clock that was used to time how long the AST generation (frontend) would take in seconds. The duration of the compilation was measured using the std::clock( ) function that belongs to the standard template library. This function returned the approximate processor time in seconds used since the beginning of an implementation-defined era related to the execution of the program. In our translator, two time recordings were made: once before executing the frontend and once after. We found the delta in time from these two recordings and converted it to seconds. That resulted in the measured compilation time.

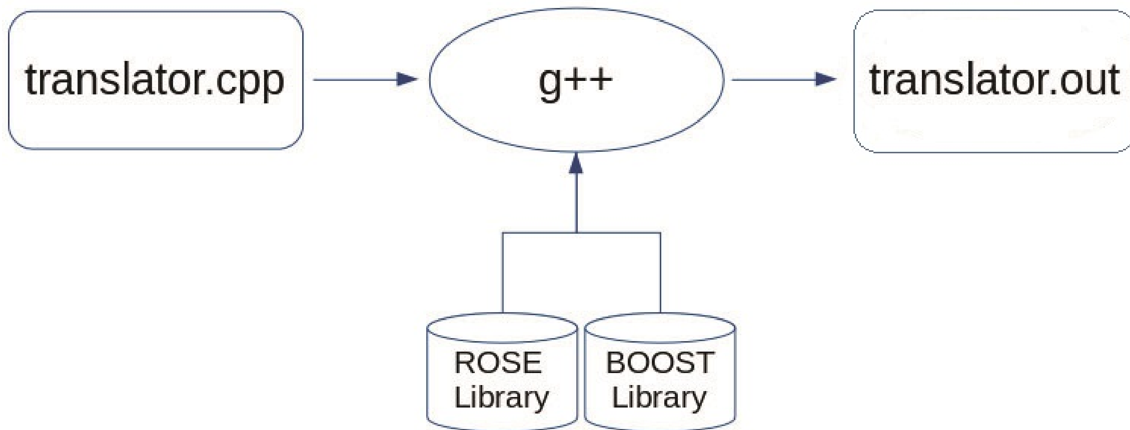

Figure 2: Compilation of translator source file with ROSE and Boost libraries

5.2 Boost

Boost is a set of libraries for the C++ programming language that provides support for specific tasks and structures. Boost was one of the software dependencies that had to be installed to provide C++ portable

runtime features. ROSE and/or software used by ROSE requires the following Boost libraries: chrono, date_time, filesystem, iostreams, program_options, random, regex, signals, system, thread, and wave.

As seen in Figure 2, the Boost and ROSE libraries were used to compile the translator source file and to generate the executable translator.

5.3 S2C Testbench Design

After completing the translator, we decided to test our translator on the S2CBench v.1.1 design models [7]. The S2CBench v.1.1 provides 16 programs written in synthesizable SystemC language. Each benchmark is designed for specific domains, such as multimedia, digital signal processing, security, image processing, etc. These different designs allow users to analyze their innovative algorithms and techniques and test the quality of their results. We used this benchmark set to find the number of AST nodes traversed and the compilation time with the original SystemC library and then the light weight SystemC library.

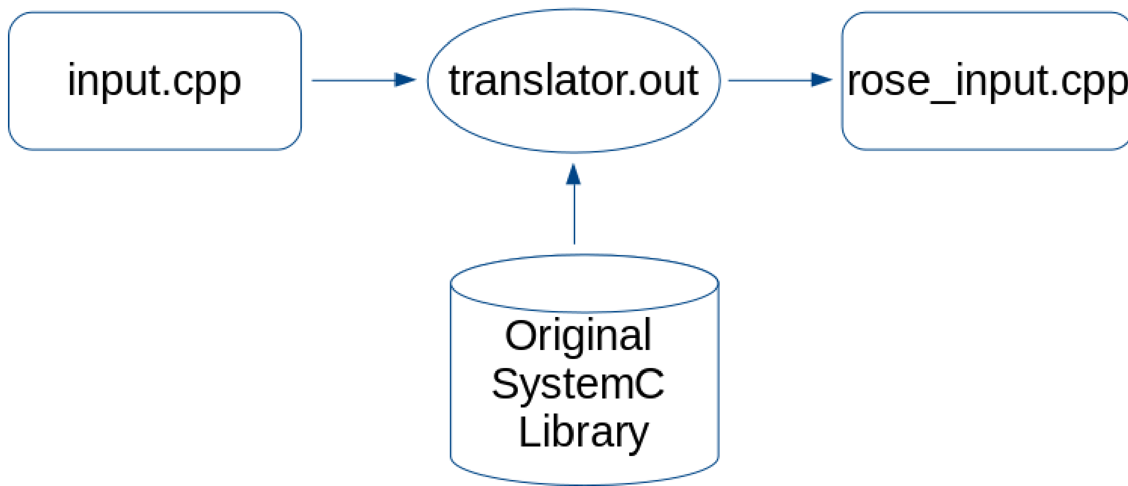

Figure 3: Compilation of input designs through ROSE compiler with light weight SystemC library

5.4 Script file

We used a script file which consists of a series of nested loops in order to facilitate executing the program multiple times. This was especially useful for timing compilation because we ran 17 test designs 5 times each. Knowing that other tasks may have been active on the same computer, it was important to run each design several times to ensure the consistency and accuracy of our results. Each of the 17 test designs was run once for the node count and 5 times for compilation timing with both the original and light weight SystemC libraries.

Figures 3 and 4 show how the input files, otherwise known as test designs, were compiled through the ROSE translator (translator.out) with the original and light weight SystemC libraries, respectively. In total, each design was run 2 times for the node count and 10 times for compilation timing.

6 Experiments and Results

After compiling the test designs with the ROSE compiler, we obtained the data displayed in the column graphs in Figure 5 and Figure 7.

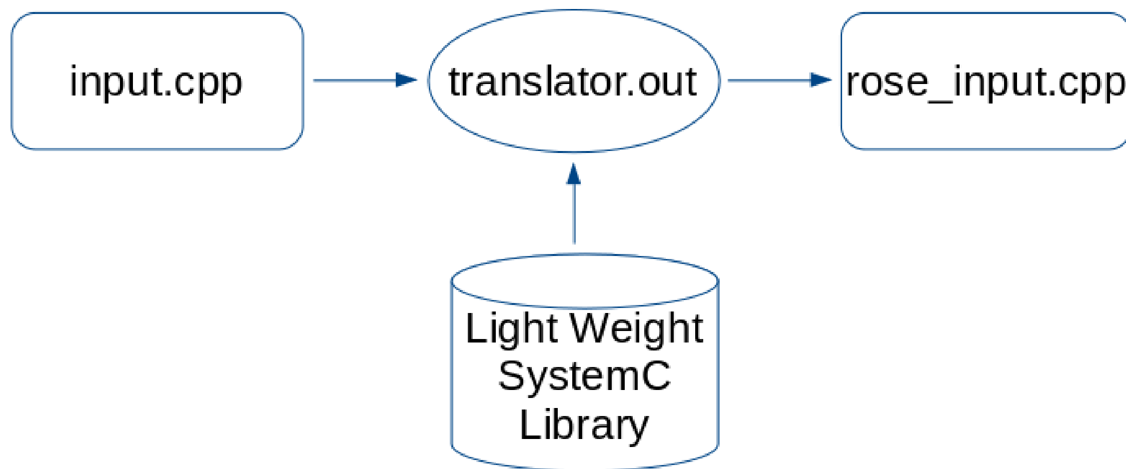

Figure 4: Compilation of input designs through ROSE compiler with original SystemC library

6.1 Compilation Time Results

Figure 5 compares the time needed to compile the test designs with the original SystemC library and the light weight SystemC library. The test design with the shortest compilation time was test_systemc. When compiled with the original SystemC library, the average measured compilation time was 12.82s in comparison to 10.37s with the light weight SystemC library. We witnessed a decrease in 2.45s which was the greatest relative decrease in compilation time among the test designs. The relative decrease, as displayed in Figure 6, was 19.14%.

The test design with the longest compilation time was test_interpolation. When this design was compiled with the original SystemC library, it took an average of 25.43s. However, when it was compiled with the light weight SystemC library, it took an average of 21.77s. That amounted to a total decrease of 3.67s. This design turned out to have the least relative decrease in compilation time which was 14.41%.

6.2 AST Size Results

As for Figure 7, it compares the number of AST nodes traversed in each test design with both the original SystemC library and the light weight SystemC library. The test design that appeared to have the smallest number of traversed nodes was also test_systemc. The number of traversed nodes when compiled with the original SystemC library was 176,655 nodes. In contrast, the number of traversed nodes when compiled with the light weight SystemC library was 135,559. That resulted in a total decrease of 41,096 nodes which is the equivalent of a 23.26% decrease in the original number of nodes, as seen in Figure 8.

The test design with the largest number of traversed AST nodes was test_interpolation. When this test design was compiled with the original SystemC library, it traversed a total of 362,245 nodes. When it was compiled with the light weight SystemC library, it traversed a total of 310,382 nodes. The total decrease in the number of traversed nodes equated 51,863 nodes which is a 14.31% decrease in the original number of nodes.

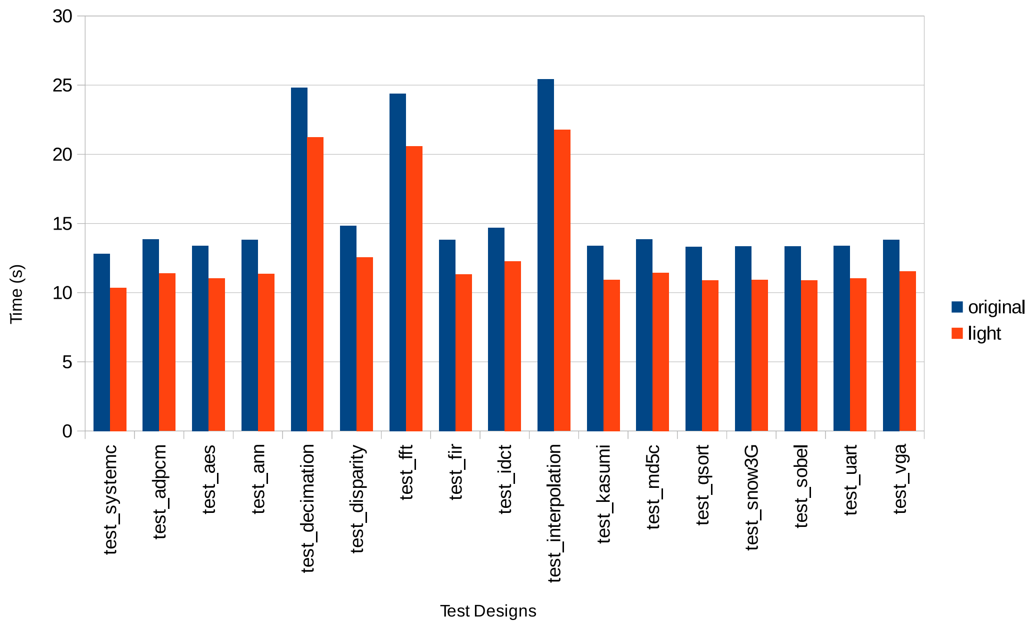

Figure 5: Compilation time of test designs with the original and light weight SystemC libraries

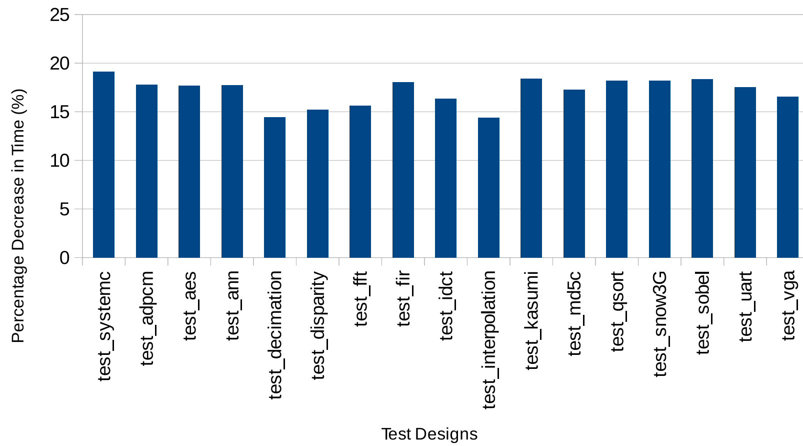

Figure 6: Relative decrease in compilation time for test designs

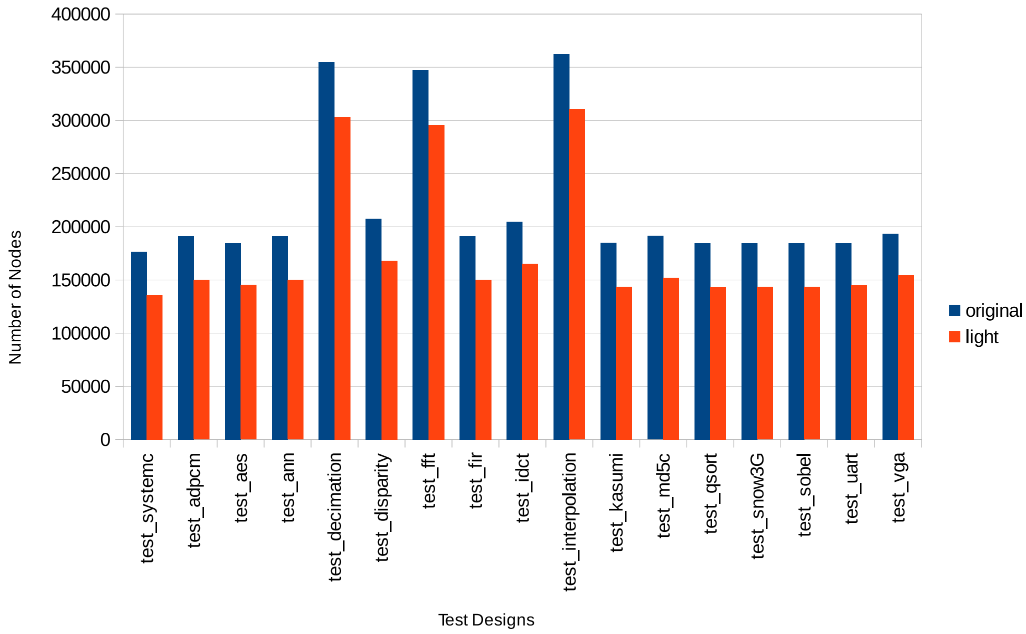

Figure 7: Number of AST nodes traversed in test designs with the original and light weight SystemC libraries

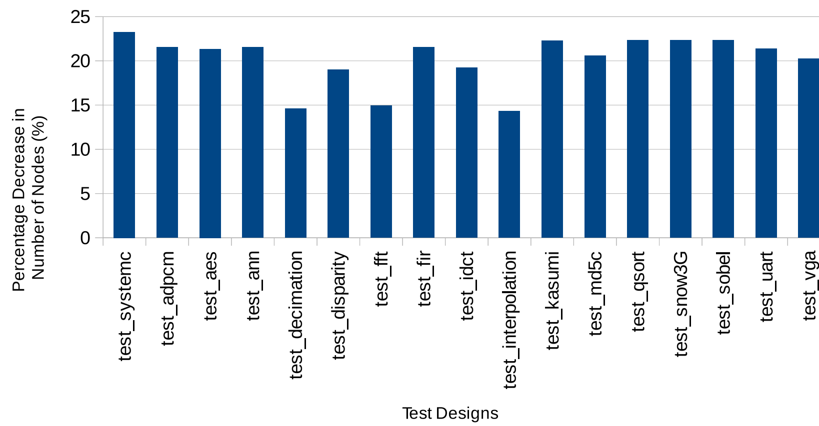

Figure 8: Relative decrease in number of AST nodes for test designs

6.3 Evaluation

To summarize our results, we obtained a percentage decrease in compilation time ranging between 14.41% and 19.14% for all the test designs. We also obtained a percentage decrease in the number of traversed AST nodes ranging between 14.31% and 23.26%.

From our results, we were able to deduce the following. The obvious conclusion is that as the number of traversed nodes in a test design increases, the time to compile it increases as well. We can also deduce that the percentage decrease in the number of nodes is inversely proportional to the number of traversed nodes in a test design compiled with either of the original or light weight SystemC libraries. Most of all, we note that our efforts of creating a light weight SystemC library with faster compilation time were successful.

7 Conclusion

After transforming the SystemC library and testing it using a ROSE compiler, we were able to verify the optimization of the SystemC library. Our results show that the time needed to compile an input design was shortened. Similarly, the improved light weight SystemC library will shorten the pre-processing time needed to compile a parallel C++ model and create a parallel executable for the *Parallel SystemC Simulation on Many-Core Architectures* project.

References

[1] R. Dömer, G. Liu, and T. Schmidt. Parallel SystemC Simulation on Many-Core Architectures. http://www.cecs.uci.edu/~doemer/risc.html.

[2] R. Dömer, G. Liu, and T. Schmidt. Parallel Simulation. In *Handbook of Hardware/Software Codesign by S. Ha and J. Teich Springer*, August 2016.

[3] Accellera Systems Initiative. SystemC Language Working Group: SystemC 2.3.1, Core SystemC Language and Examples. http://accellera.org/downloads/standards/systemc.

[4] IEEE Computer Society. *IEEE Standard 1666-2011 for Standard SystemC Language Reference Manual.* IEEE, New York, USA, 2011.

[5] W. Chen, X. Han, and R. Dömer. Out-of-Order Parallel Simulation for ESL Design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2012.

[6] D. J. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters.*, 10(2/3):215–226, 2000.

[7] B. C. Schäfer and A. Mahapatra. S2CBench: Synthesizable Systemc Benchmark Suite for High-Level Synthesis. *Embedded Systems Letters*, 6(3):53–56, 2014.