Speeding Up SystemC Simulation through Process Splitting

Youssef N. Naguib and Rafik S. Guindi
Electronics and Communication Engineering, Cairo University, Giza, Egypt
ynaguib@ieee.org, rguindi@ieee.org

Abstract

This paper presents a new approach that can be used to speed up SystemC simulations by automatically optimizing the model for simulation. The work addresses the inefficiency of the standard SystemC scheduler that may lead in some situations to unnecessary wake-up calls, as well as unnecessary code execution. The method presented analyzes the SystemC code to automatically extract signal dependencies based on a set of rules. This information is then used to split large processes into smaller ones. Process splitting is performed by a tool – SplitPro- which generates an optimized code that can be run on any standard SystemC engine. SplitPro was used to analyze the description of an Alpha super scalar processor and optimize some of its modules. A speed gain of up to 23% in simulation time was achieved over a number of split processes.

I. INTRODUCTION

Simulation of digital systems is usually performed in two phases. The first phase is the translation of the given system to some internal data structure understandable by the simulation kernel; the second phase is the scheduling of the processes describing the behavior of the system [6]. The translation phase uses efficient techniques and data structures to describe the system to be simulated. This phase may also transform the given system to a simpler system behaving identically like the original one. The scheduling phase uses the previously created data structure to evaluate signal values at different nodes in the system. When an input to a system changes, the scheduler is responsible for determining how the internal and external nodes of the system will change with time. In some cases, a change in a system input will not result in any changes in the system output. In such cases, an efficient scheduler will not go through all the unnecessary calculations of reevaluating the node values [6]. A scheduler may skip unnecessary process calls or code execution by analyzing the system to be simulated and figure out the dependency relations between signals of a given module. Using this approach a scheduler will be able to decide if it is necessary to evaluate some outputs or not. This optimization can be unnecessary if the code of the system to be simulated is written very efficiently.

One technique that is used in the translation phase converts the behavior of the system to be simulated to a compiled form, i.e. the system behavior is translated to code that can be executed to evaluate the outputs of a system when the inputs change. The scheduler should ensure that each portion of code is executed when its inputs change. The code will reevaluate the outputs according to the new inputs. This class of simulators is called compiled simulators which differ from ordinary interpreted simulators that use an engine that acts on data structures. SystemC falls in the category of compiled simulators.

In this work, we address two problems that result in slowing down the simulation speed of SystemC:

1-The unnecessary wake-up calls for SystemC processes. This includes any call for a process that does not change the final state of the system in a given simulation clock cycle.

2-The unnecessary evaluation of outputs whose inputs did not change. This includes executing portions of code in a process that do not change the final state of the system at the end of the simulation clock cycle.

The first problem has been addressed before in [1]. The solution presented needs to explicitly specify dependency information for module I/Os. The user has to manually trace signal dependencies (i.e. which process outputs depend on which process inputs), and introduce new lines of code in the given process to communicate the dependency information to the scheduler. Since the standard SystemC definition does not understand this new syntax, the authors in [1] introduced a new SystemC implementation (called FastSysC) with non-standard definition.

In this paper we present a new approach for handling unnecessary wake-up calls, while at the same time solving the problem of unnecessary evaluation of outputs. The proposed approach uses process splitting to solve the above mentioned problems. In Section II, we present an overview of simulation in SystemC 2.1 and outline the inefficiency in the scheduling algorithm. Section III describes the solution presented in [1]. Section IV presents the idea of process splitting as a remedy for the inefficiency in the SystemC 2.1 scheduler. Section V

978-3-9810801-2-4/DATE07 © 2007 EDAA

introduces a new tool called SplitPro, and describes the rules used in the tool to construct signal dependency trees. Experimental results and comparisons of performance between original SystemC engine and FastSysC with and without process splitting are shown in section VI. Section VII ends with the conclusion.

II. SIMULATION IN SYSTEMC

The scheduler in the SystemC engine is responsible for executing the functionality part associated with each module. The functionality of each module is wrapped in processes which are sensitive to a given set of signals. When the value of a signal is changed, an event is triggered; this event will resume the execution of all the processes which are sensitive to this signal [2]. The SystemC 2.1 language reference manual [3] proposes a scheduling scheme that can be used to implement the SystemC scheduler. This scheduling scheme can be changed as long as the changes are transparent for any given system running on the engine or on the SystemC 2.1 reference simulator.

A process in a module can be registered with the scheduler as being sensitive to a set of signals. Whenever a signal is changed, a request to update the value of this signal is added to the *set of update requests*. After the signal is updated all the processes sensitive to this signal are moved to a set called the *set of runnable processes*. This set represents all the processes that should be executed simultaneously by the SystemC engine. The order in which processes are run is undefined [3]. Moreover, the scheduler does not try to find any optimal order to run the processes. When a process updates a signal it will add a request to the set of update requests, which in turn may add a new process to the set of runnable processes and so on. This is called a delta cycle. A delta cycle can be thought of as a very small step of time within the simulation, which does not increase the user-visible time. Multiple delta cycles can occur at a given time point. When a signal update occurs, other processes do not see the newly updated value until the next delta cycle.

The inefficiency in SystemC Scheduler

Because processes are run directly from the set of runnable processes, in some cases unnecessary execution of processes from the set of runnable processes may take place. For a system as the one shown in Figure 1, process *W* writes to the signal *Wout*. Processes *X* and *Y* are sensitive to this signal so they are added to the set of runnable processes when *Wout* changes. A cycle appears between the two processes *X* and *Y*: *X* writes to *Xout1* to which *Y* is sensitive, and *Y* writes to *Yout* to which *X* is sensitive.

It may appear that *X* will keep on waking-up process *Y* through *Xout1* and then process *Y* will wake-up *X* through *Yout* and so on. However, this should not happen as it is an

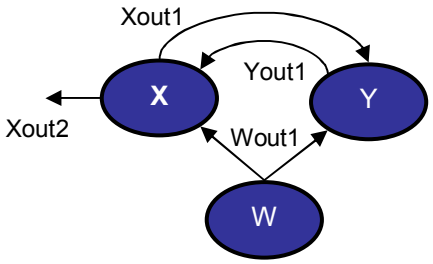

Figure 1. A system with two codependent processes

unacceptable behavior for most systems. What will happen is that after some cycles *Xout1* and *Yout* will reach a stable state and will not be changed any further. In this case *X* and *Y* will be removed from the set of runnable processes and simulation will continue.

TABLE I
DELTA CYCLES DETAILS OF THE SYSTEM IN FIGURE 1

Delta Cycle No	Set of runnable processes	Set of update requests
1	X,Y	Xout1(final), Xout2(transit), Yout (transit)
2	X,Y	Xout1(final), Xout2(transit), Yout (final)
3	X	Xout1(final), Xout2(final)

Let us assume that *Xout1* is a function of *Wout*, *Xout2* is a function of *Yout*, and *Yout* is a function of *Xout1* and *Wout*. When W updates the signal *Wout*, both processes *X* and *Y* are added to the set of runnable processes. According to our assumptions on signal dependencies, the execution of processes will follow the following sequence:

- When *X* is executed in the first delta cycle, it will write the final value of *Xout1* and a transit value in *Xout2*. *Y* is executed in the same delta cycle, however it will not see the updated value of *Xout1* until the following delta cycle. Therefore, *Y* will write a transit value to *Yout* causing *X* to be called again in the second delta cycle.
- In the second delta cycle *Y* will write the final value for *Yout*, and *X* will run again writing a final value in *Xout1* and a transit value in *Xout2*.
- In the third delta cycle, *X* will write the final value in *Xout2*.

It can be noticed from Table I that there was no need to execute *Y* in the first delta cycle or *X* in the second delta cycle. This is the problem of unnecessary wake-up calls. Moreover, there is no need to update *Xout2* and *Yout* in first delta cycle or *Xout1* in the third delta cycle. This is the problem of unnecessary evaluation of process outputs (when *X* is executed, both *Xout1* and *Xout2* are evaluated each time which is not needed).

III. FastSysC Engine

The problem mentioned in section II was outlined before in [1]. The authors presented the FastSysC engine that overcomes the problem of unnecessary wake-ups of processes. The FastSysC engine also contains a number of other refinements that help in speeding up simulation.

A developer describing a system for the FastSysC engine must provide signal dependency information. This dependency information must be written inside the SystemC code of the described system. When the scheduler runs, it knows which processes depend on which from the signal dependency information and unnecessary processes executions are skipped.

FastSysC works well in avoiding unnecessary wake-up calls. It suffers however from the following drawbacks:

1. The user must manually trace and extract signal dependencies,
2. The user must update the original code with the dependency information,
3. The augmented code is in non-standard SystemC notation and therefore will only run using a special SystemC implementation (FastSysC).

Although the FastSysC engine showed faster simulations than the SystemC 2.1 reference simulator, it still suffers from the problem of unnecessary code evaluations as can be seen from Table II.

TABLE II
DELTA CYCLES DETAILS OF THE SYSTEM IN FIGURE 1 RUNNING ON FASTSYSC ENGINE

Delta Cycle No	Set of runnable processes	Set of update requests
1	X	Xout1(final), Xout2(transit),
2	Y	Yout (final)
3	X	Xout1(final), Xout2(final)

IV. Process Splitting

In this work, we propose a solution to the problems of unnecessary wake-up calls as well as unnecessary evaluation of processes outputs. The proposed solution is based on process splitting. In order to split a process, the I/Os should allow for the splitting of the process into two or more processes which, together, will behave exactly like the same original one.

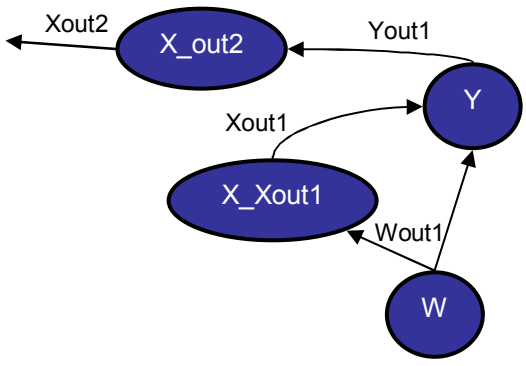

Figure 2. The system of Figure 1 after applying process splitting

Assume process P has a set of inputs I to which it is sensitive and a set of outputs O. Process P can be split if and only if there exists an element O_j in O which is function of a subset I_k of I. For the splitting to break a cycle or to introduce an improvement, I_k should not be equal to I, otherwise the splitting will result in adding the overhead of scheduling a new process.

In order to split process P, first the set of inputs I to this process and the set of outputs O are identified. A dependency tree is built for each element in the output set O. The inputs I are then searched one by one in this dependency tree to find the subset of inputs which affects the evaluation of the output.

When an output O_j of a process P is found to be function in a subset I_K of the input set I, a new process P_O_j is created with an input set I_K and an output P_O_j. The process P_O_j will only contain code that evaluates the output O_j.

These steps are repeated for each output so that any given process is broken into the largest possible number of smaller processes, each new process containing only code that evaluates its own output. If we apply the above on the simple system in Figure 1 with the dependency relation mentioned in Section II, the output system will look like the one in Figure 2. Note how the cycle disappeared from the system which can now be run on any SystemC engine.

TABLE III
DELTA CYCLES DETAILS OF THE SYSTEM IN FIGURE 2 RUNNING ON THE SYSTEMC 2.1 REFERENCE SIMULATOR

Delta Cycle No	Set of runnable processes	Set of update requests
1	X_out1,Y	Xout1(final),Yout (transit)
2	X_out2,Y	Xout2(transit), Yout (final)
3	X_out2	Xout2(final)

If this system is run on a SystemC engine with a scheduling scheme as the one proposed in the SystemC 2.1 language reference, the execution of processes will produce the update requests shown in table III. The unnecessary output evaluations have been removed but unnecessary calls to processes still exist. It is however easy now to determine the order of process execution based on the connectivity of the modules. The SystemC engine can examine the connectivity of modules and figure out the optimum order of process execution. The processes are ranked according to their place in the dependency graph. Whenever a process that depends on

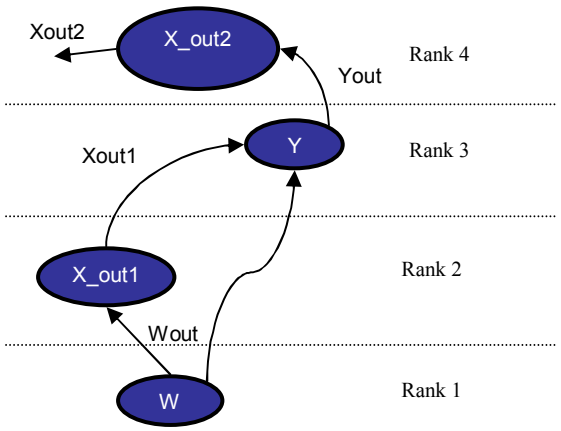

Figure 3. The dependency graph of processes

another process is being scheduled in the same delta cycle the process with higher rank will not be called.

Figure 3 shows the dependency graph of the system we are using as our example. At the beginning of each delta cycle, dependency check is applied. The scheduler will find in the first delta cycle that processes *Y* and *X_out1* are scheduled in the same delta cycle. *Y* has a higher rank than *X_out1*, so process *Y* will not be called. The same scenario will take place in the second delta cycle where *X_out2* will not be called. Table VI demonstrates the execution of delta cycles for the system of Figure 2 when run on a SystemC engine aware of the dependency between processes.

TABLE IV
DELTA CYCLES DETAILS OF THE SYSTEM IN FIGURE 2 RUNNING ON A SYSTEMC DEPENDENCY AWARE SIMULATOR

Delta Cycle No	Set of runnable processes	Set of update requests
1	X_out1,	Xout1(final)
2	Y	Yout (final)
3	X_out2	Xout2(final)

Figuring the process dependency graph as well as signals dependency graph is not always straight forward. In some cases extracting dependencies may fail. In such scenarios processes are not split and no process calls are skipped. Moreover, it is not necessary to generate one process dependency tree for the whole system. Dependency trees can be built in an incremental way where the start is a group of disconnected trees and, whenever possible, the trees are connected together and ranked. The concept of process dependency presented here is very close to the concept presented in [1].

V. SPLITPRO TOOL

The concept of process splitting presented in section IV has been implemented in a tool called SplitPro, which parses a SystemC module and generates another SystemC module with the same inputs, outputs and internal signals. The SplitPro tool may split a process to a number of smaller processes if this will result in faster simulation. To do this, the tool needs to construct a dependency tree for all the outputs of the system. The dependency tree is generated by parsing the SystemC module according to the rules described next.

1-Direct Dependency:

Direct dependency is the simplest form of signal dependency. It appears in a simple assignment statement:

1. X is said to be dependant on each and every variable appearing in expr1:

```
X=expr1;
```

2- First Level of Indirect Dependency:

The 1st level of indirect dependency appears in execution control statements as follows:

1. *If statements*: X is said to be dependant on each and every variable appearing in expr1, expr2 & expr3:

```
if(expr1)
        X=expr2;
else
        X=expr3;
```

2. *For loops:* X is said to be dependant on each and every variable appearing in expr1, expr2, expr3 & expr4:

```
for(expr1;expr2;exrp3)
        X=expr4;
```

3. *While loops:* X is said to be dependant on each and every variable appearing in expr1 and expr2:

```
While(expr1)
        X=expr2;
```

4. *Do-While loops:* X is said to be dependant on each and every variable appearing in expr1 and expr2:

```
do
        X=expr2;
While(expr1)
```

5. *Switch-Case statements*: X is said to be dependant on each and every variable appearing in expr1, constant-expr2 & expr3:

```
switch ( expr1 )
        case constant-expr2 :
                X=expr3;
```

3- Second Level of Indirect Dependency:

The 2nd level of indirect dependency appears in jump statements which are controlled by an execution control statement. In such cases, X is said to be dependant on each variable appearing in the execution control statement and the execution control statement controlling jump statement. i.e. X depends on every variable in expr1, expr2, … to expr5:

```
For(expr1; expr2; expr3){
        X=expr4;
        If(expr5)
                Break or continue or
return or goto;
}
```

The SplitPro tool analyzes a given SystemC module to determine the inputs, outputs and processes of that module. It also determines which processes write to which output ports. The tool then determines which outputs are functions of which inputs. If a process can be split according to the rules mentioned in section IV, the tool generates a new module with large processes split to smaller ones. For example, assume a module with two inputs *Z* and *AbortProcess* and two outputs *X* and *Y*. Process *Pxy* updates the values of both *X* and *Y* whenever *Z* or *AbortProcess* changes. A snippet of the *Pxy* process is shown in Listing 1.

```
SC_METHOD(Pxy){
    For(i=0;i<BusSize;i++)
       Y=Z[i] * Z [i];
    X=A+B *C;
    while(i<BusSize){
       A+=Z[i++];
           if(AbortProcess)
              break;
    }
}
```

Listing 1. Process *Pxy* which evaluates both *X* and *Y*

SplitPro starts with the output signals as roots for dependency trees. Nodes are added to the tree according to the rules listed above. New nodes are added to the tree through a number of iterations over the system code until no more nodes can be added. Figure 4 shows the dependency tree that is generated for *X*. During the first iteration on the code (see Listing 1), the tool will find that *X* appears in a direct dependency statement. *A*, *B* & *C* are therefore added as children nodes to *X*. The tool will then act on *A*, *B* & *C* to find other dependencies in the same way as with *X* and so on. The resulting dependency tree shows that *X* depends on the signals *Z* and *AbortProcess*. However a similar dependency tree for *Y* will show that *Y* depends only on *Z*. This means that there is no need to reevaluate *Y* when *AbortProcess* is triggered. *Pxy* can therefore be split into two processes: *Pxy_x* which evaluates *X* and is sensitive to *Z* and *AbortProcess,* and *Pxy_y* which evaluates *Y* and is sensitive to *Z* only. Listing 2 shows process *Pxy* after splitting.

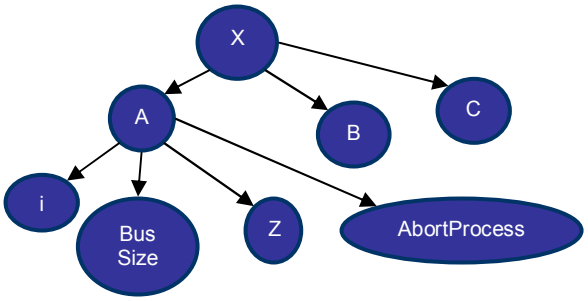

Figure 4. Dependency tree of X

```
SC_METHOD(Pxy_x){
    X=A+B *C;
    while(i<BusSize){
       A+=Z[i++];
           if(AbortProcess)
              break;
    }
}

SC_METHOD(Pxy_y){
    For(i=0;i<BusSize;i++)
       Y=Z[i] * Z [i];
}
```

Listing 2. Process *Pxy* split to *processes Pxy_x* which evaluates *X* and *Pxy_y* which evaluates *Y*.

The SplitPro tool has the following advantages:

1- SplitPro is capable of extracting the dependency information automatically
2- The generated code is in standard SystemC. It can be compiled using the SystemC 2.1 reference simulator or the FastSysC engine.
3- The tool is capable of inserting the dependency information to the system description in case this feature of the FastSysC engine will be used.
4- Process splitting opens the door to parallel simulation of SystemC modules on multiprocessor systems [4].

VI. Experimental Results

We used the same test bench used by the FastSysC team to test their engine, which is an Alpha super scalar

processor. This processor is able to execute Tru64 Alpha binaries. We randomly selected five alpha binaries to calculate their simulation time and took the average.

The Alpha processor contains 9 processes. Out of these, only 5 processes could be split using the SplitPro tool. A limitation in the SplitPro tool is that it cannot track signal dependences across more than one function or across more than one process. Of the five processes that were successfully split, four showed significant reduction in execution time. The fifth process actually increased in simulation time after splitting. This process is sensitive to four inputs and writes to four outputs. It was split into four processes, each one sensitive to two inputs and writes to a single output. A particular signal is a common input to all four outputs. Whenever this input changes, all four processes are scheduled to run which is not as efficient as evaluating the outputs in the original process. This shows that, in some cases, process splitting actually increases simulation time instead of decreasing it.

Figure 5 shows the relative gain in execution time achieved by the processes that were successfully split while boosting performance. Four different simulations were performed and compared: two on a SystemC 2.1 reference simulator (with and without process splitting), and two on the FastSysC engine (again with and without splitting).

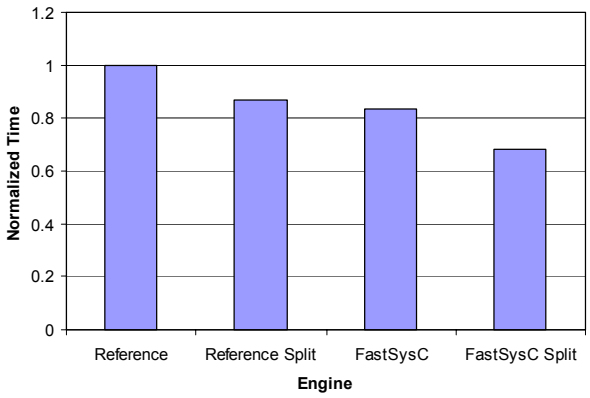

Figure 5. Simulating on SystemC 2.1 reference simulator and FastSysC Simulator with and without process splitting

Process splitting achieved a speed gain factor of 15% when run on the reference simulator engine and a speed gain factor of 23% when run on the FastSysC engine.

VII. Conclusion

In this work we have proposed a novel solution to the problem of unnecessary wake-up calls in SystemC that can manifest themselves in some situations because of the presence of I/O loops. Splitting a process into 2 or more processes can potentially break these loops and improve simulation speed. Furthermore, process splitting also addressed the issue of unnecessary code evaluation leading to further speed improvements. Signal dependencies are needed to properly split a process. The SplitPro tool automatically traces signal dependencies and generates the split processes. The generated code is in standard SystemC notation, and does not therefore require a special engine to run. When applied on a SystemC description of an Alpha super scalar processor, process splitting achieved up to 23% speed improvement on the split processes.

References

[1] Pérez, D. G., Mouchard, G., and Temam, O. 2004. A New Optimized Implementation of the SystemC Engine Using Acyclic Scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1* (February 16 - 20, 2004). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 10552.

[2] SYSTEMC: FROM THE GROUND UP, By David C. Black and Jack Donovan. Eklectic Ally, Inc.

[3] Draft Standard SystemC Language Reference Manual, April 2005.

[4] Willis, J., Li, Z., and Lin, T. 1995. Use of embedded scheduling to compile VHDL for effective parallel simulation. *In Proceedings of the Conference on European Design Automation* (Brighton, England, September 18 - 22, 1995). European Design Automation Conference. IEEE Computer Society Press, Los Alamitos, CA, 400-405.

[5] Rissa, T., Donlin, A., and Luk, W. 2005. Evaluation of SystemC Modelling of Reconfigurable Embedded Systems. *In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3* (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 253-258.

[6] Wang, Z.; Maurer, P.M., "LECSIM: a levelized event-driven compiled logic simulator," *Design Automation Conference, 1990. Proceedings. 27th ACM/IEEE* , vol., no.pp.491-496, 24-28 Jun 1990