SystemC Coding Guideline for Faster Out-of-order Parallel Discrete Event Simulation

Zhongqi Cheng, Tim Schmidt, Rainer Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine, USA

***Abstract*—IEEE SystemC is one of the most popular standards for system level design. With the Recoding Infrastructure for SystemC (RISC), a SystemC model can be executed at segment level in parallel. Although the parallel simulation is generally faster than its sequential counterpart, any data conflict among segments reduces the simulation speed significantly. In this paper, we propose for RISC users a coding guideline that increases the granularity of segments, so that the level of parallelism in the design increases and higher simulation speed becomes possible. Our experimental results show that a maximum speedup of over 6.0x is achieved on an 8-core processor, which is 1.7 times faster than parallel simulation without the coding guideline.**

I. INTRODUCTION

The IEEE SystemC standard [1] is widely used as a system level design language for specification, validation and verification of complex system-on-chip models. With the rapidly growing complexity of embedded systems, a faster simulation of SystemC models is of high demand to shorten the design cycle.

The official proof-of-concept Accellera SystemC simulator [2] is based on Discrete Event Simulation (DES), which executes the SystemC model sequentially. This means that only one thread is allowed to run at any time during the simulation. Consequently, when running the Accellera SystemC simulator on a modern multi- or many-core processor, all but one cores remain idle and the parallel computation capabilities are largely wasted.

Parallel Discrete Event Simulation (PDES) [3] has gained significant attention because it can exploit the parallel computation power of modern processors and provide faster simulation. However, regular PDES is synchronous. Earlier completed simulation threads need to wait until all the other threads have reached the same simulation cycle barrier to continue their simulation. This strict total order still imposes a limitation on high performance parallel simulation.

Out-of-order Parallel Discrete Event Simulation (OoO PDES) [4] was proposed for a better utilization of the parallel computation power. In OoO PDES, the simulation time is local to each thread, and thus the global simulation cycle barrier is removed. Independent threads can execute in parallel even if they are in different time cycles.

The Recoding Infrastructure for SystemC (RISC) [5] provides a dedicated SystemC compiler and an advanced OoO PDES simulator for SystemC. RISC is available as an open-source project and can be downloaded freely from the official website [6]. Figure 1 shows the tool flow of RISC. The RISC compiler is used as a frontend to process the input SystemC file. It statically analyzes and derives a Segment Graph (SG) representation of the model. Based on the SG, the compiler is able to analyze data conflicts and event notifications among segments, and it instruments the information as multiple lookup tables into an intermediate model. This model is then linked against the OoO PDES library to generate an executable. During the simulation, every thread executes a sequence of segments along a path over the segment graph. The simulator dynamically checks the instrumented tables to make correct thread dispatching decisions, preserving the simulation semantics and timing accuracy.

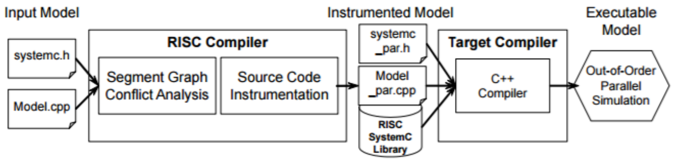


Fig. 1. RISC Compiler and Simulator for OoO PDES of SystemC [5]

A. Related Work

Various approaches have been proposed to further improve the simulation speed of OoO PDES. A segment aware thread dispatching algorithm is studied in [7]. It takes into account the execution time for a specific segment as a prediction of the next run time, so that the dispatcher more accurately predicts the run time of the thread segments ahead and makes better dispatching decisions.

In [8], the authors extended the RISC compiler with the Port Call Path (PCP) technique, which reduces false positive conflicts in the channel analysis and significantly increases the simulation speed.

PDES was also studied in [9]. The authors proposed a conservative synchronous parallel simulation approach and a SystemC framework to speedup tightly-coupled MPSoC simulations on multi-core hosts.

In [10], the authors proposed an open-source framework called systemc-clang for analyzing SystemC models with a mixture of register-transfer level and transaction-level components.

In this paper, we propose a coding guideline for SystemC users to build models with higher parallel potential that can be executed faster by the OoO PDES simulator. Specifically, the

guideline suggests for users to insert extra **wait** statements into the model, so as to increase the granularity of the SG. With the finer granularity SG, variable and event conflicts can be constrained into shorter segments, thereby reducing the time of sequential execution, which is necessary, for example, during communication between modules in the system.
Our contributions in this work are summarized as follows:
1: We propose a formal metric ψ to estimate the level of parallelism of the model under OoO PDES.
2: We propose a coding guideline for the SystemC model designers to optimize the model for faster simulation.
3: We demonstrate that the proposed coding guideline enables significant speedup of OoO PDES.

II. SG Granularity and Simulation Speed

In OoO PDES, models are simulated at segment level. As shown in Figure 2, module `M` has two sc_threads `th1` and `th2`, and a member variable `a`. `f()` and `g()` are data crunching functions which work on local variables. The corresponding SG is shown in Figure 3. Due to the data hazard over `a`, the two segments are not allowed to run in parallel. Figure 4 shows the scheduling of execution of the two sc_threads.
By inserting two new **wait** statements into the sc_threads, as shown in Figure 5, the SG becomes Figure 6. In this model, functions `f()` and `g()` are no longer in the same segment of the statements that access the shared variable `a`. Because `f()` and `g()` are conflict free, they can now be executed in parallel as shown in Figure 7, which significantly speeds up the simulation.
This leads to the conclusion that by increasing the granularity of SG, more code statements can run in parallel, and consequently increase the level of parallelism of a model and further speedup the simulation.
In the following section, we will show more details to confirm this idea and propose a coding guideline for the model designer to increase the parallel potential of the SystemC models under OoO PDES.

```
SC_MODULE(M){
    ...
    int a;
    void th1(){
        a=1;
        f();
    }

    void th2(){
        g();
        a=2;
    }
    ...
}
```

Fig. 2. Coarse Grained Source Code

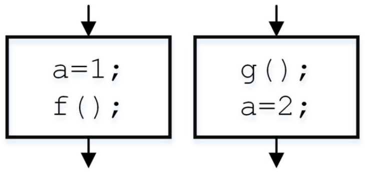


Fig. 3. SG of Fig. 2

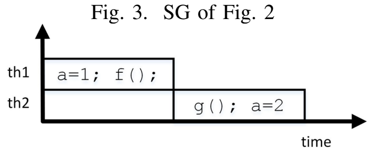


Fig. 4. Scheduling of Fig. 2

III. Recoding Infrastructure for SystemC

The fundamentals about RISC [6] are reviewed in this section for a better comprehension of the proposed coding

```
SC_MODULE(M){
    ...
    int a;
    void th1(){
        a=1;
        wait(SC_ZERO_TIME);
        f();
    }

    void th2(){
        g();
        wait(SC_ZERO_TIME);
        a=2;
    }
    ...
}
```

Fig. 5. Fine Grained Source Code

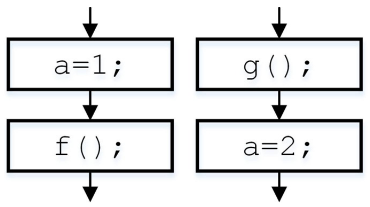


Fig. 6. SG of Fig. 5

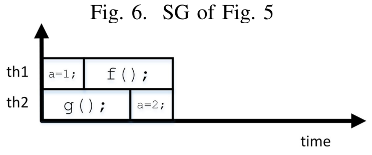


Fig. 7. Scheduling of Fig. 5

guideline.

A. Segment Graph

The SG is the foundation for both static analysis by the RISC compiler and OoO PDES by the RISC simulator. It is built on top of the Abstract Syntax Tree (AST) of the input SystemC model.
A SG is a directed graph. Each node is called a segment, which represents the code statements executed during the simulation between two scheduling steps, i.e. the entry into the simulator kernel due to a **wait** statement in SystemC. The edges in SG represent the transition between segments. An example of SystemC source code and corresponding SG is shown in Figure 8 and Figure 9.
In this example, line 8 `y++` and line 12 `s=s*s` could be

```
1   void foo(){
2       index++;
3       wait(2,SC_NS);
4       k=1;
5       if(flag){
6           x++;
7           wait(10,SC_NS);
8           y++;
9       }else{
10          a=5;
11      }
12      s=s*s;
13      wait(1,SC_NS);
14      t=s+1
15  }
```

Fig. 8. Example Source Code

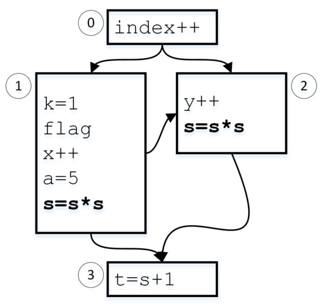


Fig. 9. SG of Fig. 8

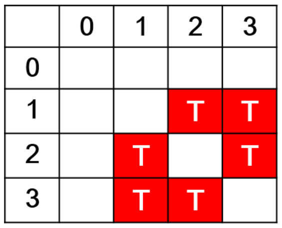
possibly executed in the same simulation cycle by a thread, so they are put both into segment2. One statement may also belong to multiple segments as it may occur on different simulation cycles. Both segment 1 and 2 have `s=s*s` in the above example. Note that a new segment starts only on **wait** statements except for the first one. The first segment is the entry point of a thread.

B. Data and Event Conflicts

The data conflict analysis takes place after the construction of the SG. It is automatically performed by the RISC compiler.

Data conflicts between segments are caused by data hazards, i.e., parallel or out-of-order accesses to shared variables. There are three types of data hazards: Read-after-write (RAW), write-after-write (WAW) and write-after-read (WAR). In the example in Figure 3, segment 1 and 2 have data conflict due to the data hazard over the variable `s`. The RISC compiler checks the data conflicts between every pair of segments, and stores the result in a Data Conflict Table (DCT). Figure 10 shows the DCT for the example in Figure 8. The red box indicates a conflict, and the blank ones mean conflict-free.

During the simulation, the OOO PDES simulator looks up the data conflict table to make safe thread dispatching decisions. If the segments of two ready-to-run threads have data conflicts, the thread with an earlier timestamp is dispatched by the scheduler. In general, segments with data conflicts are not allowed to execute in parallel.



	0	1	2	3
0				
1			T	T
2		T		T
3		T	T	

Fig. 10. Data Conflict Table for Figure 8

Event and timing conflicts are two other kinds of conflicts that are taken care of in OoO PDES. They are analyzed in a similar fashion as the data conflict. Details are described in depth in [4], but omitted here for brevity.

IV. Proposed Coding Guideline

In this section, we propose a new coding guideline for the SystemC model designers to write SystemC models with higher parallel simulation potential. Before describing the guideline, we first define a metric to estimate the level of parallelism of a SystemC model under OoO PDES.

A. Estimation for Level of Parallelism

The level of parallelism ψ is estimated as the amount of code statement pairs that can potentially execute in parallel. In OoO PDES, only code statements that belong to conflict-free segments can run in parallel, and hence our estimation is expressed as:

$$\psi = \sum_{i} \sum_{\substack{j>i \\ \text{th}_i \neq \text{th}_j}} \textsc{HasNoConflict}(\text{seg}_i, \text{seg}_j) \qquad (1)$$

Where i and j are the index of code statements in the model. seg_n is the segment that includes the n^{th} code statement. And similarly, th_i is the thread that executes the n^{th} code statement. Each single thread executes sequentially, and code statement i and j cannot execute in parallel if they belong to the same thread. $\textsc{HasNoConflict}(\text{seg}_i,\text{seg}_j)$ returns 1 if seg_i and seg_j are conflict free, otherwise it returns 0.

If two segments are in conflict, then any pair of code statements that belong to the two segments are not allowed to execute in parallel, which would reduce ψ. Thus, the larger ψ is, the higher is the parallelism level of the input model.

B. Motivation

Our idea is motivated by the following observation:
Consider we have two segments: seg_1 and seg_2, which are executed by two different threads. There are respectively p and q statements in seg_1 and seg_2. ψ for this model is simply $\psi_1 = p \times q \times \textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2)$.
Now, if a **`wait`** statement is inserted into seg_1, such that seg_1 is partitioned into two non-overlapping segments: seg_{11} and seg_{12}. After the partitioning, seg_{11} includes the first p_1 statements of seg_1, and seg_{12} includes the other $p_2 = p - p_1$ statements of seg_1. ψ for the new model becomes $\psi_2 = p_1 \times q \times \textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) + p_2 \times q \times \textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2)$. seg_{11} and seg_{12} are executed by the same thread, and hence they must run sequentially and ψ_2 does not increase.
When comparing ψ_1 and ψ_2, we get four different scenarios:

1) The conflict between seg_1 and seg_2 is only incurred by certain statements in the first p_1 statements of seg_1, and the last p_2 statements are conflict free. This indicates that $\textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) = 0$, $\textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2) = 1$ and $\textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2) = 0$. Under this scenario, $\psi_1 = 0$ and $\psi_2 = p_2 \times q$. ψ_2 is larger than ψ_1.
2) The conflict between seg_1 and seg_2 is only incurred by certain statements in the last p_2 statements of seg_1, and the other p_1 statements are conflict free. This indicates that $\textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) = 1$, $\textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2) = 0$ and $\textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2) = 0$. Under this scenario, $\psi_1 = 0$ and $\psi_2 = p_1 \times q$. ψ_2 is larger than ψ_1.
3) The conflict between seg_1 and seg_2 is incurred both by certain statements in the first p_1 statements and the other p_2 statements of seg_1. This indicates that $\textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) = 0$, $\textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2) = 0$ and $\textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2) = 0$. Under this scenario, $\psi_1 = 0$ and $\psi_2 = 0$. ψ_2 is equal to ψ_1.
4) seg_1 and seg_2 are conflict free. This indicates that $\textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) = 1$, $\textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2) = 1$ and $\textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2) = 1$. Under this scenario, $\psi_1 = p \times q$ and $\psi_2 = p_1 \times q + p_2 \times q = p \times q$. ψ_2 is equal to ψ_1.

The four scenarios suggest that

1) Partitioning a segment does not decrease the parallel potential of a model.
2) If the user carefully selects the place to insert the extra segment boundary, i.e., **`wait`** statement, ψ can be increased significantly and results in a model with higher parallelism level.

C. Overhead Consideration

One may deduce that it is always beneficial to insert as many extra **wait** statements as possible, because by doing this the ψ of the model keeps increasing. Although the deduction is correct, it is not a good practice.
Each extra **wait** statement will increase the number of segments in the segment graph by one. And the size of conflict tables is to the square of the segment count. Thus, if too many extra **wait** statements are inserted, the time cost for static analysis and dynamic checking will grow significantly, which would rather decrease the simulation performance. Besides, too many extra **wait** statements may also make the model incomprehensible.
Last but not least, each new **wait** statement creates an extra scheduler entry point into the simulator kernel which incurs significant overhead.

D. Suggestions

Motivated by the above observations and considerations, we propose the following suggestions for the SystemC model designers to properly place extra **wait** statements in the source code, so as to increase the parallel potential of the model under OoO PDES.

1) use the wait-for-delta-cycle primitive as the extra segment boundary: There are six different kinds of **wait** primitives in the SystemC standard [1]:

1) **wait**() : Wait for the sensitivity list event to occur.
2) **wait**(int) : Wait for n clock cycles in SC_CTHREAD.
3) **wait**(event) : Wait for the event mentioned as parameter to occur.
4) **wait**(double,sc_time_unit) : Wait for specified time.
5) **wait**(double,sc_time_unit, event) : Wait for specified time or event to occur.
6) **wait**(SC_ZERO_TIME): Wait for one delta cycle.

The event related **wait** primitives shall not be used because they require proper events to be notified. For the wait-for-time primitive, it is likely to change the simulation time cycle, which is not desirable. Thus, in order to maintain the semantics and timing accuracy of the original SystemC model, we suggest to the designers to use wait-for-delta-cycle primitive, i.e., **wait**(SC_ZERO_TIME) as extra segment boundaries. [1]

2) Partition the heavy segments: As mentioned in Section IV-C, the cost for one extra **wait** statement is independent of where it is inserted. Thus, in order to maximize the gain of ψ of the model, we suggest the users to partition computational intensive segments, which we refer to as *heavy segments*.
Unfortunately, it is not obvious to identify heavy segments directly from the model code. However, the RISC compiler is able to dump the statically generated SG and the DCT into files by turning on the `-risc:dump` command line option. The SG is then dumped into a `.dot` file which can be viewed graphically using the `xdot.py` tool. Also, the DCT is dumped into an HTML file which the designer can easily view in any browser. An example SystemC source code is shown in Figure 11. The dumped SG and DCT are shown in Figure 12 and Figure 13. The level of parallelism ψ for this model is $\psi 1 = 6 + 5 = 11$
From the SG, it is apparent that segment 1 and segment 3 are heavy segments which both contain loops. In order to increase the parallelism level of the model, we wish to partition the conflict-free statements from the conflicting ones in the segment, as described in the first and second scenarios in the previous section. To locate the conflicting statement, the user can refer to the dumped Data Conflict Table. In the ((1,0),(3,0)) entry of the table, it shows that the data conflict is over the variable **M::c**, and so the conflict is between statement line 8 and 17 in Figure 11[2]. In this example, the conflicting statements are not inside the computationally intensive code pieces, that are, the for loops. So we can partition the segments by inserting **wait** statements after line 9 and 18. The optimized model is shown in Figure 14. The dumped SG and DCT are shown in Figure 15 and Figure 16. Now, the level of parallelism ψ becomes $\psi 2 = 6 + 5 + 4 \times 6 = 35$. The parallel potential is further intensified during the simulation due to the two conflict free **for** loops.

V. Experiments and Results

We have applied the proposed coding guideline to several SystemC model examples. We first tested it on the synthetic benchmarks generated by the TGFF tool to validate the effectiveness of our coding guideline. Then, we evaluate the guideline with two real world designs, Canny Edge Detector and Audio/Video Decoder, to demonstrate the performance. The experiments are performed on an Intel E3-1240 host machine, which has a total of 8 cores (4 cores with 2-way hyperthreading each). The CPU frequency scaling is turned off so as to obtain repeatable results.

A. TGFF benchmarks

We first examine the performance of the proposed coding guideline on a synthetic benchmark, which is automatically generated by the TGFF tool with SystemC extension [7]. Figure 17 shows the data flow block diagram of the generated model. It has a source and a sink, and multiple parallel lanes of nodes in between. Figure 18 shows the source code for each node. Each node module first gets an input from a channel, and then does data crunching which is computationally intensive. The data crunching accesses only local variables and thus is conflict-free. After the computation the module outputs the result to another channel. In such model, data conflicts are incurred only by channel communications, which are caused by the parallel accesses to the shared variables in the channels. To optimize the model, we apply the proposed coding guideline and put **wait**(SC_ZERO_TIME) statements around the data crunching parts. The source code for the optimized module is shown in Figure 19.

[1]Note that the timing accuracy of a robust model will not be affected by extra delta cycles.

[2]The instance id is shown here, which is not of interest in this paper

```
SC_MODULE(M)
{
  int c;
  void th1()
  {
    int x = 1;
    wait(10,SC_NS);
    c = 42;

    for(int i=0;
        i<100;
        i++)  x++;
  }
  void th2()
  {
    int y = 100;
    wait(1,SC_NS);
    c = 0;

    for(int j=0;
        j<100;
        j++)  y--;
  }
  ...
}
```

Fig. 11. Source Code for Module M

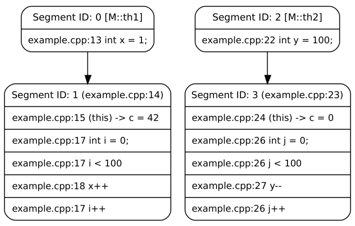
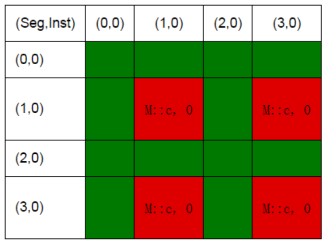

Segment ID: 0 [M::th1]
example.cpp:13 int x = 1;
Segment ID: 1 (example.cpp:14)
example.cpp:15 (this) -> c = 42
example.cpp:17 int i = 0;
example.cpp:17 i < 100
example.cpp:18 x++
example.cpp:17 i++
Segment ID: 2 [M::th2]
example.cpp:22 int y = 100;
Segment ID: 3 (example.cpp:23)
example.cpp:24 (this) -> c = 0
example.cpp:26 int j = 0;
example.cpp:26 j < 100
example.cpp:27 y--
example.cpp:26 j++

Fig. 12. SG for Figure 11



(Seg,Inst)	(0,0)	(1,0)	(2,0)	(3,0)
(0,0)	🟩	🟩	🟩	🟩
(1,0)	🟩	🟥 M::c, 0	🟩	🟥 M::c, 0
(2,0)	🟩	🟩	🟩	🟩
(3,0)	🟩	🟥 M::c, 0	🟩	🟥 M::c, 0

Fig. 13. DCT for Figure 11

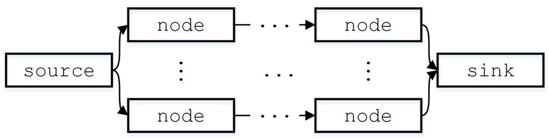


Fig. 17. Block Diagram of TGFF Models

```
SC_MODULE(Node)
{
  sc_port<input> in;
  sc_port<output> out;
  void th1()
  {
    int a=in.read();
    for(int i=0;
        i<WORKLOAD;
        i++) a++;
    out.write(a)
  }
  ...
}
```

Fig. 18. Original Source Code of Generated Testbench Model

```
SC_MODULE(Node)
{
  sc_port<input> in;
  sc_port<output> out;
  void th1()
  {
    int a=in.read();
    wait(
  SC_ZERO_TIME);
    for(int i=0;
        i<WORKLOAD;
        i++) a++;
    wait(
  SC_ZERO_TIME);
    out.write(a)
  }
  ...
}
```

Fig. 19. Optimized Source Code of Generated Testbench Model

Through a parameter to the TGFF generator, we are able to control the total number of lanes as well as nodes per lane, and each lane may consist of various number of nodes. The data crunching workload of each node is controlled by the number

```
SC_MODULE(M)
{
  int c;
  void th1()
  {
    int x = 1;
    wait(10,SC_NS);
    c = 42;
    wait(SC_ZERO_TIME);
    for(int i=0;
        i<100;
        i++)  x++;
  }
  void th2()
  {
    int y = 100;
    wait(1,SC_NS);
    c = 0;
    wait(SC_ZERO_TIME);
    for(int j=0;
        j<100;
        j++)  y--;
  }
  ...
}
```

Fig. 14. Source Code for Module M after partitioning

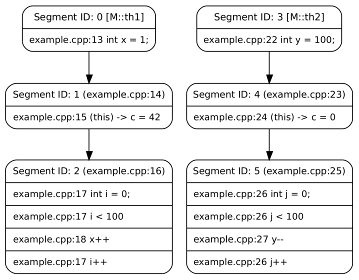
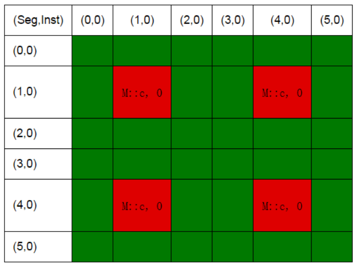

Segment ID: 0 [M::th1]
example.cpp:13 int x = 1;
Segment ID: 1 (example.cpp:14)
example.cpp:15 (this) -> c = 42
Segment ID: 2 (example.cpp:16)
example.cpp:17 int i = 0;
example.cpp:17 i < 100
example.cpp:18 x++
example.cpp:17 i++
Segment ID: 3 [M::th2]
example.cpp:22 int y = 100;
Segment ID: 4 (example.cpp:23)
example.cpp:24 (this) -> c = 0
Segment ID: 5 (example.cpp:25)
example.cpp:26 int j = 0;
example.cpp:26 j < 100
example.cpp:27 y--
example.cpp:26 j++

Fig. 15. SG for Figure 14



(Seg,Inst)	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
(0,0)	🟩	🟩	🟩	🟩	🟩	🟩
(1,0)	🟩	🟥 M::c, 0	🟩	🟩	🟥 M::c, 0	🟩
(2,0)	🟩	🟩	🟩	🟩	🟩	🟩
(3,0)	🟩	🟩	🟩	🟩	🟩	🟩
(4,0)	🟩	🟥 M::c, 0	🟩	🟩	🟥 M::c, 0	🟩
(5,0)	🟩	🟩	🟩	🟩	🟩	🟩

Fig. 16. DCT for Figure 14

TABLE I
PERFORMANCE OF TGFF BENCHMARKS, SIMULATOR RUN TIMES[SEC] AND CPU UTILIZATION

Benchmark	SEQ	PAR	GDL
1	63.55 (99%)	17.85 (377%)	10.48 (690%)
2	63.54 (99%)	17.63 (379%)	10.91 (663%)
3	134.41 (99%)	88.41 (155%)	81.55 (172%)
4	349.86 (99%)	165.41 (214%)	93.44 (400%)
5	493.02 (99%)	169.12 (301%)	99.17(552%)
6	134.40 (99%)	92.00 (155%)	81.10 (173%)
average	206.46 (99%)	91.74 (263.5%)	62.77 (441%)

of iterations of the **`for`** loop.

We studied 6 test cases with different data flow configurations in this experiment. Table I shows the performance of the simulations before and after applying the coding guideline. The first column SEQ refers to the sequential simulation with the reference Accellera SystemC simulator. Under the sequential simulation, the CPU utilization is always below 100% because only one thread is running at any time during the simulation. The second column PAR refers to the OoO PDES before applying the coding guideline. It shows that on average, the simulation of the original models is 2.3x faster than SEQ. The third column GDL refers to the OoO PDES after applying the coding guideline. It is 3.2x faster than SEQ, and 1.4x faster than PAR. For the first benchmark, GDL achieved a maximum speedup of 1.7x over PAR, and the latter one is 3.5x faster than SEQ. Note that the CPU utilization is larger than the speedup over SEQ. This is because in OoO PDES there is some overhead for checking conflict tables. The results confirm that our coding guideline can be very effective in achieving higher speedup under OoO PDES.

B. *Real world examples*

We then evaluate the proposed coding guideline with two real world examples, namely Canny Edge Detector and Audio/Video Decoder modeled similarly to the benchmarks used in [7] and [8].

1) Canny Edge Detector: Our first real world example is the Canny edge detector, which filters edges in an image. The edge detector is a structurally five-stage pipeline, and each stage has a communication-computation-communication code structure. Communication between two pipeline stages is via a user-defined channel in which the read and write functions access the shared channel variable. In this experiment, a sequence of 20 images is fed into the pipeline and correspondingly generates 20 outputs. The outputs are verified to ensure a correct simulation.

Table II shows the simulation time and CPU utilization before and after applying the coding guideline. By using the original model, a CPU utilization of 127% is achieved, which is due to the conflicts among communications. With the optimized model, the CPU utilization is increased to 149%, and the OoO PDES speed is increased by 1.2x. The speedup is not as impressive as in the TGFF test cases. This is because the workload of each pipeline stage varies greatly, and the bottleneck of the simulation speed is determined by the longest stage. However, this experiment still confirms the effectiveness of the proposed coding guideline.

TABLE II
PERFORMANCE OF CANNY EDGE DETECTOR

	SEQ	PAR	GDL
simulation time (sec)	24.85	19.96	17.23
CPU utilization	100%	127%	149%
speedup	1.00	1.24	1.44

C. *A/V decoder*

The second real world test case is an Audio/Video decoder. The model structure is shown in Figure 20. The stimulus sends the encoded stream to one video decoder and the left and right audio decoders. Then, the video decoder outputs the result to a monitor, and the audio decoders output the results to two speakers. The results for this test case are shown in Table III. The execution times cost for OoO PDES before and after applying the coding guideline are 48.24 secs and 26.67 secs, which suggest the optimized model executes 1.8x faster. The speedup is reasonable because the encoding and decoding stages have similar computation loads. The result again confirms the effectiveness of the proposed coding guideline.

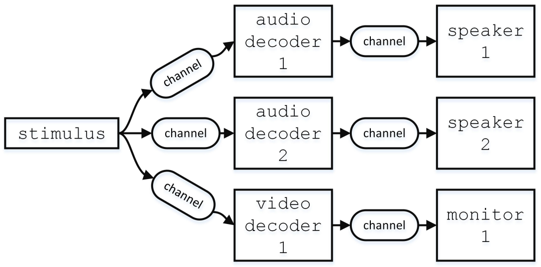


Fig. 20. Block Diagram of Audio/Video Decoder

TABLE III
PERFORMANCE OF AUDIO/VIDEO DECODER

	SEQ	PAR	GDL
simulation time (sec)	73.41	48.24	26.67
CPU utilization	100%	152%	247%
speedup	1.00	1.52	2.75

VI. CONCLUSION

In this paper, we proposed a coding guideline for the SystemC model designers who use OoO PDES parallel execution enabled by the Recoding Infrastructure for SystemC. By applying the coding guideline, the granularity of the Segment Graph becomes larger, and thus results in a faster execution speed. Our experiments show that by applying the proposed coding guideline, the optimized SystemC model is able to achieve a speedup of up to 1.7x on a 8 core machine, on top of the 3.5x speedup due to PDES.

For future work, we plan to develop a technique that automatically identifies heavy segments and applies the coding guideline automatically.

ACKNOWLEDGEMENTS

This work has been supported in part by substantial funding from Intel Corporation for the project titled ”Scaling the Recoding Infrastructure for Parallel SystemC Simulation”. The authors thank Intel Corporation for the valuable support.

REFERENCES

[1] IEEE Standard 1666-2011 for Standard SystemC® Language Reference Manual, IEEE Computer Society, January 2012.
[2] SystemC Language Working Group. SystemC 2.3.1, Core SystemC Language and Examples, Accellera Systems Initiative. [Online]. Available: http://accellera.org/downloads/standards/systemc, 2014
[3] R. Fujimoto. Parallel discrete event simulation. Commun. ACM, 33:3053, Oct. 1990.
[4] W. Chen, X. Han, C. W. Chang, G. Liu, and R. Dömer. Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 33(12):1859-1872, 2014.
[5] Dömer R., Liu G., Schmidt T. Parallel Simulation. In: Ha S., Teich J. (eds) Handbook of Hardware/Software Codesign. Springer, Dordrecht, 2016
[6] Lab for Embedded Computer Systems (LECS). Recoding Infrastructure for SystemC [Online]. Available: www.cecs.uci.edu/ doemer/risc.html#RISC042
[7] G. Liu, T. Schmidt, R. Dömer: ”A Segment-Aware Multi-Core Scheduler for SystemC PDES”, Proceedings of the International High Level Design Validation and Test Workshop, Santa Cruz, California, October 2016.
[8] T. Schmidt, Z. Cheng, R. Dömer: ”Port Call Path Sensitive Conflict Analysis for Instance-Aware Parallel SystemC Simulation”, Proceedings of Design, Automation and Test in Europe, Dresden, Germany, March 2018.
[9] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures, in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis , pp. 241246, 2010
[10] Kaushik A, Patel HD SystemC-clang: an open-source framework for analyzing mixed-abstraction SystemC models. Proceedings of the forum on specification and design languages (FDL), Paris, 2013