Removal of Unnecessary Context Switches from the SystemC Simulation Kernel for Fast VP Simulation

Kun Lu, Daniel Müller-Gritschneder and Ulf Schlichtmann
Institute for Electronic Design Automation
Technische Universität München, Munich, Germany

Abstract—Virtual prototypes are widely employed in today’s development of embedded hardware and software. To model and simulate the VPs, SystemC has been adopted as a standard language tool. With SystemC, hardware modules and software codes can be modeled as processes. To model concurrency, one process can be suspended and then the SystemC scheduler selects the next process to resume. This is also known as context switching. Context switching can consume a large portion of simulation time and decrease the simulation performance heavily. In this paper, a method is proposed to avoid unnecessary context switches. It achieves this by taking concurrency into account to determine the necessity of a context switch. The case study shows that, by applying the method, performance could be increased significantly without any loss in simulation accuracy.

I. INTRODUCTION

Virtual prototypes (VPs) have gained increasing importance in the development of embedded systems on chip (SoC). They provide an effective platform for HW/SW codesign and system verification. In developing the VPs, SystemC has gained wide acceptance as the standard modeling language [1].

Using SystemC, HW components are modeled as modules. Processes within modules represent the corresponding HW activities. The communication between these modules can be easily achieved by binding modules using interfaces or ports. Software execution on a CPU is also modeled using processes within the CPU module.

The SystemC kernel selects the process to execute using a scheduling algorithm. However, the SystemC kernel itself and the processes it manages are all tasks to the operating system of the simulation host. Only one of these tasks is running at a time, and the running one is said to have the control on the simulation host.

There are two process variants supported by SystemC: the method process and the thread process. For a method process, once selected, it completes its execution and returns control to the kernel. For a thread process, once selected, it continues its execution until it decides to suspend itself and returns control to the SystemC kernel by issuing a synchronization request. This thread process will resume from where it suspended when it regains control. SystemC uses a scheduler to select the next process to simulate. In the context of this paper, the swapping between two processes in SystemC is referred to as context switching. An example is shown in Figure 1. To complete a context switch, the SystemC kernel must first switch from the active process to the scheduler and then from the scheduler to the next process.1

Context switching is time-consuming and does not contribute to executing the code of a process. It is often considered as a major cause of low performance when simulating concurrency. The simulation time can be dominated by the time for context switching if the processes are implemented such that they issue synchronization requests too frequently. In this case the effort of context switching can become unaffordable. In standard SystemC, a context switch is performed each time a process sends a synchronization request. However, certain context switches can be avoided if the process issuing the synchronization request will be the only process to resume immediately after the context switch. For example in Figure 1, the context switch at \( t_2 \) is unnecessary for the synchronization of \( p_2 \), because only \( p_2 \) can resume immediately after this context switch.

In this paper, a method is proposed to detect the necessity for a context switch. It can be used to avoid the context switches when they are not required. To achieve this, we added new APIs of synchronization requests to the kernel. These APIs check the states of several sets that the kernel uses to control the scheduling of processes. The context switch will be performed only if the states satisfy certain conditions. Otherwise it is avoided and the process continues its execution. The gain of simulation performance depends on the implementation of the VP, namely two aspects: the impact of the context switching on the overall performance and the ratio of avoidable context switches. The efficacy of this work is verified by a case study. The results shows a factor of \( 1.2\times \sim 7\times \) increase of the simulation performance.

1It is implemented by qt (QuickThread) library in our version of SystemC, which is installed on Linux.
Meanwhile, the simulation accuracy is preserved completely. As expected, this method introduces no trade-off of accuracy for performance.

A. Related Work

To improve the performance of SystemC when simulating VPs, several types of approaches are proposed by researchers.

- Increasing the abstraction level: The most representative works use the transaction-level-modeling methodology [2], [3]. This type of approaches abstract away the concrete hardware implementation of communication protocol between two HW modules. Consequently, many context switches due to the protocol can be saved. But raising the abstraction level implies a loss of simulation accuracy.

- Host-compiled simulation: The embedded software is compiled for the simulation host CPU instead of for the target CPU [4], [5]. Using such approaches, the embedded SW is directly compiled for the simulation host. As a result, the instruction count and the data communication between the target CPU and the memory are lost. Extra effort is required to annotate timing information into the SW [6], [7].

- Unconditionally avoiding context switches: Some works in this category propose new synchronization methods between processes, such as using the time quantum [8]. A process suppresses its synchronization requests and continues execution until its time quantum is reached. There are several drawbacks of this approach. First, the order of synchronization requests can be required to assure a correct simulation of the system behaviors. But using the time quantum may lead to out of order execution of the system activities. Second, the timing accuracy may decrease, when shared resources are used. Third, it can cause timing inconsistency between modules.

In contrast to these works, the proposed method avoids the context switches without loosing accuracy. It achieves this by providing the designers new APIs, which can be considered as an enhancement of the SystemC kernel. The APIs are not limited to a specific modeling style. Therefore, they are suitable at every abstraction level and for different VP implementations. Designers can choose to use the APIs based on the implementation of the VP.

The remaining content of this paper is organized as follows: Section II analyzes the scheduling algorithm used by the SystemC kernel and illustrates cases in which its efficiency can be improved. Following this analysis, Section III describes the modification of the SystemC kernel using the proposed method. In Section IV, the case study is discussed. Section V concludes this paper.

II. PROCESS SCHEDULING OF THE SYSTEMC KERNEL

Once the SystemC kernel is running, its scheduling algorithm determines which process to execute next by examining which processes are sensitive to the most pending event. In the following, the time managed by the SystemC kernel is called simulated time. In contrast, the time spent by the simulation host to perform a simulation is called simulation time. Simulation time is a measure for simulation performance.

There are two types of sets explicitly managed by the SystemC scheduling algorithm: the set of processes to run at the current simulated time, and the set of events that control the process scheduling. Both of them are discussed in the following. Afterwards, the scheduling algorithm will be discussed.

A. Process States

As shown in Figure 2, a process can have the following states: running, waiting, runnable and expired. A process in the running state is executing on the simulation host. A process in the waiting state is waiting for certain events to occur. This process is said to be sensitive to these events. The occurrence of such events will cause this process to enter the runnable state. Depending on whether a process needs to execute when the simulation starts, it can enter the state diagram at either runnable state (through edge 1) or waiting state (through edge 2).

A process in the runnable state is ready to execute. There can be multiple processes in the runnable state. The SystemC kernel must ensure that all the processes entering the runnable state at a simulated time point must execute at that time point. Only the processes in the runnable state are tracked explicitly in the scheduling algorithm. The set of all the runnable processes is denoted as \( R \) in the following analysis.

B. Events

Being event-driven in nature, the kernel maintains two primary sets of events. It examines these two sets to determine the scheduling of the processes. The kernel decides which event should occur and be triggered. Triggering an event will cause the processes that are sensitive to this event to be added to \( R \). In the following, we describe the composition of these two sets.

- Set of timed events, denoted as: \( E_T \).

It contains the events which will occur at discrete values of the simulated time. A timed event is added to this set when a process calls either a non-zero timed-out statement (e.x.: `wait(5,SC_NS)` or a non-zero timed notification of an event (e.x.: `ev.notify(5,SC_NS)`). This set is organized as a priority queue, in which the events are prioritized according to their notification time, as in Figure 3. As a result, the events residing at the top of this queue are the next ones to occur, meaning their
occurrence time determines the next simulated time for the kernel to advance to.

- **Set of delta events: \( \mathcal{E}_D \).**
  It contains all events that should occur at the current simulated time. These events must be triggered before the kernel advances the simulated time, so that the processes sensitive to them will execute. A delta event can be caused by two types of operations:
  1. A process calling either a zero timed-out statement or a zero timed event notification.
  2. An update request of a primitive channel, such as a signal. Every time a primitive channel is written, it is updated and causes a delta event. The primitive channels that have update requests are also explicitly maintained by the kernel. This set is denoted as \( \mathcal{U} \).

**C. The Scheduling Algorithm of the SystemC Kernel**

According to [1], the simulation phase of SystemC consists of a main loop of delta cycle and advance steps, which is shown as loop 1 in Figure 4(a).

**Delta cycle**: the processes that should execute at the current simulated time are exhaustively and circularly executed.

By this definition, two delta cycles must occur at different simulated time points. One delta cycle can be further broken down into two inner loops, as loop 2 and loop 3 in Figure 4(a). The existence of loop 3 is due to the fact that a process can be added to \( \mathcal{R} \) by immediate notification. The existence of loop 2 is to process all the update requests in \( \mathcal{U} \) and simulate all processes that are sensitive to existing delta events in \( \mathcal{E}_D \).

**Advance** of the simulated time: the kernel extracts the notification time of the top events in \( \mathcal{E}_T \). Let this time be \( t_{top} \). It is the occurrence time of the next event. Then the kernel advances the simulated time to \( t_{top} \) and adds all the processes sensitive to those top events to \( \mathcal{R} \).

To illustrate the course of the scheduling algorithm, the states of \( \mathcal{R}, \mathcal{E}_T, \mathcal{E}_D \) are dynamically examined for one iteration in Figure 5. There are several important notes on this scheduling algorithm:

1. When entering the delta cycle at the entry point, the kernel simulates all runnable processes in \( \mathcal{R} \). And \( \mathcal{R} \) can be modified by triggering events in \( \mathcal{E}_T \) and \( \mathcal{E}_D \).
2. The set \( \mathcal{E}_D \) must be empty at the exit point of the delta cycle. The events in \( \mathcal{E}_T \) can only be triggered outside a delta cycle. No delta event can be generated in the advance step.
3. The simulated time only advances forward. The occurrence time of the events in \( \mathcal{E}_D \) and \( \mathcal{E}_T \) can not be earlier than the current simulated time.

**D. Redundant Context Switches**

In the current implementation of SystemC, a process will suspend itself and enter the waiting state every time it issues a wait statement. Consequently, a wait statement always causes a context switch. Consider the scenario in Figure 6. Process \( p_1 \) suspends itself at \( t_1 \) until \( t_6 \). Process \( p_2 \) resumes after this suspension at \( t_2 \) and issues several wait statements. Context switches will be performed at \( t_2, t_3, t_4 \) respectively. However, all of them are unnecessary. The next section will discuss the mechanism to detect and avoid such unnecessary context switches.
III. MODIFICATIONS TO THE KERNEL

The purpose of the proposed method is to avoid certain unnecessary context switches. Thus, it must be able to determine the necessity of a context switch. The decision is based on the following rule $S$:

$S$: The context switch is unnecessary if the process causing it would be the only one to resume immediately after this context switch.

To apply this rule, a concept of concurrency is introduced:

Concurrence within a time interval $[t_i, t_j]$ is the number of processes that should execute. It is denoted by $C_{[t_i, t_j]}$.

For example, in Figure 6, $C_{[t_1, t_2]} = 2$, while $C_{[t_2, t_3]}, C_{[t_3, t_4]}$ and $C_{[t_4, t_5]}$ are 1.

For each wait statement of a process, the simulated time interval it needs to wait is referred to as a synchronization step. In the following, process $p_i$’s synchronization step at $t_m$ is denoted as $p_i.s_{tm}$. And this wait statement is denoted as $p_i.w_{tm}$. For example, in Figure 6, $p_2.s_{t_2} = [t_2, t_3]$.

With the above terms, the rule $S$ on a context switch due to a wait statement $S_{p_i.w_{tm}}$ can be formed as:

$$C_{p_i.s_{tm}} = 1 \Rightarrow S_{p_i.w_{tm}}$$  \hspace{1cm} (1)

If this condition holds, the current process will be the only one to resume immediately after its suspension. Even if the context switch were performed, the kernel would pass control back to this process. Thus, the context switch due to this wait statement is not required. The context switch will be performed if $C_{p_i.s_{tm}} > 1$, because there are multiple processes to execute within $p_i.s_{tm}$. The resulting synchronization using rule $S$ is said to be concurrency aware.

Rule $S$ can be illustrated using the example in Figure 5. Suppose a process $p_1$ is running at the simulated time $t_{now}$ and calls wait($\Delta t$), which can cause a timed event $e_1$ to occur at $t_{now} + \Delta t$. If $C_{p_1.s_{t_{now}}} = 1$, then it implies that $e_1$ must be at the top of $E_T$, and there exists no other process to execute in the current delta cycle. Thus, if the kernel were to resume, it would definitely pass over the loop 2 and loop 3 and trigger $e_1$. As a result, $p_1$ would resume its execution. Hence, the context switch due to $p_1.w_{t_{now}}$ is in this case unnecessary.

A. Implementation

Checking $S$ is implemented for two types of wait statements, resulting in two new APIs. The first is for timeouts such as wait($t$), where $t$ is a time variable, be it zero or non-zero. The second is for events such as wait($e$), where $e$ is an event. Other types of wait calls are not modified, such as wait($e, t$) or wait($e_1, e_2$). The details for these two APIs are discussed in the following respectively.

1) New API for Timeouts: When a process $p_i$ calls a wait on a timeout, such as wait($\Delta t$), its synchronization step $p_i.s_{t_{now}}$ is $[t_{now}, t_{now} + \Delta t]$. Two steps are performed to check if the rule $S$ is satisfied for the calling process:

A1) The simulated time $t_{next}$ for the next process is queried. This time is either the current simulated time $t_{now}$ or the notification time $t_{top}$ of the top events in $E_T$. It is $t_{now}$ if one of the following condition is satisfied:

$c_1$: There exist at least one process to execute in $R$.
$c_2$: There exists at least one delta event in $E_D$.
$c_3$: There exists at least one update request for the primitive channels, which might cause delta events.

Formally, the above conditions are expressed as:

$$R \neq \emptyset \lor E_D \neq \emptyset \lor U \neq \emptyset \Rightarrow t_{next} = t_{now} \hspace{1cm} (2)$$

To check that $c_1$ holds, the organization of the runnable process set $R$ needs to be explained. It consists of two queues $R_o$ and $R_i$, illustrated in Figure 7. For $R = \emptyset$ to hold, it must hold that $R_o = \emptyset \land R_i = \emptyset$.

If none of the above conditions is satisfied, the queried time $t_{next}$ will be $t_{top}$.

A2) Check whether $t_{next}$ resides in the synchronization step of the calling process. Since

$$t_{next} \in p_i.s_{t_{now}} \Rightarrow C_{p_i.s_{tm}} > 1 \Rightarrow \neg S_{p_i.w_{tm}}, \hspace{1cm} (3)$$

the rule $S$ is not satisfied by $p_i.s_{tm}$ and the context switch can not be avoided. Otherwise, the simulated time is advanced to $t_{now} + \Delta t$ and $p_i$ continues its execution, avoiding the context switch.

Now when a process calls wait($\Delta t$), it checks whether the context switch is unnecessary, following the two steps discussed above. The control is not yielded to the kernel and this process completes this check in place. The pseudo-code of this API is given in Algorithm 1. With this new API, the current process $p_1$ calling the wait statement sees no difference at all in terms of timing, compared to the original API. If the context switch is necessary, then the kernel handles it exactly as before. Otherwise, the simulated time is directly advanced to when $p_1$ should resume. And it appears to $p_1$ as if the

Algorithm 1 New API for wait($\Delta t$)

if $C_{p_i.s_{tm}} > 1$
    wait($\Delta t$) //do context switch
else
    advance_sim_time($\Delta t$) //avoid context switch

Fig. 7. Inner organization of the runnable process set.
context switch were already performed by the kernel. Hence, the code after this *wait* statement will execute at the correct simulated time.

2) **New API for Wait on Events:** To avoid the context switch when a process waits on an event such as `wait(e)`, three steps are performed to check if the rule *S* is satisfied:

A1) The occurrence time \( t_{noti} = t_{now} + \Delta t \) is obtained for the event. It is when the calling process is supposed to resume. In case the event is not notified at all, \( t_{noti} \) will be equal to the largest simulated time that the kernel can handle.

A2) The simulated time for the next process \( t_{next} \) is obtained in the same manner as for the timeouts.

A3) A series of conditions are checked:

- \( c_1 \): \( t_{next} < t_{noti} \), meaning concurrency is larger than 1 within the time interval \([t_{now}, t_{noti}]\).
- \( c_2 \): There exist more than one sensitive processes registered at this event.
- \( c_3 \): There exists at least one delta event in \( \mathcal{E}_D \) or at least one update request in \( \mathcal{U} \).
- \( c_4 \): The notification time for the top two events in \( \mathcal{E}_T \) are the same.

If one of the condition in step A3 is satisfied, then the current process might not be the next one to resume. Thus, rule *S* fails. If all these conditions fail, then *S* is satisfied by \( p_i.w_f \) and the context switch for this *wait* statement is avoided. In this case, the simulated time advances to \( t_{noti} \) and the notification for this event is canceled since this event is triggered in-place. The pseudo-code of this API is given in Algorithm 2.

```
Algorithm 2 New API for wait(e)
if \( c_1 \lor c_2 \lor c_3 \lor c_4 \)
    wait(e) //do context switch
else
    advance_sim_time(\( \Delta t \)) //avoid context switch
cancel(e)
```

### B. Impact on Performance

The performance improvement by applying the new API depends on two aspects of the virtual prototype.

1) The ratio of context switching time in the overall performance, denoted as \( J \).
2) The ratio of avoided context switches in the overall context switches, denoted as \( K \).

The larger \( K \) and \( J \) are, the larger the performance gain will be. Ideally, \( K = 1 \) gives the maximum gain, since all context switches are avoided. In the worst case, \( K = 0 \) will negatively affect the performance due to the overhead of concurrency check. Thus, designers should use the original wait APIs if low \( K \) is expected. For example in Figure 1, checking \( S \) for \( p_1 \) will not avoid the context switch and negatively impact the performance, while that for \( p_2 \) avoids the context switch and increases the simulation performance.

### C. Impact on Modeling

There exits an additional concern when using the new APIs. It is about the tracing mechanism in the SystemC kernel. Tracing is performed between the delta cycle and the advance step (see Figure 4). If a context switch is avoided, the control will not be returned to the scheduling algorithm. Thus, tracing will not be performed for this delta cycle. Due to this fact, if a variable of certain type (for example, 

\[ \text{int, bool} \]

is modified before the avoided context switch, then it will not be traced. If this variable is a primitive channel, then it will be traced because the context switch will not be avoided since now \( \mathcal{U} \) is not empty. Therefore, if the designer needs to trace a variable of non-primitive-channel type, then the original *wait* statement should be used, which will always suspend the process.

### IV. Case Study

The test cases in this section consider systems with different properties in terms of the context switching effort. The purpose is to test the gain and overhead of the proposed APIs. In the first test, application programs are run based on host compilation, where the effort of context switching dominates the overall performance. In the second test, an instruction set simulator (ISS) is used to simulate the target binary. In this test, the context switching effort does not dominate the overall simulation performance. In the third test, a fifo is used with various dynamics by configuring the data rates of its producer and consumer. The final test creates a worst case scenario and examines the overhead of the proposed APIs.

#### A. Test on a MPSoC Using Host Compilation

The proposed method is tested by running different SW programs on a small SystemC VP (in Figure 8) as in [9]. The application SW is executed based on host compilation [5], [10]. In this case, the SW is compiled directly for and executed by the simulation host. The CPU can issue wait calls at certain points to accumulate its timing. It also issues wait calls when accessing the bus for communication, using sockets provided in TLM 2.0 [8].

Two scenarios are tested. In the first one, only one CPU is used. In the second one, two CPUs are used. The results are given in Table I. In this table, the PV columns show the results using the *untimed* programmer’s view (PV) mode, in which the CPU continues its execution and does not call wait at all. It provides the best performance but no timing information. The
TABLE I
RESULTS OF GAIN, ACCURACY AND SAVED CONTEXT SWITCHES WHEN USING HOST-COMPILED

<table>
<thead>
<tr>
<th>Case of mem. access</th>
<th>PVT-STD Perf.(s)</th>
<th>PVT-STD # of Waits (1e6)</th>
<th>PV(untimed) Perf.(s)</th>
<th>PV(untimed) Gain</th>
<th>PVT-CW Perf.(s)</th>
<th>PVT-CW Avoided C.Sw.(%)</th>
<th>PVT-CW Gain</th>
<th>Accuracy(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.96</td>
<td>4.1</td>
<td>0.05</td>
<td>18.72</td>
<td>0.13</td>
<td>100</td>
<td>7.17</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>1.16</td>
<td>4.6</td>
<td>0.18</td>
<td>6.49</td>
<td>0.25</td>
<td>100</td>
<td>4.59</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1.54</td>
<td>5.6</td>
<td>0.44</td>
<td>3.51</td>
<td>0.51</td>
<td>100</td>
<td>3.02</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>2.14</td>
<td>7.1</td>
<td>0.86</td>
<td>2.47</td>
<td>1.23</td>
<td>100</td>
<td>1.73</td>
<td>100</td>
</tr>
</tbody>
</table>

PVT-STD columns show results using the standard SystemC kernel. The PVT-CW columns show the results using the proposed concurrency-aware synchronization scheme. The gain in column PV or PVT-CW is the performance gain over the corresponding performance in column PVT-STD. The analysis is given in the following.

1) Results of Using One CPU: A SW program that computes prime numbers is executed in this test. In PV mode, the timing information and memory accesses are annotated to the source code by analyzing the cross-compiled target binary. The CPU calls the SystemC wait function before it accesses the data memory. Different cache miss rates are used to emulate caching effect. On a cache miss, the CPU uses a blocking transaction to access data memory over the bus. The results for different memory access densities caused by different cache miss rates.

The gains in PV and PVT-CW mode are calculated by comparing the simulation performance with that in PVT-STD mode. In PV mode, no wait statements are called. Hence, the performance gain in this column can be used to examine the proportion of context switching effort in the overall simulation performance. The higher this gain is, the more simulation time is consumed by context switches in PVT-STD mode. This gain also sets the upper bound of performance gain when using the proposed concurrency-aware APIs. Using the proposed APIs, all context switches (C.Sw.) are avoided. Compared to PV mode, the performance is slowed down in PVT-CW mode, due to the overhead of the introduced APIs to check for concurrency. Since there is only one CPU executing the SW, all context switches (C.Sw.) are avoided. Compared to PV mode, the performance is slowed down in PVT-CW mode, due to the overhead of the introduced APIs to check for concurrency. Still, the performance gain in PVT-CW mode is up to 7.17× in the first case when the effort of context switching is high. With more memory accesses, the gain in PV mode decreases, indicating that the context switching effort decreases in the overall simulation performance. It is because the computation effort for constructing and performing one TLM2.0 transaction to access memory is comparable to the effort of one context switching.

2) Results of Using Two CPU: Now CPU2 is added which calculates Fibonacci numbers. The two CPUs share the bus and data memory. The execution time of CPU1 is comparable to that of CPU2. From the results, the majority of the total context switches (> 80%) are avoided. With the highest and lowest context switching effort, the gain is 3.85 and 1.77 respectively. The gain is not as significant as that for one CPU. This is because the remaining context switches still consume the performance to certain degree.

For both cases, timing accuracy is completely preserved with zero timing mismatch observed.

B. Test on a MPSoC Using Instruction Set Simulator

Now the program is cross-compiled to MIPS instructions. Then these instructions are executed by an instruction set simulator (ISS) of MIPS32-4K. An ISS simulates the instruction execution pipeline of the target CPU. Much computation effort is spend for each instruction on the fetching, decoding, execution, etc. As a result, the effort of context switching versus computation is low in this case. In such a less ideal system, the performance gain offered by the proposed method is examined in the following.

1) Results of Using One CPU: First, one ISS is used. The results are given in Table II. In PV mode, the gain is not high, since the effort of context switching is not dominating the overall simulation effort. The column K gives the proportion of avoided context switches. Using the proposed APIs for wait statements, all context switches are avoided. The performance gain is slightly lower than that in PV mode, due to the overhead of checking for concurrency. But since the effort of context switching is low in the overall performance, this overhead is also marginal (around 3%). The exact value of this overhead is given in Section IV-D. More importantly, the timing accuracy can be completely preserved. Hence, in this case, using the proposed APIs offers the performance gain close to PV mode, while preserves the timing accuracy.

2) Results of Using Two CPU: Now two ISSs to used. The purpose is to test how much the proposed APIs can negatively
affected. The performance, when the concurrency check can not avoid the context switch. To stress the proposed method, the two ISSs have same clock cycle and run same program. The results are given in Table II. First it can be seen that the gain in PV mode is low, indicating low context switching effort in the overall performance. Second, there is still performance gain in this intentionally configured least ideal case. This is because there are still avoided context switches, though there are concurrent threads in every cycle. The reason for the avoided context switches is as follows: Suppose both ISS1 and ISS2 are to execute at t1. ISS1 is chosen to run first and it executes an instruction before calling a wait for 1 cycle(dt1). This waits will cause a context switch, since the next thread to run is at t1 for ISS2. Now, still at t1, ISS2 starts to execute an instruction and calls to wait for 1 cycle. The context switch due to this wait will be avoided, since the time for the next thread is t1 + dt for ISS1 and this time is not within (t1, t1 + dt). Now the simulated time will be advanced to t1 + dt. ISS1 and ISS2 will further execute their instructions in such a manner, where each of them execute two instructions and wait once. Thus, there are still avoided context switches and the performance gain is still positive in this least optimal case.

Again, for both cases in this test, timing accuracy is completely preserved with zero timing mismatch observed.

C. Test on a fifo

A fifo example from the SystemC package is tested, where the fifo is connected by a producer and a consumer. The producer waits a random time within [1, Rp] before it writes the fifo, while the consumer waits a random time within [1, Rs] before it reads the fifo. The total number of data written is $1 \times 10^5$. The results are given in Table III. In the column ‘# of waits’, the numbers before/after the ‘+’ sign are the numbers of wait statements called by the producer/consumer respectively. Different ratios of Rp : Rs leads to different fifo dynamics. When this ratio is 1:1, concurrency exists most of the time and only a few context switches are avoided. When it is 10 : 1, the consumer reads the fifo 10 times faster than the producer. As a result, the majority context switches caused by the consumer are avoided. Same reasoning applies to ratio 1 : 10. As for the gain, it is proportional to the percent of avoided context switches.

D. Test for Worst Case

To test the overhead of checking for concurrency, we simulate $1 \times 10^7$ wait calls without running the application SW. In one case, the overhead for checking concurrency is added before each standard wait call. In the other case, that overhead is not added. Without the overhead, the simulation time is $2.34s$. With the overhead, it is $2.51s$. In this case, the overhead versus the context switching effort is around $\frac{2.51 - 2.34}{2.34} \approx \frac{1}{14}$. This means, using the proposed APIs in PVT-CW mode can slow down the simulation at most by $\frac{1}{14} \approx 7\%$ compared to the PVT-STD mode. One can also make this reasonable assumption: the propose APIs will give a positive performance gain, as long as there are more than 7% context switches avoided. In practice, there are usually more than 7% unnecessary context switches and hence performance will be improved using the proposed APIs.

V. Conclusion

The paper proposed 2 synchronization APIs to improve the simulation efficiency of SystemC. In these APIs, the concurrency is checked to determine the necessity of context switch for a wait statement. Unnecessary context switches are avoided and the timing accuracy is preserved completely at the same time. Experimental results show that in case context switching impacts the simulation performance heavily, then using the proposed APIs can increase the simulation performance significantly.

References