Event Delivery using Prediction for Faster Parallel SystemC Simulation

Zhongqi Cheng
Lab for Embedded Computer Systems
University of California, Irvine
Irvine, California, USA, 92697
e-mail: zhongqc@uci.edu

Emad Arasteh
Lab for Embedded Computer Systems
University of California, Irvine
Irvine, California, USA, 92697
e-mail: emalekza@uci.edu

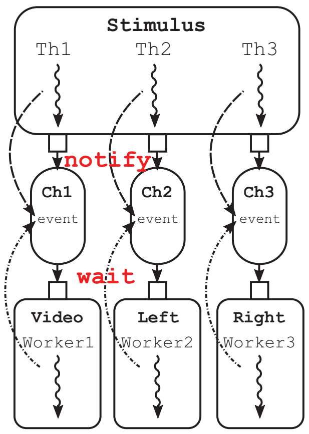
Rainer Dömer
Lab for Embedded Computer Systems
University of California, Irvine
Irvine, California, USA, 92697
e-mail: doemer@uci.edu

Abstract— Out-of-order Parallel Discrete Event Simulation (OoO PDES) is an advanced simulation approach that efficiently verifies and validates SystemC models. To preserve the simulation semantics, OoO PDES performs a conservative event delivery strategy which often postpones the execution of waiting threads due to unknown future behaviors of the model. In this paper, based on predicted behaviors of threads, we introduce a novel event delivery strategy that allows waiting threads to resume execution earlier, resulting in significantly increased simulation speed. Experimental results show that the proposed approach increases the OoO PDES simulation speed by up to 4.9x compared to the original one on a 4-core machine.

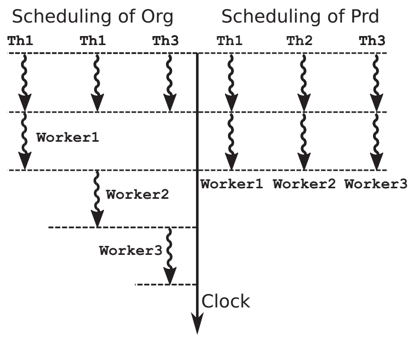
I. Introduction

In recent years, the functional complexity of embedded systems has grown dramatically. As a widely used system-level design language, IEEE SystemC [1] has been established as a de-facto and official standard for modeling and simulating embedded systems. The proof-of-concept Accellera SystemC simulator is based on Discrete Event Simulation (DES), which is sequential and does not utilize the parallel computation power of modern multi-core platforms.
Out-of-order Parallel Discrete Event Simulation (OoO PDES) [2] is studied to increase the multi-core CPU utilization. Compared to traditional Parallel Discrete Event Simulation (PDES), OoO PDES has a higher parallelism level as it allows threads to run in parallel even if they are in different cycles. Two techniques, namely static analysis and dynamic checking, are performed to preserve the simulation semantics and timing accuracy of OoO PDES.
Static analysis is a compile-time approach that analyzes the data conflicts, timing and events in a SystemC model. The analysis results are instrumented as look-up tables back into the model. Note that the event notification table with prediction (ETP) used in our work is one of the tables. During simulation, these look-up tables are dynamically checked by OoO PDES scheduler to make aggressive but safe thread dispatching decisions.
One bottleneck of current OoO PDES that limits the simulation speed is its event delivery strategy. The OoO PDES scheduler is conservative and only delivers the earliest event notifications on every scheduling step. This limits the number of threads to be waked up and and reduces the parallelism level of simulation.

As a motivating example, Figure 1a shows a simplified high-level SystemC model of a DVD player. The `Stimulus` has three parallel threads and each sends data to a corresponding channel. When the data is sent, an event is notified inside the channel for synchronization purpose. `Video`, `Left` and `Right` are three decoder modules and each has a single worker thread that waits for the corresponding channel event. When the event is delivered, the worker thread wakes up and starts processing the received data.



(a) SystemC model of a DVD player



(b) Scheduling of Org and Prd

Fig. 1: SystemC model and OoO PDES scheduling

Figure 1b shows the scheduling of threads under the original event delivery strategy (Org) and the optimized event delivery strategy using prediction (Prd) proposed in this paper. Under Org, only one worker thread can wake up each scheduling step due to the in-order event delivery strategy. While under Prd, the scheduler is able to get more context information provided by the predictions of future thread behaviors. More events are delivered every scheduling step and more threads are allowed to wake up in parallel. As a result, the simulation speed is increased significantly.

A. Related Work

Parallel Discrete Event Simulation (PDES) was first proposed in [3]. In [2], OoO PDES was introduced to further increase the simulation speed, and was well studied in [4], [5] and [6]. [4] proposed a scheduling algorithm that predicted thread run time at segment level for better multi-core scheduling. [5] exploited data-level parallelization on top of OoO PDES for faster SystemC simulation. [6] introduced a static approach to predict future behaviors of threads, and used the information for advanced data conflict analysis of threads. Although our work reuses ETP proposed in [6], it is totally different from [6] because in our work the prediction information is used for optimized event delivery strategy. Note also that the approaches in [4], [5] and [6] are orthogonal with ours and can be applied together for parallel SystemC simulation.

SystemC simulation was also widely studied in many other works. The SystemC-clang framework was proposed in [7]. It analyzed SystemC models at register-transfer level and transaction-level. [8] introduced a parallel SystemC simulation kernel. However, it required the user to manually translate the sequential design into a safe parallel design. [9] provided users with a set primitives to manually parallelize SystemC tasks. Our work is different as it supports automatic and safe simulation.

II. Background

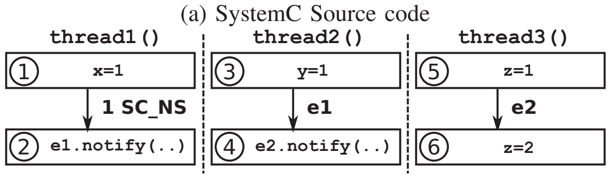
In this section, we briefly review the Segment Graph (SG) data structure which is fundamental to OoO PDES , the ETP that provides the prediction information about event notifications and the original event delivery strategy.

A. Segment Graph

The SG is a directed graph that represents behavior of functions in a SystemC model. Each node in the graph is a set of code statements executed between two scheduling steps [2]. A scheduling step is an entry to the scheduler, which includes `wait` statements, start of a thread and end of a thread. In SG, a scheduling step is indicated by an edge between segments. An example of SystemC source code is shown in Figure 2a. The corresponding SG is shown in Figure 2b. Segment 3 and 4 are separated by the wait-for-event statement `wait(e1)` in line 8, as indicated by `e1` beside the edge. Segment 4 wakes up when event `e1` is notified by segment 2.

```
1 void thread1(){    6 void thread2(){    11 void thread3(){
2   x = 1;           7   y = 1;           12   z = 1;
3   wait(1,SC_NS);   8   wait(e1);        13   wait(e2);
4   e1.notify(       9   e2.notify(       14   z = 2;
    SC_ZERO_TIME);       SC_ZERO_TIME);   15 }
5 }                 10 }
```

(a) SystemC Source code



(b) SG of 2a

Fig. 2: Example of SG

B. Event Notification Table with Prediction

ETP was first introduced in [6] for optimized data conflict analysis in OoO PDES, which is a table that stores the prediction information about the time advance for a segment to wake up another segment. ETP is formally defined in Equation 1. It is automatically built by the compiler with the algorithm proposed in [6].

$$ETP[i,j] = \begin{cases} (t_\triangle, \delta_\triangle) & \text{if a thread in } seg_i \text{ may wake up a thread in } seg_j \text{ with least time advance of } (t_\triangle, \delta_\triangle) \\ (\infty, 0) & \text{if a thread in } seg_i \text{ will never wake up a thread in } seg_j \end{cases} \quad (1)$$

Take the SG in Figure 2b as an example. Segment 4 is directly waked up by segment 2 via event `e1`, ETP[2, 4] is thus one delta cycle, denoted as (0,1) in this paper[1]. Indirect event notifications are also considered in ETP. Segment 1 does not notify any event, however, it is followed by segment 2 with time advance of `1 SC_NS` and segment 2 directly wakes up segment 4. Therefore, segment 1 indirectly wakes up segment 4 with a minimum time advance of (1,1). Segment 1 can also indirectly wake up segment 6 by first indirectly wakes up segment 4, then segment 4 directly wakes up segment 6. In this case, EPT[1,6] = (1,2). The corresponding ETP is shown in Figure 3. Note that there are several (∞,0) entries in the table. For instance, ETP(2,3) is (∞,0). This is because segment 3 does not wait for any event and thus no other segment may wake it up.

In our approach, ETP is used to calculate the predicted wake-up time of a waiting thread by another thread. Take Figure 2b as an example. Suppose `th3` is waiting and `th1` is running at timestamp (0,0) in segment 1. According to ETP, the scheduler predicts that thread 3 will probably wake up at timestamp (0,0) + (1,2) = (1,2).

Seg	1	2	3	4	5	6
1	∞,0	∞,0	∞,0	**1,1**	∞,0	**1,2**
2	∞,0	∞,0	∞,0	**0,1**	∞,0	**0,2**
3	∞,0	∞,0	∞,0	∞,0	∞,0	**0,2**
4	∞,0	∞,0	∞,0	∞,0	∞,0	**0,1**
5	∞,0	∞,0	∞,0	∞,0	∞,0	∞,0
6	∞,0	∞,0	∞,0	∞,0	∞,0	∞,0

Fig. 3: ETP for 2b

C. Original Event Delivery Strategy

We now briefly introduce the original event delivery strategy and discuss its limitations.

In OoO PDES, an event notification is not delivered immediately when notified. Instead, it is first stored into an *event notification set* Σ that keeps all event notifications during the simulation. Then, on every scheduling step, the scheduler

[1]the first element in the tuple is the time count and the second one is the delta count

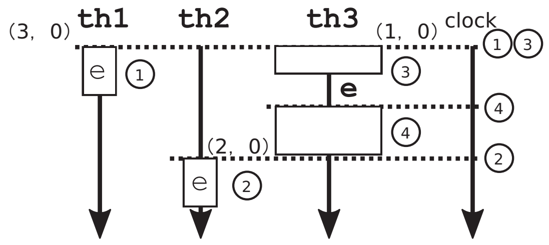
checks every event notification in Σ to determine if the event notification satisfies the following two requirments:

1) Earlier than all **RUNNING** or **READY** threads
2) Earliest among all the event notifications stored in Σ that can wake up threads

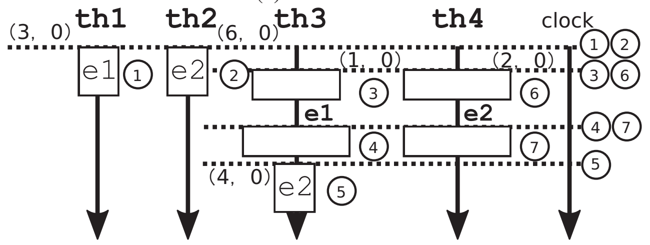
If the two requirements are fulfilled, the scheduler delivers the event notification and wakes up all the threads that are waiting on this event.

The two requirements are demanded to avoid potential *late wake-up* of threads due to unknown future behaviors of other threads. Two possible scenarios are shown in Figure 4. The rectangles represent the segments executed by threads. Note that the clock axis represents the wall clock time.

The scenario in Figure 4a explains requirement 1. Segments 1 and 2 of threads `th1` and `th2` notify event `e`. Segment 4 of thread `th3` waits for `e`. We assume that under out-of-order execution, `th1` and `th3` are scheduled to run first. Segment 1 notifies `e` at timestamp (3,0). Next, `th3` starts waiting for `e` at (1,0). Although `e` has already been notified at (3,0), the scheduler decides not to deliver it to wake up `th3`. This is because `th2` is still in **READY** at (2,0) and the scheduler cannot predict if `th2` notifies `e` before (3,0). In this example, `th2` does notify `e` at (2,0) and therefore `th3` should wake up at (2,1). Requirement 1 successfully prevents a late wake-up of `th3` at (3,1).



(a) Scenario 1



(b) Scenario 2

Fig. 4: Scenarios explaining requirement 1 and 2

The scenario in Figure 4b explains requirement 2. Two events `e1` and `e2` are notified by threads `th1` and `th2` at timestamps (3,0) and (6,0). By requirement 1, the two event notifications are not delivered because threads `th3` and `th4` are still in **READY** at earlier timestamps. Next, `th3` and `th4` are scheduled to run and then wait for `e1` at (1,0) and `e2` at (2,0) respectively. Now, both event notifications: e1 at (3,0) and e2 at (6,0) meet requirement 1 as there are no threads in **RUNNING** or **READY**. However, as specified by requirement 2, only the earliest notification: e1 at (3,0) can be delivered. Notification e2 at (6,0) is not delivered. This is because the simulator is not able to predict whether the thread waked up by `e1` will notify `e2` at an earlier timestamp before (6,0). In this example, `th3` notifies `e2` at (4,0) in segment 5 after being waked up by `e1`, and therefore `th4` should wake up at (4,1). Requirement 2 successfully prevents a late wake-up of `th4` at (6, 1).

The original event delivery strategy is very conservative and sometimes makes false decisions to not deliver an event notification. In scenario 1, if segment 2 is modified to not notify event `e`, then notification of `e` at (3,0) is actually safe to deliver immediately after `th3` starts waiting because no earlier notification of `e` will happen in the future. However, forced by requirement 1, the scheduler needs to wait until segment 2 finished execution to make this delivery. Similar for requirement 2. If the scheduler knows what will happen in the future, an event notification may be delivered earlier instead of being held until fulfilling requirements 1 and 2. Consequently, the corresponding waiting threads resume execution earlier and result in faster simulation. This motivates our idea of optimizing the event delivery strategy in OoO PDES with prediction information.

III. Optimized Event Delivery Strategy With Prediction

In this section, we first propose the optimized event delivery algorithm and then discuss about the optimization of Σ to reduce the space and time complexity.

A. Optimized Event Delivery Algorithm

Due to the out-of-order execution, a waiting thread th may be waked up by 1) event notifications that are already notified and stored in Σ or 2) event notifications that will be notified in the future by current running, ready or waiting threads. Based on this observation, we first use ETP to predict the earliest timestamp a waiting thread can wake up, and then use this information to determine if an event notification can be delivered to wake up a waiting thread. The details are shown in Algorithm 1. Note that the earliest timestamp a waiting thread th can wake up is denoted as $th.\tau$ in Algorithm 1.

The algorithm contains four steps. In the first three steps, it calculates $th.\tau$ of each waiting thread th:

1) In lines 4-8, we initialize $th.\tau$ with the earliest timestamp at which th is waked up by event notifications in Σ.
2) In lines 11-21, we update $th.\tau$ by *predicting* the earliest timestamp at which a running or ready thread th_r directly/indirectly wakes up th. Specifically, EPT is looked up according to the segment IDs of th_r and th. As discussed in Section II-B, the look-up result is the minimum timestamp advance for th_r to wake up th. By adding this result to the timestamp of th_r, we get the earliest predicted wake up timestamp of th by th_r.
3) In lines 24-38, we update $th.\tau$ by *predicting* the earliest timestamp at which another waiting thread th_k directly/indirectly wakes up th. The prediction is done the similar way as in step 2: we first look up ETP and then add the result to the timestamp of th_k. However, th_k is still waiting and does not have a valid timestamp assigned until it has been waked up. Therefore, we instead use $th_k.\tau$ as the timestamp of th_k because $th_k.\tau$ is the earliest timestamp at which th_k can wake up. At first this seems like an

Algorithm 1 Optimized Event Delivery Strategy using Prediction

```
 1: function ANALYZEANDDELIVEREVENTS
 2:     do
 3:         ▷ Step 1: event notifications affect $\tau$ of waiting threads
 4:         for all $notification \in \Sigma$ do
 5:             for all $th \in$ GETWAITINGTHREADS($notification$) do
 6:                 $th.\tau \leftarrow$ MIN($th.\tau$, GETTIMESTAMP($notification$))
 7:             end for
 8:         end for
 9:
10:         ▷ Step 2: running and ready threads affect $\tau$ of waiting threads
11:         for all $th \in$ WAITING do
12:             $segID \leftarrow$ GETSEGMENTID($th$)
13:             for all $th_r \in$ RUNNING $\cup$ READY do
14:                 $segID_r \leftarrow$ GETSEGMENTID($th_r$)
15:                 $t_r \leftarrow$ GETTIMESTAMP($th_r$)
16:                 $t_{pred} \leftarrow ETP[segID_r, segID]$
17:                 if ISVALID($t_{pred}$) then
18:                     $th.\tau \leftarrow$ MIN($th.\tau, t_r + t_{pred}$)
19:                 end if
20:             end for
21:         end for
22:
23:         ▷ Step 3: waiting threads affect $\tau$ of other waiting threads
24:         $L \leftarrow$ number of waiting threads
25:         $\mathbb{D} \leftarrow \varnothing$                    ▷ Waiting threads with determined $\tau$
26:         for $k \leftarrow 1$ to $L$ do
27:             $th_k \leftarrow$ Waiting thread that has the $k^{th}$ smallest $\tau$
28:             insert $th_k$ to $\mathbb{D}$
29:             $segID_k \leftarrow$ GETSEGMENTID($th_k$)
30:             $t_k \leftarrow th_k.\tau$
31:             for all $th \in$ WAITING do
32:                 $segID \leftarrow$ GETSEGMENTID($th$)
33:                 $t_{pred} \leftarrow ETP[segID_k, segID]$
34:                 if ISVALID($t_{pred}$) then
35:                     $th.\tau \leftarrow$ MIN($th.\tau, t_k + t_{pred}$)
36:                 end if
37:             end for
38:         end for
39:
40:         ▷ Step 4: Deliver event notifications by checking $\tau$s
41:         for all $notification \in \Sigma$ do
42:             for all $th \in$ GETWAITINGTHREADS($notification$) do
43:                 if GETTIMESTAMP($notification$) = $th.\tau$ then
44:                     DELIVERNOTIFICATIONTOTHREAD($notification, th$)
45:                 end if
46:             end for
47:         end for
48:     while no waked up thread and event notification delivered
49: end function
```

endless loop: τs are used to update τs. In fact, there exists a topological order among all the τs: τ of a waiting thread may only be updated by τs of other waiting threads with smaller values. According to this topological order, we design a loop to update τs, as shown in lines 26-38. On the k^{th} iteration, we use $th_k.\tau$ of the waiting thread th_k that has the k^{th} smallest τ among all waiting threads to update other τs and stores th_k into the set $\mathbb{D}$.

Theorem 1. At the end of the k^{th} iteration, $\forall th_D \in \mathbb{D}$, $th_D.\tau \leq th_k.\tau$.

Proof. Theorem 1 is proved by induction.
a) *Base case*: At the end of the first iteration, $\mathbb{D}$ contains only th_1. It is correct that $\forall th_D \in \mathbb{D}$, $th_D.\tau \leq th_1.\tau$.
b) *Inductive hypothesis*: Let th_{last} be the last waiting thread added to $\mathbb{D}$. Let $\mathbb{D}'$ = $\mathbb{D} \cup th_{last}$. Our I.H. is: $\forall th_{D'} \in \mathbb{D}'$, $th_{D'}.\tau \leq th_{last}.\tau$
c) *Using the I.H.*: Assume I.H. is correct on the $(k-1)^{th}$ iteration. Let th_{k-1} be the waiting thread added to $\mathbb{D}$ on the $(k-1)^{th}$ iteration, th_k be the waiting thread added to $\mathbb{D}$ on the k^{th} iteration, $\mathbb{D}_{k-1}$ be the $\mathbb{D}$ at the end of the $(k-1)^{th}$ iteration, $\mathbb{D}_k$ be the $\mathbb{D}$ at the end of the k^{th} iteration. By using the I.H., $\forall th_{D_{k-1}} \in \mathbb{D}_{k-1}$, $th_{D_{k-1}}.\tau \leq th_{k-1}.\tau$. Also, since $\mathbb{D}$ only grows in size, $\mathbb{D}_k = \mathbb{D}_{k-1} \cup th_k$.
Because on the $(k-1)^{th}$ iteration, we select th_{k-1} rather than th_k, this indicates that at the beginning of the $(k-1)^{th}$ iteration, $th_{k-1}.\tau \leq th_k.\tau$. Also, on the $(k-1)^{th}$ iteration we can only update other τs to be values larger than $th_{k-1}.\tau$ because ETP contains only positive timestamps. Therefore, $th_{k-1}.\tau \leq th_k.\tau$ still holds on the k^{th} iteration. Since 1) $th_{k-1}.\tau \leq th_k.\tau$, 2) using the I.H., $\forall th_{D_{k-1}} \in \mathbb{D}_{k-1}$, $th_{D_{k-1}}.\tau \leq th_{k-1}.\tau$, 3) $\mathbb{D}_k = \mathbb{D}_{k-1} \cup th_k$, by combining these inequalities we prove that $\forall th_{D_k} \in \mathbb{D}_k$, $th_{D_k}.\tau \leq th_k.\tau$. Therefore, the I.H. is still correct on the k^{th} iteration. □

According to Theorem 1, once a waiting thread th_k is inserted to $\mathbb{D}$, $th_k.\tau$ will not change in following iterations. Therefore, it is safe to use $th_k.\tau$ to update other τs.
In lines 41-47, Algorithm 1 checks for each waiting thread th if it can be waked up by an event notification $notification$ in Σ at $th.\tau$. If true, $notification$ is then safe to be delivered to wake up th because th is impossible to wake up any earlier.

The `do-while` loop in line 48 handles the situation where the only thread that can wake up is waiting for a `sc_event_and_list`. If this is not correctly handled, the simulation may stop early and violate the SystemC semantics. Details are not described in this paper for brevity.
Now we demonstrate Algorithm 1 using the scenario in Figure 4a. When thread `th3` starts waiting, it cannot wake up immediately though event `e` is already notified at (3,0) in segment 1 by `th1`. This is because the scheduler predicts that segment 2 will in the future notify `e` and therefore `th3`.τ is (2,1). However, if the scenario is changed such that segment 2 does not notify `e` or notifies `e` after (3,0), notification of `e` at (3,0) would be delivered immediately when `th3` enters segment 3 because `th3`.τ is now (3,1). Similar for the scenario in Figure 4b. In conclusion, Algorithm 1 helps the scheduler to deliver event notifications earlier. As a result, waiting threads can resume execution earlier.

B. Complexity Analysis and Optimization

In this section we first analyze the complexity of Algorithm 1. Let:

1) N be the number of event notifications in Σ
2) W be the number of threads in **WAITING**
3) R be the number of threads in **RUNNING** $\cup$ **READY**

The time complexity of step 1 is $N \times W$ because the two functions MIN() and GETWAITINGTHREADS() are both of constant time complexity. The time complexity of step 2 is $W \times R$ because ISVALID() and table look-up of ETP are both of constant time complexity. The time complexity of step 3 is $W \times W$ based on the implementation. GETSEGMENTID() is implemented as a table look-up function and has constant time complexity. The time complexity of step 4 is $N \times W$. Therefore, the overall time complexity of Algorithm 1 is $W \times (W + R + 2 \times N)$.
Note that W and R are fixed and specified by the SystemC model, so we can only optimize N. Because Σ stores all the event notifications since the start of simulation, its size N keeps growing dramatically and decreases the simulation speed over time. To solve the problem, we remove event

notifications from Σ which are earlier than all running or ready threads after ANALYZEANDDELIVEREVENTS() is called [2]. The optimization is safe and valid because the removed event notifications will not in the future wake up any threads. The proof is omitted here for brevity. With this optimization, N is no longer the number of event notifications since the start of simulation but the number of active event notifications. This practically speeds up Algorithm 1 and reduces space cost.

IV. Experiments and Results

We have implemented Algorithm 1 as an extension of the RISC OoO PDES simulator download from [10]. The RISC infrastructure also provides a SystemC compiler that statically analyzes an input SystemC model. ETP is automatically generated by the compiler and provided to the simulator in the instrumented SystemC model. We have evaluated our approach with synthetic examples generated by the TGFF tool and also real-world DVD decoder and GoogLeNet [11] examples. For evaluation, we have measured the execution times under the sequential Accellera simulator (Seq), the original RISC OoO PDES simulator (Org) and the RISC OoO PDES simulator with optimized event delivery strategy using prediction proposed in this work (Prd). Experiments were performed on an Intel Xeon E3-1240 multi-core processor with 4 cores, 2-way hyperthreaded. The CPU frequency-scaling was turned off so as to provide accurate and repeatable results.

A. TGFF Examples

In this experiment, we evaluate the performance of the proposed approach with synthetic examples which are automatically generated by the TGFF tool with SystemC extension [4]. Figure 5 shows the generic structure of the generated SystemC models. The model contains multiple lanes of nodes between `Stimulus` and `Monitor`. All nodes are connected by user-defined fifo channels. Each channel contains two events. Each node is a SystemC module with a single thread that first reads the data from the input port, performs some intensive computation and finally sends the result to the output port. The model is parameterized and we are able to control:

1) the number of lanes m.
2) the number of nodes per lane n.
3) the computation workload of each node w.

In this experiment, each node has random workload and each lane has the same amount of nodes. The workload of each node is determined by the iterations of a `for` loop, which varies from then thousand to one million. We generated 20 benchmarks with m varying from 1 to 16 and n varying from 2 to 16. Table I shows the run-times of the examples with Seq, Org and Prd. It also shows the speedups of Org vs. Seq and Prd vs. Seq. The speedups are shown in bold font. Table I allows the following observations:

[2] `sc_event_and_list` causes an exception here. If a thread th is waiting for a `sc_event_and_list`, the timestamp that th started waiting is also considered in the comparison

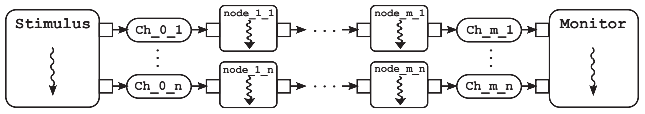


Fig. 5: SystemC model of synthetic examples

1) PRD is faster than SEQ. A maximum speed-up of 6.3x is achieved by PRD over SEQ with m=4 and n=16, which is impressive on a 4-core machine with 8 hyperthreads. Note that a naive theoretical speed-up of 8x cannot be achieved. This is because there are only four FPUs on the host processor. Due to the intensive computations in each node, the eight hyperthreads are not allowed to run fully in parallel. We also notice that the speedup slightly drops when $m = 16$ and $n = 16$. This is due to the largely increased context switching and contentions of FPUs between threads.
2) PRD is faster than ORG. The speedup is most obvious when there is only one lane in the model. ORG has no speedup against SEQ while PRD has an increasing speedup with the number of nodes. Under ORG, requirement 1 and 2 in Section II-II-C become a global barrier and forbid threads that have different timestamps to execute out-of-order. On the other hand, PRD makes correct predictions and identifies that only neighboring nodes have dependency while others are able to run out-of-order. Therefore, more events are delivered and more threads wake up in the same scheduling step, which as a result increases the execution speed of the model. Due to the hardware limitations (number of hyperthreads and FPUs), the speedup decreases with the increasing of *m*. However, PRD is still significantly faster than ORG.

The observations confirm the effectiveness of the proposed event delivery strategy using predictions. We achieve a maximum speedup of 6.3x over Accellera sequential simulation and 4.9x over the original OoO PDES, which are impressive on a 4-core machine.

B. DVD Player

In this experiment we evaluate our approach using the DVD player example that is similar to the one in Figure 1a. After decoding, the results are sent to a `Monitor` module. The communication in this model is via user-defined double-handshake channels. The results of run-times and speedups compared to the Accellera sequential simulation are shown in Table II.

In this example, the three decoding lanes are independent and are able to execute in parallel. However the original OoO PDES (Org) imposes a false global barrier such that one lane can only continue execution until the other two lanes have finished execution. The barrier reduces the parallelism level of the model and decreases the execution speed, resulting in a speedup of 2.3x over Seq. On the other hand, our proposed approach successfully predicts that the three lanes are independent and therefore decoder threads wake up out-of-order. As a result, the execution speed increases and achieves a speedup of 2.8x over Seq and 1.2x over Org. Note that

TABLE I: Results of Synthetic Examples: run-time(secs) and speedup(%)

m \ n	2					4					8					16				
	Seq	Org		Prd		Seq	Org		Prd		Seq	Org		Prd		Seq	Org		Prd	
1	76.71	76.77	**100**	52.34	**147**	119.77	119.86	**100**	53.08	**225**	176.71	176.81	**100**	54.94	**322**	355.52	355.81	**100**	72.36	**491**
2	119.80	100.16	**120**	53.11	**225**	176.69	128.62	**137**	54.77	**323**	355.50	241.22	**147**	71.51	**497**	672.03	436.20	**154**	112.16	**599**
4	176.83	101.22	**175**	55.61	**318**	355.55	161.40	**220**	72.56	**490**	672.33	276.70	**243**	113.86	**590**	1365.01	530.30	**257**	215.25	**634**
8	355.88	115.23	**309**	84.73	**420**	672.14	190.62	**353**	117.95	**570**	1364.99	353.10	**387**	216.28	**631**	2712.08	664.17	**408**	430.36	**630**
16	672.12	161.94	**415**	129.86	**518**	1364.71	290.58	**470**	219.88	**621**	2712.02	549.42	**494**	431.16	**629**	5491.15	1140.51	**481**	895.85	**613**

TABLE II: Results of DVD Player: run-time(secs) and speedup(%)

Seq	Org		Prd	
209.51	88.49	**236**	73.35	**284**

a naive maximum speedup of 3x is not achieved because the workload of `VideoDecoder` and `AudioDecoders` are different. Specifically, `VideoDecoder` takes longer to process its frames and becomes a sequential bottleneck. According to Amdahl's law, a speedup of 2.8x is reasonable. This experiment confirms the correctness and effectiveness of our proposed event delivery strategy.

C. GoogLeNet

GoogLeNet [11] is a deep convolutional neural network for image classification and detection. In this experiment, we implement a SystemC model of GoogLeNet and the details of the model can be found in [12]. Each layer is implemented in a separate module. The communications are via channels. The architecture is shown in Figure 6. Including `Stimulus` and `Monitor`, there are a total of 146 module instances in this SystemC model and each module instance has a single thread. In this experiment, 500 images are fed into the GoogLeNet SystemC model for classification. The results are verified and are correct. The experimental results of run-times and speedups compared to the Seq are shown in Table III.

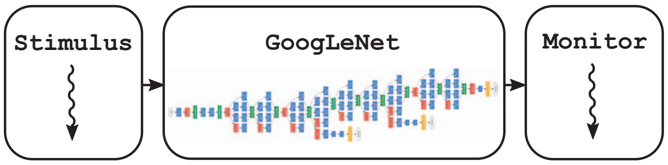


Fig. 6: SystemC model of GoogLeNet [11]

TABLE III: Results of GoogLeNet: run-time(secs) and speedup(%)

Seq	Org		Prd	
947.30	361.31	**262**	210.97	**450**

In this experiment, Prd achieves a speedup of 4.5x against Seq. A naive maximum speedup of 8x cannot be achieved. This is because the workloads of module instances (layers in GoogLeNet) are not perfectly balanced and heavy ones become sequential bottlenecks in the data flow of the model. Therefore, eight hyperthreads cannot execute totally in parallel. Also, there are only four FPUs which may introduce contentions between threads. Nevertheless, the result is still impressive on a 4-core machine. Compared to Org, Prd is 1.7x faster. This experiment confirms again that the proposed event delivery strategy using prediction is effective and correct.

V. Conclusion

In this paper, we propose an optimized event delivery strategy using prediction information for OoO PDES. Our work allows the scheduler to deliver more events each scheduling step, resulting in more threads running in parallel and increased simulation speed. We have demonstrated the effectiveness of the proposed approach with synthetic and demonstration examples. Significant and impressive speedups are achieved against Accellera sequential simulation and the original OoO PDES. In the future, we plan to further optimize the time complexity of the proposed event delivery strategy.

Acknowledgments

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] 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.

[3] R. Fujimoto, "Parallel discrete event simulation", Communications of the ACM, 33:3053, Oct. 1990.

[4] G. Liu, T. Schmidt, R. Dömer, "A Segment-Aware Multi-Core Scheduler for SystemC PDES", IEEE International High-Level Design Validation and Test Workshop, California, October 2016.

[5] T. Schmidt, G. Liu, R. Dömer, "Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation", Design Automation Conference, 2017, Austin, TX, June 2017.

[6] W. Chen, R. Dömer: "Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions", Design, Automation and Test in Europe Conference, Grenoble, France, March 2013.

[7] A. Kaushik, H.D. Patel, "SystemC-clang: an open-source framework for analyzing mixed-abstraction SystemC models", Forum on specification and Design Languages, Paris, 2013.

[8] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments", Design, Automation and Test in Europe Conference, 2016.

[9] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach", Design, Automation and Test in Europe Conference, 2013.

[10] Lab for Embedded Computer Systems (LECS), Recoding Infrastructure for SystemC [Online]. Available: http://www.cecs.uci.edu/~doemer/risc.html#RISC050.

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, "Going deeper with convolutions," Conference on Computer Vision and Pattern Recognition, Boston, MA, 2015.

[12] E. Arasteh, R. Dömer: "An Untimed SystemC Model of GoogLeNet", The 6th International Embedded Systems Symposium, 2019.