Lazy Event Prediction using Defining Trees and Schedule Bypass for Out-of-Order PDES

Daniel Mendoza, Zhongqi Cheng, Emad Arasteh, Rainer Dömer
Center for Embedded and Cyber-physical Systems
University of California, Irvine, USA

***Abstract*—Out-of-order parallel discrete event simulation (PDES) has been shown to be very effective in speeding up system design by utilizing parallel processors on multi- and many-core hosts. As the number of threads in the design model grows larger, however, the original scheduling approach does not scale. In this work, we analyze the out-of-order scheduler and identify a bottleneck with quadratic complexity in event prediction. We propose a more efficient *lazy* strategy based on *defining trees* and a *schedule bypass* with O($m \log_2 m$) complexity which shows sustained and improved performance gains in simulation of SystemC models with many processes. For models containing over 1000 processes, experimental results show simulation run time speedups of up to 90x using lazy event prediction against the original out-of-order PDES approach.**

***Index Terms*—Discrete event simulation, SystemC**

I. Introduction

As system developers desire to increase the functionality of their products, the complexity of the design process increases. Fast and accurate embedded system simulation is important to aid system developers to make and validate design decisions. However, simulating complex systems is often time consuming and can become a bottleneck in the design process. SystemC [1] is the official IEEE standard for modeling and simulating embedded systems. The Accellera proof-of-concept simulator runs simulations with only one active simulation thread at any time. With the wide-spread availability of multi-core and many-core hosts, exploitation of thread-level parallelism is an appealing way to speedup simulation time.

Out-of-Order (OoO) Parallel Discrete Event Simulation (PDES) [2] for SystemC has been shown to provide significant speedup against the reference sequential approach. One state-of-the-art OoO PDES approach [3] utilizes the Recoding Infrastructure for SystemC (RISC) compiler for static analysis of data conflicts and event notifications between SystemC processes. The RISC compiler then passes the analysis results to an OoO scheduler for dynamic decision making to enable highly parallel multi-threading.

Algorithm 1 illustrates the original OoO scheduling approach with event prediction [2]. OOOSCHEDULE executes event prediction, event delivery, and process dispatch. OOOSCHEDULE is then called whenever a process calls WAIT in order to deliver events or dispatch more threads.

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."

In order to run a simulation in OoO fashion, the scheduler must perform event prediction, which involves predicting the simulated earliest wake up time of all waiting processes. Consider a process p_x that is waiting on an event that has been notified at time t_y by process p_y. With event prediction, we determine that p_x can potentially wake up at time t_z where $t_z < t_y$ due to a possible event notification from another process p_z. Then the event at t_y that wakes up p_x should not be triggered yet since p_x may wake up at an earlier time than t_y. Without event prediction, the OoO scheduler avoids this situation by only delivering the earliest event notifications at each OOOSCHEDULE call [4].

Thus with event prediction, the OoO scheduler can accurately perform event delivery early while ensuring that no event notifications are lost or incorrectly triggered.

Algorithm 1 OoOScheduling

```
1: function WAIT(p)                1: function OOOSCHEDULE(p)
2:     LOCK(L)                     2:     EVENTPREDICTION()
3:     WAITING ← p                 3:     DELIVEREVENTS()
4:     OOOSCHEDULE(p)              4:     DISPATCHTHREADS()
5:     UNLOCK(L)                   5:     HANDLEENDOFSIMULATION()
6: end function                    6: end function
```

A. Problem Definition

The objective of event prediction is to conservatively predict the *earliest possible wake up time* τ_p of all processes p. This task can be mapped to a directed graph algorithm which calculates the minimum distances between all pairs. The well-known Floyd-Warshall algorithm [5] can complete this task with O(N^3) time complexity. The original OoO PDES approach uses a modified Floyd-Warshall algorithm to execute it with O(N^2) time complexity. However, the approach does not maintain any event prediction information between OOOSCHEDULE calls and requires the all pairs minimum distances algorithm to be executed every time. As a result, redundant event prediction information is often recalculated. This places a heavy limitation on the scalability of speedup with OoO PDES.

In contrast, we propose a lazy event prediction technique that only calculates event prediction information when needed. We introduce the *defining tree* data structure that allows event prediction information to be maintained between OOOSCHEDULE calls and thus avoids redundant operations. In order to increase scalability, we present an OoO PDES scheduling algorithm that is effectively O($m \log_2 m$), where $m \leq N$ and m is often significantly less than N.

978-3-9819263-4-7/DATE20/©2020 EDAA

Fig. 1 shows a simple demonstration model. A manager sends data to w workers waiting on event e and then notifies e to wake up all w workers (in the same delta cycle) where each worker spends a total of t_w time executing its total workload. The sequential run time of the model's simulation is effectively $\alpha + t_w * w$, where α denotes the time spent in the scheduler.

OoO PDES is very effective in the simulation of the model in Fig. 1 as all w workers can safely execute in parallel. With unlimited cores, the theoretical run time of the model's simulation with the OoO PDES scheduler is $\beta + t_w$ where β is the time spent in the OoO scheduler. β has a time complexity of O(w^2) with the original OoO scheduler since $w \approx N$. If the parallel work done by the workers is the dominating factor of the simulation run time, then the speedup of OoO against sequential simulation is $\frac{t_w * w}{t_w} = w$.

However, w may be large such that β (which grows quadratic with w) becomes the dominating factor in the run time. In this case, the theoretical speedup is effectively $\frac{t_w * w}{w^2 * c} = \frac{t_w}{w * c}$ where c represents a constant unit of time spent in OOOSCHEDULE. When $w * c > t_w$, speedup is less than 1. In other words, simulation actually slows down.

In this paper, we propose a novel lazy event prediction approach based on defining trees and a scheduling bypass strategy that reduces the effective time complexity of scheduling to O($m \log_2 m$), where $m \leq w$ and m is often significantly less than w. This enables sustained and improved speedup with OoO PDES against standard sequential simulation and the simulation of Fig. 1 becomes more scalable with a large number of workers. In the worst case, our new lazy event prediction strategy is $w/\log_2(w)$ times faster than the OoO approach.

```
0  void worker_thread()
1  { while(1)
2    { wait(e);
3      inport->nb_read();
4      work();
5    }
6  }
7  void manager_thread()
8  { while(1)
9    { wait(SC_TIME);
10     outport0->nb_write(…);
11     outport1->nb_write(…);
12     …
13     outportw->nb_write(…);
14     e.notify(SC_ZERO_TIME);
15   }
16 }
```

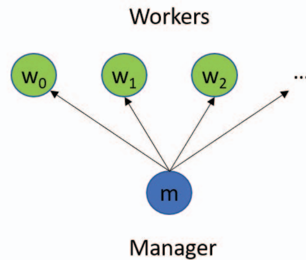

Fig. 1: Example model with 1 manager and w workers

B. Related Work

PDES was first proposed in [6] and then OoO PDES was introduced in [2] to further decrease simulation run time. [4] and [7] proposed event prediction for OoO thread dispatch and event delivery respectively, and is the state-of-the-art approach which we use as comparison to our new OoO PDES strategy with lazy event prediction.

Other techniques to speedup system simulation have been proposed. In standard sequential simulation, time decoupling has been implemented which allows processes to execute in an unsynchronized fashion for reduced context switches [8]. However, time decoupling results in lower accuracy of the simulation [8]. Our OoO PDES strategy does not suffer from any loss of simulation accuracy. Furthermore, [9], [10], and [11] proposed techniques to enable parallel multi-threading but each require developers to manually translate the sequential design into a parallel design. In contrast, our approach supports an automatic source transformation framework that does not require any manual manipulation from the developers.

II. Defining Trees

In original OoO PDES [2], [7], event prediction out of all scheduling tasks has the highest order time complexity of O(N^2) where N is the number of processes in the simulation. In this section, we outline how we can effectively reduce this time complexity to O($m \log_2 m$), where $m \leq N$.

Our newly proposed *defining tree* data structure keeps track of which processes determine the earliest possible wake up time τ of other processes. As an example, Fig. 2 illustrates the 4 different defining trees each represented by a unique color on top of the existing process graph of a model. Each vertex in the graph represents a process p with a corresponding τ_p and at any time p must be in a defining tree. All defining trees in the simulation make a defining forest. A directed edge[1] indicates that a process p_x may wake up process p_y in 1δ from its current τ_{p_x}. For instance, vertex A represents a process whose τ_A is 1δ and defines the τ_D and τ_C of vertices D and C where $\tau_D = \tau_C = 2\delta$. Since vertex A defines the τ_D and τ_C, vertices A, D, and C are part of the same defining tree. Furthermore, vertex G has $\tau_G = 3\delta$ and is defined by vertex D and thus vertex G is also part of the same defining tree. Vertex E is in the defining tree with the root B since B provides the minimum $\tau_E = 2\delta$. Furthermore, vertex K represents a process that waits on an event notified by itself, and thus K is the root of its own defining tree.

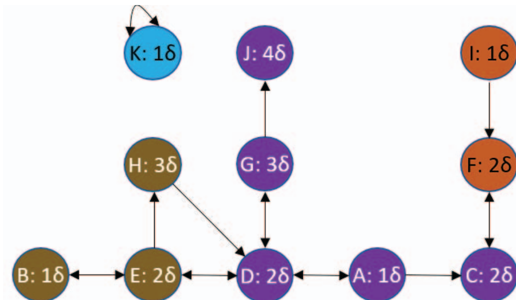

Fig. 2: Example of defining tree data structure.

A. Algorithm Definition

Algorithm 2 formally defines our proposed approach for OoO PDES lazy event prediction. In line 1, we introduce a set of weighted vertices $\mathbb{V}$ that represents the processes in the simulation and the weights represent the τ of the processes.

[1] Without loss of generality, we assume all event notifications carry a delay of a single delta cycle.

In line 2, we introduce a set of directed weighted edges $\mathbb{E}$ that represent that p_x can wake up vertex p_y by event notification. The weights indicate the notification delay.

Algorithm 2 Lazy Event Prediction with Defining Trees

```
1: 𝕍 = {p_0, p_1, p_2, ...}
2: 𝔼 = {(p_0, p_1), (p_1, p_0), (p_0, p_2), (p_2, p_0), ...}
3: 𝕌 ← ∅                                              ▷ Set of new waiting threads
4:
5: function WAIT(p)                                     ▷ 𝕌 is filled by WAIT
6:     LOCK(L)
7:     𝕌 ← 𝕌 ∪ {p}
8:     OOOSCHEDULE(p)
9:     UNLOCK(L)
10: end function
11:
12: function EVENTPREDICTION(𝕌)
13:     ℚ ← ∅                                          ▷ Set of updated threads
14:     𝔻 ← ∅
15:     RESETDEFININGTREES(𝕌, 𝔻)
16:     CHECKEVENTNOTIFICATIONS(𝔻)
17:     UPDATEDEFININGTREES(𝔻, ℚ)
18:     MULTISOURCEDIJKSTRA(ℚ)
19: end function
20:
21: function RESETDEFININGTREES(𝕌, 𝔻)
22:     for all p ∈ 𝕌 do        ▷ all new waiting threads in 𝕌 inserted to WAITING
23:         WAITING ← p
24:     end for
25:     while ∃p_x ∈ 𝕌 do                          ▷ reset all threads in defining tree
26:         for all p_y ∈ getNeighborsDefinedByMe(p_x) do
27:             if ∃notify ∈ getEventofWaitProcess(p_y) then
28:                 if getweight(notify) ≠ getweight(p_y) then
29:                     𝕌 ← 𝕌 ∪ {p_y}
30:                 end if
31:             end if
32:         end for
33:         weight(p_x) := ∞
34:         DefiningNeighbor(p_x) := ∅
35:         𝔻 ← 𝔻 ∪ {p_x}
36:         𝕌 ← 𝕌 \ {p_x}
37:     end while
38: end function
39:                                                ▷ check all active notifications
40: function CHECKEVENTNOTIFICATIONS(𝔻, ℚ)
41:     for all p ∈ 𝔻 do
42:         if ∃notify ∈ getEventofWaitProcess(p) then
43:             if getweight(notify) ≤ getweight(p) then
44:                 weight(p) := getweight(notify)
45:                 DefiningNeighbor(p) := ∅
46:                 ℚ ← ℚ ∪ {p}
47:             end if
48:         end if
49:     end for
50: end function
51:                          ▷ update all threads in defining tree and place into ℚ
52: function UPDATEDEFININGTREES(𝔻, ℚ)
53:     for all (p_s, p_t) ∈ getIncomingEdges(𝔻) do
54:         if getweight(p_s, p_t) + getweight(p_s) < getweight(p_t) then
55:             weight(p_t) := getweight(p_s, p_t) + getweight(p_s)
56:             ℚ ← ℚ ∪ {p_t}
57:         end if
58:     end for
59: end function
```

For example, Fig. 3 illustrates the execution of lazy event prediction with defining trees. Step 0 shows the initial state of the defining forest. Blue vertices indicate that the corresponding process is currently in the running or ready state while green vertices imply the waiting state. In step 1, vertex A goes into the waiting state and its τ_A is set to infinity. Furthermore, RESETDEFININGTREES is executed and all vertices who were in the same defining tree as vertex A and were on a directed path that stemmed from A have their τ set to infinity and are colored red. In step 2, UPDATEDEFININGTREES and CHECKEVENTNOTIFICATIONS are called. We check the event notifications and all incoming edges of each vertex that was reset in order to set their new τ. Each vertex that is set to a finite value is colored yellow. Notice after this step all other vertices in the defining forest still have a valid τ. Thus, we can simply omit these vertices in the defining forest except the yellow and red vertices to complete the lazy event prediction algorithm.

Step 3 is the final step where we call MULTISOURCEDIJKSTRA. Algorithm 3 formally defines this procedure as a multi-source variation of Dijkstra's algorithm which calculates the remaining τ of the yellow and red vertices. Other vertices are marked gray since the algorithm no longer considers those vertices. In step 3a, we create a min-priority queue of yellow vertices whose τ has already been computed. In step 3b, we pop the min-priority queue and check each of its outgoing edges to calculate the τ of its neighbors. The popped vertex is then marked green as it has already been visited by the multi-source Dijkstra's algorithm. In step 3c, we visit vertices C and A which are marked green and do not affect any others in the defining forest. In step 3d, first vertex G is visited which updates vertex J, and then vertex J is visited and the algorithm terminates.

Algorithm 3 Multi-Source Variation of Dijkstra's Algorithm

```
1: function MULTISOURCEDIJKSTRA(ℚ)
2:     heapify(ℚ)
3:     while ∃p ∈ ℚ do
4:         p_x := PopPriorityQueue(ℚ)
5:         for all p_y ∈ getNeighbors(p_x) do
6:             if getweight(p_x, p_y) + getweight(p_x) < getweight(p_y) then
7:                 weight(p_y) := getweight(p_x, p_y) + getweight(p_x)
8:                 PushPriorityQueue(p_y, ℚ)
9:             end if
10:        end for
11:    end while
12: end function
```

B. Maintaining Defining Trees

Lazy event prediction with defining trees only updates the τ_p of a process p if need be. When a process p changes from waiting to ready or ready to running, τ_p does not change. However, τ_p may change due to process p changing from running to waiting state. When a process p goes from running to waiting, τ_p is set to infinity. If the previously running and now waiting process p did not notify any events, then all the processes in the same defining tree as p that are on a path stemming from p each have their τ reset to infinity as well. However, in the case where p notifies a process n whose τ_n was defined by p at τ_p, then only τ_p is reset to infinity.

In addition, p may enter a new segment in the segment graph [2]. For brevity, we omit the discussion of segments in this paper (but our implementation supports them). Whenever a process changes its segment, there are implicit changes to the weights of the edges in the defining forest. Our current implementation handles segment changes dynamically with linear time complexity of the number of processes in the simulation. Future work can be done to implement a static analysis approach which would remove the necessity to handle segment changes dynamically.

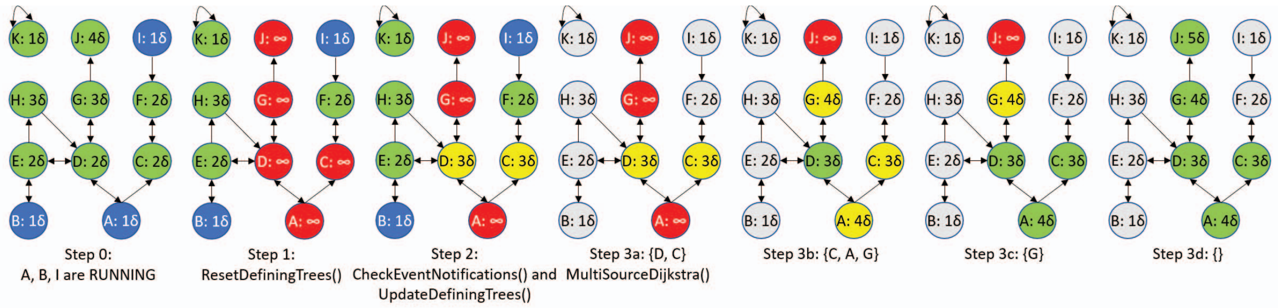

Fig. 3: Illustration of EVENTPREDICTION with *defining trees*

C. Complexity Analysis

In the following discussion, we analyze the complexity of lazy EVENTPREDICTION with defining trees. Let:

- N be the total number of processes
- $\mathbb{D}$ be a defining tree of m vertices
- $E_{\mathbb{D}}$ be the number of edges in defining tree $\mathbb{D}$
- $E_{\text{neighbors}(\mathbb{D})}$ be the number of incoming edges of immediate neighbors of defining tree $\mathbb{D}$

Only when process p in defining tree $\mathbb{D}$ calls WAIT can the m processes in $\mathbb{D}$ be updated. In the case where p calls WAIT, the time complexity of RESETDEFININGTREES is m since the algorithm executes in breadth first search fashion via the function getNeighborsDefinedByMe through $\mathbb{D}$ and functions getEventofWaitProcess and getweight have constant time complexity. Next, the complexity of CHECKEVENTNOTIFICATIONS is m because the number of elements in $\mathbb{D}$ is at most m and getDefiningNeighbor has constant time complexity. Furthermore, the complexity of UPDATEDEFININGTREES is at most $E_{\mathbb{D}} + E_{\text{neighbors}(\mathbb{D})}$. Finally, the time complexity of MULTISOURCEDIJKSTRA is $m + E_{\mathbb{D}} + m \log_2 m$ because we execute Dijkstra's algorithm with a binary heap and the only edges and vertices visited must be in $\mathbb{D}$. Thus the cumulative time complexity of EVENTPREDICTION is $E_{\mathbb{D}} + 2E_{\text{neighbors}(\mathbb{D})} + 3m + m \log_2 m$. Observe that $m \log_2 m$ is the highest order of complexity and thus the effective total time complexity is O($m \log_2 m$).

In the worst case, $m = N$, thus the worst case time complexity is $N \log_2 N$. As previously stated, the time complexity of the original OoO PDES event prediction strategy is O(N^2) and thus the worst case complexity is N^2. Thus the worst case speedup of EVENTPREDICTION is effectively $\frac{N^2}{N \log_2 N} = \frac{N}{\log_2 N}$ with the lazy approach against the original strategy .

Notice that the time complexity of EVENTPREDICTION with defining trees is only affected by edges and vertices of the defining tree and its neighbors. The lazy algorithm only visits the minimum subset of vertices that are necessary for correctly updating the defining forest. The τ_y of a process p_y is only updated if τ_x of process p_x that defines τ_y is reset to infinity and there is no event notification that wakes up process p_y at a time less than or equal to τ_y.

III. SCHEDULE BYPASS

We observe that the original OoO scheduler is often called only to produce redundant information that is immediately overwritten. In this section, we propose a lazy technique to largely eliminate redundant work by inserting a *bypass* into the algorithm.

In OoO PDES, there often is contention for OOOSCHEDULE where processes are blocked by the kernel lock into the scheduler. In the original approach, if there are N processes calling WAIT, then OOOSCHEDULE is called N times. However, many of these calls are repeating the same operations. Thus we propose the schedule bypass strategy to avoid redundant work. The essential idea is that whenever another process is attempting to grab the kernel lock L in a call to WAIT, the process owning the lock skips OOOSCHEDULE because there exists another process that will call OOOSCHEDULE immediately thereafter.

Algorithm 4 formally defines our proposed schedule bypass. Line 1 introduces a new set $\mathbb{K}$ of processes currently requesting kernel lock L. In line 2, we begin redefining WAIT. Instead of calling OOOSCHEDULE unconditionally, whenever there is another process in $\mathbb{K}$, the OOOSCHEDULE call is skipped and the process goes directly to sleep.

Algorithm 4 Schedule Bypass Strategy

1: $\mathbb{K} \leftarrow \varnothing$ ▷ Processes requesting kernel lock L protected by lock K
2: **function** WAIT(p)
3: LOCK(K)
4: $\mathbb{K} \leftarrow \mathbb{K} \cup \{p\}$
5: UNLOCK(K)
6: LOCK(L)
7: **WAITING** $\leftarrow p$
8: LOCK(K)
9: $\mathbb{K} \leftarrow \mathbb{K} \setminus \{p\}$
10: $K' \leftarrow \mathbb{K}$
11: UNLOCK(K)
12: **if** $K' = \varnothing$ **then**
13: OOOSCHEDULE(p)
14: **else**
15: SLEEP(p) ▷ implies atomic UNLOCK(L)/LOCK(L)
16: **end if**
17: UNLOCK(L)
18: **end function**

Fig. 4 demonstrates the effect of our bypass when there are N processes simultaneously calling WAIT. The left side illustrates the existing OoO PDES approach in which there are N calls to OOOSCHEDULE. The right side shows the lazy approach with schedule bypass with only two calls to

OOOSCHEDULE. As one can observe, only the first and last process that call WAIT closely after another actually enter the scheduler. Note that there is no deadlock possible since we have no circular locking.

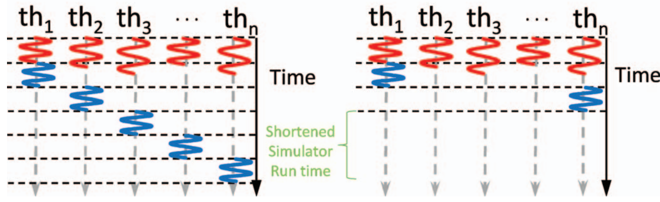

Fig. 4: Illustration of *schedule bypass*

IV. EXPERIMENTAL RESULTS

We have implemented the defining tree and schedule bypass optimizations on top of the RISC open-source implementation [12]. We show experimental results of five different models, specifically the sequential reference, the original RISC simulator, and our lazy OoO PDES simulator. Execution times are measured on an Intel Xeon CPU E5-2680 with 16 physical cores at 2.70GHz with CPU frequency scaling turned off.

A. Manager and Workers SystemC Model

We have implemented the model discussed in Section I with 500 worker processes. Table I shows the elapsed time, speedup against the sequential simulation, and the total time spent in event prediction. Notice that with the original OoO PAR approach, there is no speedup because event prediction becomes the dominating factor in the elapsed time. However, enabling defining trees drastically decreases the time spent for event prediction to less than 1%. Together, defining tree and schedule bypass achieve an overall speedup of more than 10x.

TABLE I: 1 Manager and 500 Workers Simulation Results

Scheduler	SEQ	OoO PAR			
		Orig.	Defining Trees	Schedule Bypass	Both
Elapsed Time	84.54s	179.79s	12.82s	128.13s	8.28s
Speedup	1x	0.47x	6.59x	0.66x	10.21x
Pred. Time	N/A	160.43s	1.30s	118.19s	0.27s

B. Mandelbrot Renderer

The Mandelbrot Renderer is an existing demo example in RISC. The model resembles the computations performed in a graphics rendering pipeline and generates 20 Mandelbrot frames with varying zoom factor. Fig. 5 shows the speedup of our lazy scheduling strategy compared to the original OoO PDES approach. We varied the number of processes rendering the images from 1 to 1024. Event prediction with the original OoO PDES increases quadratically with the number of processes while our new lazy algorithm is at worst O($N \log_2 N$). From Fig. 5, we can see that with increasing N, the lazy scheduler can sustain speedup while the original OoO scheduler crashes early to below 1x speedup. In the case of 1024 rendering processes, our proposed scheduler maintains 4x speedup over the sequential simulation and is 90x faster than the simulation running with the original OoO PDES scheduler. For lower N, the original and our lazy OoO PDES perform equally well. This is because for lower N, event prediction is not the dominating factor in the elapsed time for the original OoO PDES approach.

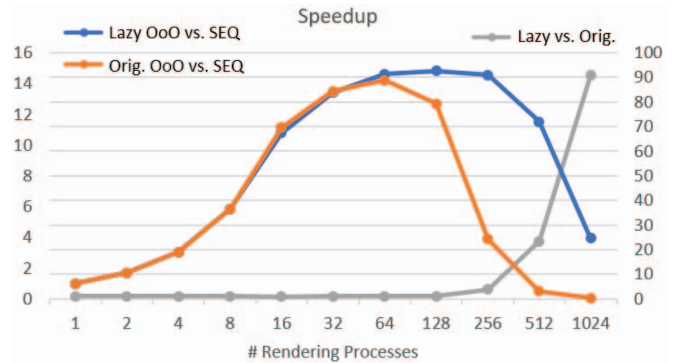

Fig. 5: Speedup of Lazy OoO PDES for Mandelbrot

C. Fibonacci

The Fibonacci model, also provided with RISC, computes the 45th number in the Fibonacci sequence. The model structure is a perfect binary tree where a child of a parent computes $fibo(n-1)$ and the other child of the same parent computes $fibo(n-2)$. We varied the number of leaf nodes N_l in the binary structure from 2 to 1024. The total number of processes in the simulation is $2^{\log_2 N_l + 1} - 1$, so $N_l = 1024$ means there are 2047 processes. Fig. 6 displays the speedup of our proposed lazy approach against both the original OoO PDES strategy and the sequential simulation. We can see that the lazy event prediction results in drastic speedup against the original approach while having sustained speedup against the sequential simulation. For $N_l = 1024$, our proposed scheduler maintains 4x speedup over the sequential simulation and is close to 50x faster than the simulation running with the original OoO PDES scheduler. For lower values of N_l, again both OoO PDES approaches perform the same since event prediction is not a dominating factor in the elapsed time for the original OoO PDES approach.

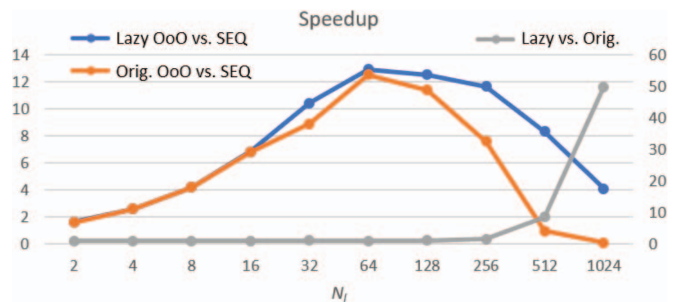

Fig. 6: Speedup of Lazy OoO PDES for Fibonacci

D. Network-on-Chip Particle Simulation

The Network-on-Chip (NoC) Particle Simulation model [13] resembles a realistic embedded system model. Fig. 7 shows

the model structure resembling a 2-D torus in which a tile represents a process that communicates with its neighbor tiles via blocking communication. The NOC particle model simulates the physical behavior of 24,000 particles in vacuum using NxN tiles. Fig. 8 shows the same speedup plot as before, with N varied from 1 to 8. We can see again the same sustained speedup behavior, confirming the benefits of our lazy approach for models with many processes.

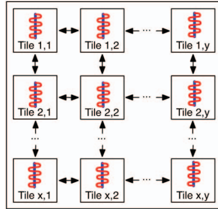

Fig. 7: Illustration of Network-on-Chip Particle Simulation

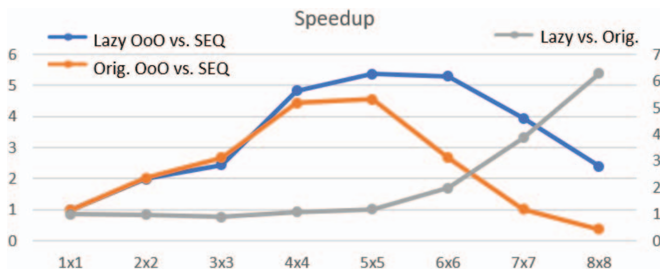

Fig. 8: Speedup of Lazy OoO PDES for NoC

E. *Image Classifier System using GoogleNet DNN*

The image classifier SystemC model [14] represents a real-world device using deep neural networks (DNN) for image classification. The application's purpose is to use frames from video to detect an attack in any of the images. During the simulation, the camera module sends images to the GoogleNet DNN [15] for classification. Each node in the DNN is modeled as a SystemC thread. In total, the model contains 145 threads. The detector module then receives the image classification from the DNN and sends a signal to the control terminal if it detects a threat.

We ran two versions of the classifier system by varying the number of parallel DNNs. In both cases, our proposed lazy event prediction shows high speedup against the sequential simulator and the original OoO PDES approach, as listed in Table II.

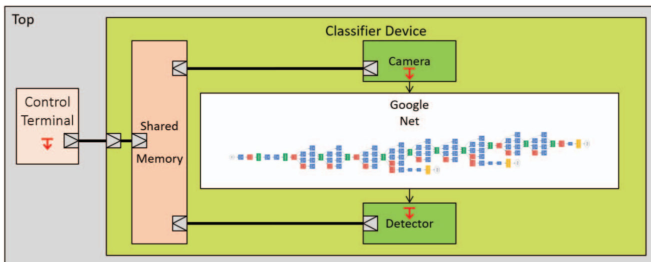

Fig. 9: Illustration of the Image Classifier Model

TABLE II: Classifier System Simulation Results

	Scheduler	SEQ	OoO PAR	Lazy OoO PAR
1 DNN	Elapsed Time	39.04s	12.79	7.37s
	Speedup	1x	3.05x	5.30x
2 DNN	Elapsed Time	39.49s	31.63s	6.64s
	Speedup	1x	1.25x	5.95x

V. Conclusion

In this paper, we proposed a lazy event prediction strategy for OoO PDES that reduces the time complexity of scheduling from O(N^2) to O($m \log_2 m$) where m is often significantly less than N. Through the use of defining trees and schedule bypass, our lazy event prediction approach only updates vertices in the defining forest if need be. This in turn increases the scalability of OoO PDES with the number of processes in the simulation. For simulations containing many processes, we showed improved and sustained speedups with lazy event prediction. A real world image classification application shows increased speedup of OoO PDES vs. sequential simulation from 1.25x to 5.95x. Experimental results show up to 90x speedup of OoO PDES with lazy event prediction against the original OoO PDES approach.

References

[1] IEEE Standard 1666-2011 for Standard SystemC® Language Reference Manual, IEEE Computer Society, January 2012.
[2] W. Chen, X. Han, C. Chang, G. Liu, 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] T. Schmidt, G. Liu, R. Dömer, "Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation", DAC 2017, Austin, TX, June 2017.
[4] W. Chen, R. Dömer, "Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions", DATE, Grenoble, France, March 2013.
[5] R. Floyd, "Algorithm 97: Shortest Path", Commun. ACM, June 1962.
[6] R. Fujimoto, "Parallel discrete event simulation", Commun. ACM, 33:3053, Oct. 1990.
[7] Z. Cheng, E. Arasteh, R. Dömer, "Event Delivery using Prediction for Faster Parallel SystemC Simulation", accepted for publication at the Asia and South Pacific Design Automation Conference, Beijing, China, January 2020.
[8] G. Glaser, G. Nitschey, and E. Hennig, "Temporal Decoupling with Error-Bounded Predictive Quantum Control," in FDL, 2015.
[9] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments," in DATE, 2016.
[10] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach", in DATE, 2013.
[11] N. Ventroux and T. Sassolas, "A New Parallel SystemC Kernel Leveraging Manycore Architectures," in DATE, 2016.
[12] G. Liu, T. Schmidt, Z. Cheng, D. Mendoza, R. Dömer, "RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset", CECS Technical Report 18-03, September 2018.
[13] K. Moazzemi, R. Dömer, and A. Chandramowlishwaran, "A SystemC Model for N-body Problems and its Parallel Design Space Exploration", CECS Technical Report 16-09, November 2016.
[14] E. Arasteh, R. Dömer, "An Untimed SystemC Model of GoogLeNet", Proceedings of the International Embedded Systems Symposium, Springer, Friedrichshafen, Germany, September 2019.
[15] C. Szegedy, et.al., "Going Deeper with Convolutions", CVPR, 2015.