# Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions

Weiwei Chen, Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine, USA
weiwei.chen@uci.edu, doemer@uci.edu

***Abstract*—Parallel Discrete Event Simulation (PDES) enables efficient validation of ESL models on multi-core simulation hosts. Out-of-order PDES is an advanced scheduling technique which allows multiple threads to run in parallel even in different simulation cycles. To maintain simulation semantics and timing accuracy, the compiler performs complex static conflict analysis so that the scheduler can make quick and safe decisions at run time and issue threads early. Often, however, out-of-order scheduling is prevented because of the unknown future behavior of the threads. In this paper, we extend the analysis in order to predict the future of candidate threads. Looking ahead of the current simulation state allows the scheduler to issue more threads in parallel, resulting in significantly reduced simulator run time. Our experimental results show simulation speedup up to 1.92x with only negligible increase in compile time.**

## I. Introduction

The validation of Electronic System-Level (ESL) designs is typically based on Discrete Event (DE) simulation. The explicit parallelism in ESL models is reflected in multiple concurrent threads controlled by the simulator. The simulator schedules the threads according to the execution semantics of the system-level description language (SLDL) used.

Traditional DE simulation, as implemented by the reference simulators for both SystemC and SpecC SLDLs, uses a *cooperative* multi-threading model. This allows only one thread to be active at any time, making it impossible to utilize the multiple computation resources that today are commonly available in multi-core hosts. [1], [2], [3] extend the simulation kernel for *synchronous* PDES. Multiple OS kernel threads with appropriate synchronization run in parallel in each simulation cycle so that the available cores in the host can be utilized. However, the number of parallel threads that can actually run in each cycle, is often very limited. The global simulation time restricts the usable parallelism in the model.

*Out-of-order PDES* (OoO PDES) [4] is an advanced technique that increases the multi-core CPU utilization by letting suitable threads run in parallel even when they are in different cycles. To preserve the simulation semantics and timing accuracy, OoO PDES relies on static conflict analysis by the compiler and dynamic checking in the scheduler to make aggressive but safe scheduling decisions.

In this paper, we propose an optimization of OoO PDES using prediction of potential conflicts. Our compiler generates conflict information for multiple scheduling steps in advance so that the scheduler can predict future potential conflicts. While existing OoO PDES prevents threads from being issued for any potential conflicts, our optimization uses conflict prediction to eliminate false positives, allowing them to execute as early as possible and run ahead as far as possible, so as to increase the simulation parallelism.

978-3-9815370-0-0/DATE13/©2013 EDAA

After a brief discussion of related work, we review OoO PDES in Section II. We then outline the idea of conflict prediction in Section III and describe the optimized scheduling algorithm and corresponding compiler support in Section IV. Finally, Section V provides experimental results for several embedded applications.

### A. Related Work

PDES is a well-studied subject [5], [6], [7]. Recently, it has gained attention again for ESL model validation as it allows to utilize the multiple cores in today's host PCs.

Synchronous PDES approaches are proposed in [1], [2], [3] which extend the simulator kernels to run threads in parallel in the same simulation cycle, i.e. same *delta* and *time*. However, synchronous PDES imposes a total order on simulation cycle advances, making them absolute barriers for thread execution. Available CPU cores remain idle while waiting for the threads mapped to other cores to reach the cycle barrier.

Out-of-order PDES [4] breaks the global time and cycle barrier and issues multiple threads in parallel even if they are in different simulation cycles. OoO PDES is a conservative approach that speeds up simulation on multi-core hosts without roll backs or sacrificing the simulation semantics and timing accuracy. It relies on static conflict analysis at compile time for quick scheduling decisions.

Parallel simulation on specialized hardware, such as Graphics Processing Units (GPU), has been studied in [8], [9]. However, model partitioning is difficult on heterogenous simulator units in [8], and the task dependency graph needs to be acyclic for partitioning in [9].

The idea in this paper is similar to the hardware strategy of *branch prediction* [10] which accelerates execution by looking ahead for future status. In contrast to branch prediction which updates possible conditional branches dynamically at run time, however, our technique generates prediction information for scheduling statically at compile time. Since OoO PDES is conservative, there is also no stalling or rolling back.

## II. Out-of-order Parallel DE simulation

OoO PDES localizes simulation time to each thread and instead of using global barriers, threads synchronize with each other only when necessary [4].

Conservative static analysis of potential conflicts is the key to fully preserve simulation semantics and timing accuracy. We distinguish three types of hazards:

- **Data hazards** are caused by parallel or out-of-order accesses to shared variables, namely read-after-write (RAW), write-after-read (WAR), or write-after-write (WAW).
- **Timing hazards** are caused by local time advances for individual threads. For example, consider two threads, a running thread $th_r$ and a candidate ready-to-be issued $th_c$. If $th_r$ may run with a time before $th_c$ after the next scheduling step, and it is not clear whether or not $th_r$'s future statements have any conflicts with $th_c$, then it is dangerous to issue $th_c$ out-of-order (even though there are no immediate data conflicts between $th_r$ and $th_c$).
- **Event hazards** are caused by out-of-order event notifications. For example, when a running thread $th_r$ wakes another thread $th_w$, it is dangerous to issue $th_c$ out-of-order since $th_w$ may impose hazards with respect to $th_c$.

In case of any of such hazards, the OoO PDES scheduler cannot issue threads out of the order.

For the OoO PDES conflict analysis, we define:

- **Segment** $seg_i$: statements executed by a thread between two scheduling steps.
- **Segment Boundary** $b_i$: SLDL primitives which call the scheduler, e.g. *wait*, *wait-for-time*, *par*.

Here, segment boundaries $b_i$ start segments $seg_i$. Thus, a directed graph is formed by segments. We define formally:

- **Segment Graph (SG)**: SG=(V, E), where V = $\{v \mid v_i$ is segment $seg_i$ started by segment boundary $b_i\}$, E=$\{e_{ij} \mid e_{ij}$ exists if $seg_j$ is reached after $seg_i\}$.

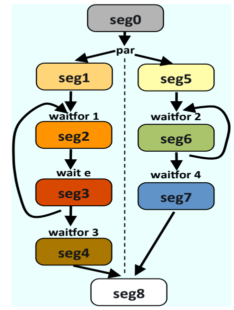
A corresponding segment graph can be derived from the control flow graph of a design. For example, Fig. 1(a) and (b) show a simple model written in SpecC SLDL and its segment graph. Starting from the initial segment $seg_0$, two separate segments $seg_1$ and $seg_5$ represent the two parallel threads after the *par* statement in line 19. New segments are created after each segment boundary, such as *waitfor 1* (line 8), *wait e* (line 9), and so on, and segments are connected following the control flow of the model. For instance, $seg_3$ is followed by $seg_2$ due to the *do-while* loop in lines 7-11.

At run time, the scheduler needs to check whether a *ready-to-run* thread at a particular segment can be issued out-of-order, i.e. without conflict. OoO PDES compiles the following data structures to detect potential conflicts among the $N$ segments in the model:

- **Variable Access List:** $segAL_i$ is the list of the variables that are accessed in $seg_i$. Each entry for a variable in this list is a tuple of (*Var, AccessType*).

```
1: #include <stdio.h>
2: int x = 0, y;
3: behavior B1(event e) s1        behavior B2(event e) s5
4: {                    s1        {                     s5
5:   void main(){       s1          void main(){        s5
6:     int i = 0;       s1            int i = 0;        s5
7:     do {          s1 s3            do {           s6 s5
8:       waitfor 1;  s2                 waitfor 2;   s6
9:       wait e;     s3                 y = i;       s6
10:      x = i;      s3                 notify e;    s6
11:    }while( i ++ < 2); s3          } while( i ++ < 2); s6
12:    waitfor 3;    s4               waitfor 4;     s7
13:    x = 27; }     s4               x = 42;  }     s7
14: };               s4           };                 s7
15: behavior Main()
16: {                   s0
17:   event e; B1 b1(e); B2 b2(e); s0
18:   int main(){       s0
19:     par{
20:       b1.main();  s1
21:       b2.main();  s5
22:   }               s8
23:   printf("x = %d\n", x);} s8
24: };                s8
```

(a) A simple design example




(b) Segment Graph

| seg | VAlist |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | x(W) |
| 4 | x(W) |
| 5 | |
| 6 | y(W) |
| 7 | x(W) |
| 8 | x(R) |

Data Conflict Table

| seg | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | F | F | F | F | F | F | F | F | F |
| 1 | F | F | F | F | F | F | F | F | F |
| 2 | F | F | F | F | F | F | F | F | F |
| 3 | F | F | F | T | T | F | F | T | T |
| 4 | F | F | F | T | T | F | F | T | T |
| 5 | F | F | F | F | F | F | F | F | F |
| 6 | F | F | F | F | F | F | T | F | F |
| 7 | F | F | F | T | T | F | F | T | T |
| 8 | F | F | F | T | T | F | F | T | F |

| seg | NT |
|---|---|
| 0 | (0:0) |
| 1 | (1:0) |
| 2 | (0:1) |
| 3 | (1:0) |
| 4 | (0:0) |
| 5 | (2:0) |
| 6 | (2:0) |
| 7 | (0:0) |
| 8 | (∞,0) |

Event Notification Table

| seg | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | F | F | F | F | F | F | F | F | F |
| 1 | F | F | F | F | F | F | F | F | F |
| 2 | F | F | F | F | F | F | F | F | F |
| 3 | F | F | F | F | F | F | F | F | F |
| 4 | F | F | F | F | F | F | F | F | F |
| 5 | F | F | F | F | F | F | F | F | F |
| 6 | F | F | F | T | F | F | F | F | F |
| 7 | F | F | F | F | F | F | F | F | F |
| 8 | F | F | F | F | F | F | F | F | F |

(c) Variable access list, data conflict, next time advance, and event notification table

Fig. 1. Simple design example.

- **Data Conflict Table (CT[N,N])**:

$$CT[i,j] = \begin{cases} true & \text{if } seg_i \text{ has data conflict with } seg_j \\ false & \text{otherwise} \end{cases}$$

  Note that CT[N,N] is symmetric and can be built by comparing pairs of the segment access lists.
- **Next Time Advance Table (NT[N])**: NT[i] = min{ time increment for a thread in $seg_i$ when it enters the next segment }.
- **Event Notification Table (ET[N,N])**:

$$ET[i,j] = \begin{cases} true & \text{if } seg_i \text{ notifies an event } seg_j \text{ waits for} \\ false & \text{otherwise} \end{cases}$$

  Note that ET[N,N] is asymmetric.

Fig. 2 shows the OoO PDES scheduling algorithm.

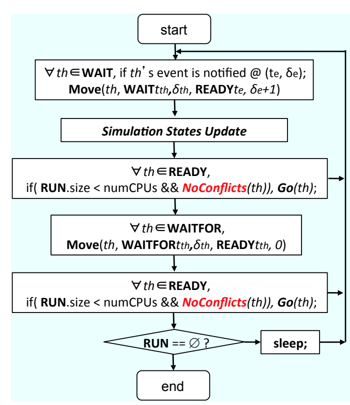



Fig. 2. Out-of-order Parallel DE simulation scheduling.

The conflict checking at run time, as listed in Algorithm 1, executes in constant time (O(1)) based on table lookups. The data conflict table is checked in lines 13-14 to avoid data hazards. The next time advance table serves in lines 15-16 to determine the time of a thread when entering into its next segment. Finally, the event notification table is used in lines 17-18 to identify any other threads that may wake up due to event delivery and run before the ready-to-run thread $th$.

**Algorithm 1** Conflict Detection in OoO PDES

```
1:  bool NoConflicts(Thread th)
2:  {
3:    for all th2 ∈ RUN ∪ READY,
4:    where (th2.t, th2.δ) < (th.t, th.δ) do
5:      if (Conflict(th, th2))
6:        then return false end if
7:    end for
8:    return true
9:  }
10:
11: bool Conflict(Thread th, Thread th2)
12: {
13:   if (th has data conflicts with th2) then
14:     return true end if /*check data hazards*/
15:   if (th2 may enter another segment before th) then
16:     return true end if /*check time hazards*/
17:   if (th2 may wake up another thread to run before th) then
18:     return true end if /*check event hazards*/
19:   return false
20: }
```

## III. State Prediction to Avoid False Conflicts

Out-of-order PDES issues threads in different simulation cycles to run in parallel if there are no potential hazards.

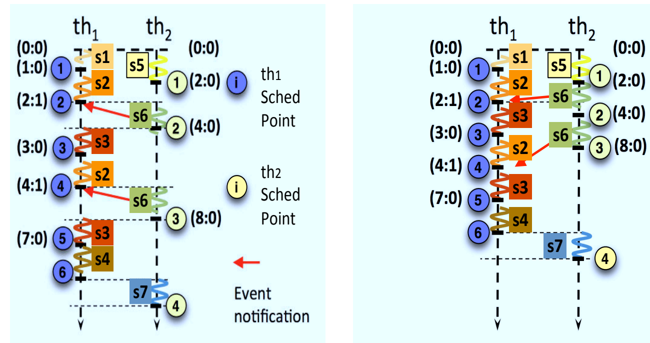
Fig. 3(a) shows the scheduling of thread execution for the example in Fig. 1. The threads $th_1$ and $th2$ are running in different segments with their own time. When one thread finishes its segment, shown as bold black bars as *scheduling point*, the scheduler is called for thread synchronization and issuing.

The OoO PDES scheduling algorithm is very conservative. Sometimes it makes *false conflict* detections at run time. For instance, in Fig. 3(a), when $th_2$ finishes its execution in $seg_5$ and hits the *scheduling point* $th_2$.①, $th_1$ is running in $seg_2$. The current time is (1:0) for $th_1$ and (0:0) for $th_2$. As listed in Fig. 1(c), the next time advance is (0:1) for $seg_2$ and (2:0) for $seg_5$. Therefore, the earliest time for $th_1$ to enter the next segment, i.e. $seg_3$, is (1:1), and for $th_2$ is (2:0). Since $th_1$ may run into its next segment ($seg_3$) with an earlier timestamp (1:1) than $th_2$ (2:0), the *Conflict()* in Algorithm 1 will return *true* at line 16. The scheduler therefore cannot issue $th_2$ out-of-order at scheduling point $th_2$.①.

However, this is a *false conflict* for out-of-order thread issuing. Although $th_1$ may run into next segment ($seg_3$) earlier than $th_2$, there are no data conflicts between $th_1$'s next segment $seg_3$ and $th_2$'s current segment $seg_6$. Moreover, the next time advance of $seg_3$ is (1:0). So $th_1$ will start a new segment no earlier than (2:0) after finishing $seg_3$. It is actually safe to issue $th_2$ out-of-order at scheduling point $th_2$.① since $th_2$'s time is not after (2:0).

If the scheduler knows what will happen with $th_1$ in more than one scheduling step ahead of scheduling point $th_2$.①, it can issue $th_2$ to run in parallel with $th_1$ instead of holding it back for the next scheduling step.

This motivates our idea of optimizing out-of-order PDES scheduling. With prediction information, as shown in Fig. 3(b), $th_2$ can be issued at both scheduling point $th_2$.① and $th_2$.⑥. The simulation time can thus be shortened.




(a) Original Scheduling (b) Optimized Scheduling

Fig. 3. Out-of-order PDES scheduling.

## IV. Optimized Out-of-order Parallel Scheduling with Predictions

Out-of-order PDES relies on static code analysis for safe scheduling decisions. The knowledge of future thread status helps the scheduler to issue more threads out of the order for faster simulation.

In this section, we will first discuss the static code analysis to generate the prediction information and then the optimized scheduling algorithm for out-of-order PDES using predictions.

### A. Static Prediction Analysis

At run time, threads switch back and forth between the states of RUNNING and WAITING. While RUNNING, the threads execute specific *segments* of their code. The out-of-order PDES scheduler checks the status of the threads by looking up the data structures for the *segments*.

The *Segment Graph* illustrates the execution order of the *segments* and their boundaries when the scheduler is called. The future segment information from any current segment can be derived from the *Segment Graph* at compile time.

We define the following data structures for static prediction analysis:

*1) Data hazards prediction:*

- **Segment Adjacency Matrix (A[N,N])**:

$$A[i,j] = \begin{cases} 1 & \text{if } seg_i \text{ is followed by } seg_j; \\ 0 & \text{otherwise.} \end{cases}$$

- **Data Conflict Table with $n$ prediction steps ($\mathbf{CT}_n$[N,N])** as follows:

$$CT_n[i,j] = \begin{cases} \textit{true} & \text{if } seg_i \text{ has a potential data conflict} \\ & \text{with } seg_j \text{ within n scheduling steps;} \\ \textit{false} & \text{otherwise.} \end{cases}$$

Here, $CT_0$[N,N] is the same as segment data conflict table CT[N,N]. However, $CT_n$ (n>0) is asymmetric.

Fig. 4(a) and (b) shows a partial segment graph and its *Adjacency Matrix*. The *Data Conflict Table* is shown in Fig. 4(c) where a data conflict exist between $seg_3$ and $seg_4$.

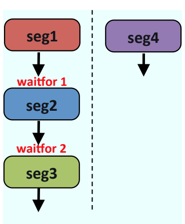
The *Data Conflict Tables with 0, 1 and 2 prediction steps* are shown in Fig. 5(a), (b) and (c), respectively. Since $seg_2$ is followed by $seg_3$ and $seg_3$ has a conflict with $seg_4$, a thread in $seg_2$ has a conflict with a thread who is in $seg_4$ after one scheduling step. Thus, $CT_1[2,4]$ is *true* in Fig. 5(b). Similarly, $seg_1$ is followed by $seg_2$ and $seg_2$ is followed by $seg_3$, so $CT_2[1,4]$ is *true* in Fig. 5(c).

The *Data Conflict Table with $n$ prediction steps* can be built recursively by using Boolean matrix multiplication. Basically, if $seg_i$ is followed by $seg_j$, and $seg_j$ has a data conflict with $seg_k$ within the next $n-1$ prediction steps, then $seg_i$ has a data conflict with $seg_k$ within the next $n$ prediction steps. Formally,

$$CT_0[N,N] = CT[N,N] \tag{1}$$

$$CT_n[N,N] = A'[N,N] * CT_{n-1}[N,N], \text{ where n} > 0. \tag{2}$$

Here, A'[N,N] is the *modified Adjacency Matrix* (e.g. Fig. 5(d)) with 1s on the diagonal so as to preserve the conflicts from the previous data conflict prediction tables. Note that more conflicts will be added to the conflict prediction tables when the number of prediction steps increases.




(a) Segment Graph

| seg | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |

(b) Segment Adjacency Matrix

| seg | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | F | F | F | F |
| 2 | F | F | F | F |
| 3 | F | F | F | T |
| 4 | F | F | T | F |

(c) Data Conflict Table

Fig. 4. A partial Segment Graph with Adjacency Matrix and Data Conflict Table.

| seg | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | F | F | F | F |
| 2 | F | F | F | F |
| 3 | F | F | F | T |
| 4 | F | F | T | F |

(a) Data Conflict Table w 0 prediction step ($CT_0$)

| seg | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | F | F | F | F |
| 2 | F | F | F | T |
| 3 | F | F | F | T |
| 4 | F | F | T | F |

(b) Data Conflict Table w 1 prediction step ($CT_1$)

| seg | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | F | F | F | T |
| 2 | F | F | F | T |
| 3 | F | F | F | T |
| 4 | F | F | T | F |

(c) Data Conflict Table w 2 prediction steps ($CT_2$)

| seg | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 |
| 2 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 |

(d) Modified Segment Adjacency Matrix

| seg | 0 | 1 | 2 |
|---|---|---|---|
| 1 | (1:0) | (3:0) | (∞,0) |
| 2 | (2:0) | (∞,0) | (∞,0) |
| 3 | (∞,0) | (∞,0) | (∞,0) |
| 4 | (∞,0) | (∞,0) | (∞,0) |

(e) Time Advance Table w 0, 1, 2 prediction steps

| seg | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 3 |
| 2 | 0 | 0 | 0 | 2 |
| 3 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 0 |

(f) Combined Data Conflict Table

Fig. 5. Data structures for optimized out-of-order PDES scheduling.

*Theorem 4.1:* $\exists M_{FP}$, $M_{FP} > 0$, so that $\forall n \geq M_{FP}$, no more conflicts will be added to $CT_n$.

*Proof:* Eq. (1) and (2) $\Rightarrow CT_n = A'^n * CT$.

In $A'$, $A'[i,j] = 1 \iff seg_j$ directly follows $seg_i$.

In $A'^2$, $A'^2[i,j] = 1 \iff \exists k$ that $A'[i,k] = A'[k,j] = 1$ or $A'[i,j] = 1$. In other words, $A'^2[i,j] = 1$ means that $seg_j$ can be reached from $seg_i$ via at most 1 other segment (1 segment apart). Hence, $A'^n[i,j] = 1$ means $seg_j$ can be reached from $seg_i$ via at most $n$ other segments ($n$ segments apart).

Since there are a limited number of segments in the *Segment Graph*, $\exists L$ that $\forall i, j$, $seg_i$ and $seg_j$ are either at most $L$ segments apart or they can never be reached from each other. $\Rightarrow$ There exists a *fixpoint* $M_{FP} = L$, that $\forall n > M_{FP}$, $A'^n = A'^{M_{FP}}$, and $CT_n = A'^n * CT = A'^{M_{FP}} * CT = CT_{M_{FP}}$. ■

Theorem 4.1 states that the number of prediction conflict tables for each design is limited. The maximum number of predictions is at most the length of the longest path in the Segment Graph.

*2) Time Hazards Prediction:*

- **Time Advance Table with $n$ prediction steps (NT$_n$[N])**: NT$_n$[i] = min{thread time advance after n + 1 scheduling steps from $seg_i$}. Here, $NT_0 = NT$.

Fig. 5(e) shows the segment *Time Advance Table with Predictions* ($NT_n$) for the example in Fig. 4. If a thread is now running in $seg_1$, it will be in $seg_2$ after one scheduling step and in $seg_3$ after two scheduling steps. The thread time will advance by at least (3:0) after two scheduling steps since $seg_2$ starts from *waitfor 1* and $seg_3$ starts from *waitfor 2*. Therefore, $NT_1[1] = (3:0)$.

*3) Event Hazards Prediction:* We need prediction information for event notifications to handle event hazards.

- **Event Notification Table with predictions (ETP[N, N])**:

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

Here, we have table entries of time advances.

Note that a thread can wake up another thread directly or indirectly via other threads. For instance, $th_1$ wakes up $th_2$, and $th_2$ then wakes up $th_3$ through event delivery. In this case, $th_1$ wakes up $th_2$ directly, and $th_3$ indirectly via $th_2$. We predict the minimum time advances between each thread segment pair in respect of both direct or indirect event notifications. The scheduler needs the predicted event notification information to know when a new thread may be ready to run for conflict checking at run time.

## B. Out-of-order PDES scheduling with Predictions

The out-of-order PDES scheduler issues threads out of the order at each scheduling step only when there are no potential hazards. With the help of static prediction analysis, we can optimize the scheduling conflict detection algorithm to allow more threads to run out-of-order.

Algorithm 2 shows the conflict checking function with M (0≤M≤$M_{FP}$) prediction steps. Note that when M=0, it is the original out-of-order PDES conflict detection.

**Algorithm 2** Conflict Detection with M Prediction Steps

1: **bool** Conflict(Thread $th$, Thread $th_2$)
2: {
3: /*iterate the prediction tables for data and time hazards*/
4: **for** ($m$ = 0; $m$<M; $m$++) **do**
5: **if** ($CT_m[th_2$.seg, $th$.seg] == *true*) **then**
6: **return** true; **end if** /*data hazards*/
7: **if** ($th_2$.timestamp + $NT_m[th_2$.seg]$\geq th$.timestamp) **then**
8: **break**; /*no data or time hazards between $th_2$ and $th$*/ **end if**
9: **end for**
10: **if** (m > M && $M < M_{FP}$ ) **then**
11: **return** true; **end if** /*time hazards*/
12: /*check event hazards*/
13: **for all** $th_w \in$ **WAIT do**
14: **if**(ETP[$th_2$.seg, $th_w$.seg] + $th_2$.timestamp<$th$.timestamp) **then**
15: /*$th_w$ may wake up before $th$*/
16: check data and time hazards between $th_w$ and $th$; **endif**
17: **end for**
18: **return** false;
19: }

Now, assume that $th_1$ and $th_2$ are two threads in the simulation of a model whose Segment Graph is Fig. 4(a). $th_1$is ready to run in $seg_4$ with timestamp (3:0), and $th_2$ is still running in $seg_1$ with timestamp (1:0).

*Conflict($th_1$)* in Algorithm 1 will return *true* because $th_2$ is possible to enter $seg_2$ with timestamp of (2:0) that is before $th_1$. Since the scheduler does not have information about the future status of $th_2$, it cannot issue $th_1$ to run out-of-order at the current scheduling step.

*Conflict($th_1$)* in Algorithm 2 will return *false* when M=1 or 2. With prediction information, the scheduler will figure out that $th_1$ (in $seg_4$) will not have data conflicts with $th_2$ after its next scheduling step (then in $seg_2$). Moreover, after $th_2$ finishes $seg_2$, the time for the next segment is at least (4:0), which is after $th_1$'s current one, i.e. (3:0). It is safe to issue $th_1$ out-of-order at the current scheduling step. As shown, the prediction information helps the run-time conflict checking to eliminate a *false conflict*.

### C. Optimized out-of-order PDES scheduling conflict checking with a Combined Prediction Table

We observe that $CT_m$ contains all the conflicts from $CT_0$ to $CT_{m-1}$ (m>0). In Algorithm 2, the checking loop in line 4-9 stops when the first conflict is found from the $CT_n$s.

We propose an optimized conflict checking algorithm (Algorithm 3) by using the following data structure:

- **Combined Conflict Prediction Table (CCT[N,N])**:

$$CCT[i,j] = \begin{cases} \textit{k+1} & \min\{k \mid CT_k[i,j] = \textit{true}\}; \\ \textit{0} & \text{otherwise.} \end{cases}$$

As shown in Fig. 5(f), the number of prediction steps is stored in $CCT$ instead of Boolean values.

There is no loop iteration for checking the conflict prediction table in Algorithm 3 since only one NxN combined table is used instead of $M$ NxN data conflict prediction tables.

Note that, Theorem 4.1 proves that only a *fixed* number of data conflict tables with predictions are needed for a specific design. The compiler can generate the complete series of

**Algorithm 3** Optimized Conflict Detection with Combined Prediction Tables for M steps

1: **bool** Conflict(Thread $th$, Thread $th_2$)
2: {
3: /*check the combined prediction table for data and time hazards*/
4: m = $CT[th_2$.seg, $th$.seg] - 1;
5: **if**($m \geq 0$) **then** /*There are data conflicts within M scheduling steps*/
6: /*$th_2$ may enter into a segment before $th$ and cause data hazards*/
7: **if**($th_2$.timestamp + $NT_m[th_2$.seg]<$th$.timestamp) **then**
8: **return** true; **end if**
9: **else if** ($M < M_{FP}$)
10: /*hazards may happen after M scheduling steps*/
11: **if**($th_2$.timestamp + $NT_M[th_2$.seg]<$th$.timestamp) **then**
12: **return** true; **end if**
13: **endif**
14: /*check event hazards*/
15: **for all** $th_w \in$ **WAIT do**
16: **if**(ETP[$th_2$.seg, $th_w$.seg] + $th_2$.timestamp< $th$.timestamp) **then**
17: /*$th_w$ may wake up before $th$*/
18: check data and time hazards between $th_w$ and $th$; **endif**
19: **end for**
20: **return** false;
21: }

conflict prediction tables and combine them into one table, i.e. CCT[N,N]. With this complete combined prediction table $CCT$, line 9-12 can be removed from Algorithm 3.

## V. Experiments and Results

We have implemented the proposed static prediction analysis and the optimized out-of-order PDES scheduler in a SpecC-based system design environment, and conducted experiments on three multi-media applications.

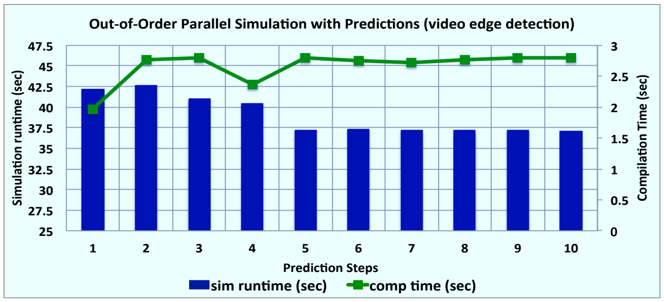
To demonstrate the benefits of out-of-order PDES scheduling using predictions, we show the compiler and simulator run times with different number of predictions in this section [1].

TABLE I
EXPERIMENTAL RESULTS FOR EMBEDDED APPLICATIONS

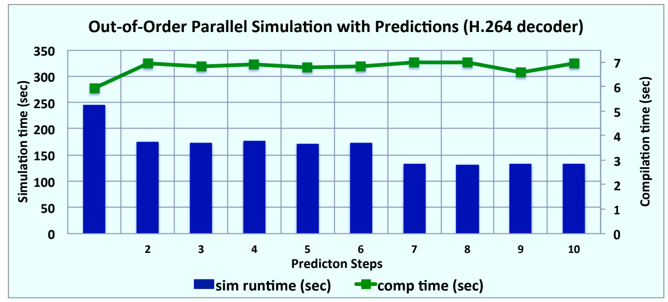
| Simulator: | | Out-of-order PDES without Predictions | | Out-of-order PDES with Predictions | | |
|---|---|---|---|---|---|---|
| | | compile time [sec] | sim time [sec] | compile time [sec] / speedup | sim time [sec] / speedup | max pred steps |
| Edge Detection | | 2.0 | 42.3 | 2.8 / 0.83 | 37.1 / **1.15** | 8 |
| H.264 Decoder | spec | 6.0 | 243.0 | 7.0 / 0.85 | 132.0 / **1.87** | 8 |
| | arch | 6.5 | 243.0 | 7.0 / 0.94 | 132.8 / **1.89** | 7 |
| | sched | 6.8 | 244.3 | 7.2 / 0.96 | 133.2 / **1.87** | 8 |
| | net | 6.7 | 244.6 | 7.2 / 0.91 | 132.9 / **1.92** | 9 |
| H.264 Encoder | | 38.0 | 2719.4 | 43.8 / 0.71 | 1448.8 / **1.88** | 62 |

Our first embedded application example, a **Video Edge Detector**, calculates edges in the images of a video stream. The application parallelizes the most computationally complex function *Gaussian Smooth* in the design. Fig. 6(a) shows the result with a test video stream of 100 frames with 1280x720 pixels. The simulation speed increases with more prediction steps. With the maximum prediction information, Table I shows a speedup of 1.15 with very small increase of compilation time.

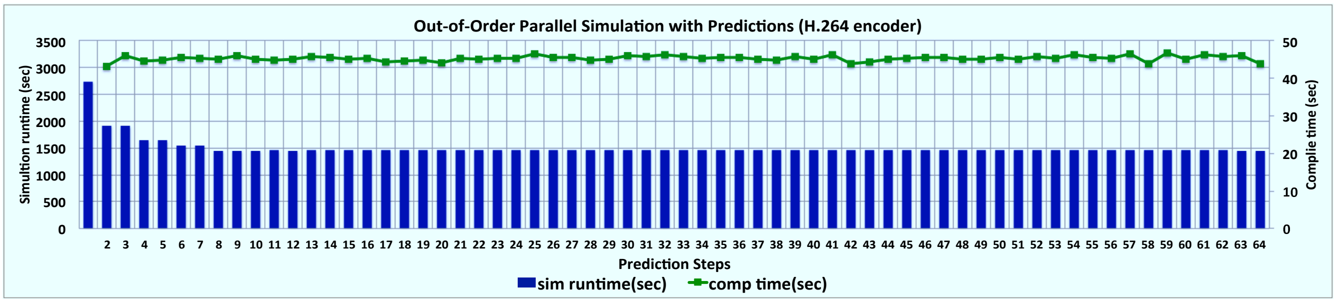
[1] All experiments have been performed on a symmetric multi-processing (SMP) capable server running 64-bit Fedora 12 Linux. The SMP hardware specifically consists of 2 Intel$^{(R)}$ Xeon$^{(R)}$ X5650 processors running at 2.67 GHz Each CPU contains 6 parallel cores, each of which supports 2 hyper-threads per core.




(a) the video edge detection model




(b) the H.264 decoder model




(c) the H.264 encoder model

Fig. 6. Simulation runtime and compilation time for OoO PDES with predictions

Our second application is a parallelized **H.264/AVC Video Decoder**. The model uses four parallel slice decoders to decode the independent slices in a video frame simultaneously. This design model is of industrial-size and consists of about $40k$ lines of code. We use a test stream of 1079 video frames with 1280x720 pixels per frame and simulate the model at four different abstraction levels, i.e. specification, architecture mapped, scheduling refined, and network linkage allocated. Table I shows an average speedup of 1.89 for simulation with maximum prediction information compared to the baseline out-of-order PDES simulation without predictions. Note that even for such a large design, the increased compile time due to the static prediction analysis is negligible. Fig. 6(b) shows that more simulation speedup can be gained with more prediction steps.

The third application is a parallelized **H.264/AVC Video Encoder** with parallel motion search. In our model, multiple motion search units are processing in parallel so that the comparison between the current image and multiple reference frames can be performed simultaneously. The test stream is a video of 95 frames with 176x144 pixels per frame. Table I shows a speedup of 1.88 for simulation with complete prediction information. As a large industrial design, the prediction conflict tables get to the *fixpoint* after 62 prediction steps. Fig. 6(c) shows the same trend of simulation speedup vs. prediction steps.

## VI. Conclusions and Future Work

Out-of-order PDES is an advanced technique for fast multi-core validation of ESL models. In this paper, we propose an optimized scheduling algorithm using static prediction analysis. The prediction information is derived from the Segment Graph at compile time and it helps the out-of-order PDES scheduler to avoid *false conflicts* at run time, allowing more threads to run out of order. Our experimental results show significant gains in simulation speed with negligible compilation costs.

In future work, we will optimize the thread scheduling order and look into additional approaches to further improve the simulation speed of ESL models.

## References

[1] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," in *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pp. 80–87, 2009.

[2] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures," in *Proceedings of the International Conference on Hardware/-Software Codesign and System Synthesis*, pp. 241–246, 2010.

[3] W. Chen, X. Han, and R. Dömer, "Multi-Core Simulation of Transaction Level Models using the System-on-Chip Environment," *IEEE Design and Test of Computers*, vol. 28, pp. 20–31, May/June 2011.

[4] W. Chen, X. Han, and R. Dömer, "Out-of-Order Parallel Simulation for ESL Design," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2012.

[5] K. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 440–452, Sept 1979.

[6] R. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, pp. 30–53, Oct 1990.

[7] D. Nicol and P. Heidelberger, "Parallel Execution for Serial Simulators," *ACM Transactions on Modeling and Computer Simulation*, vol. 6, pp. 210–242, July 1996.

[8] R. Sinha, A. Prakash, and H. D. Patel, "Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[9] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi, "SAGA: SystemC Acceleration on GPU Architectures," in *Proceedings of the Design Automation Conference (DAC)*, 2012.

[10] J. A. Fisher and S. M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.