Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models

Weiwei Chen, Xu Han, Che-Wei Chang, Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine, USA
weiwei.chen@uci.edu, hanx@uci.edu, cheweic@uci.edu, doemer@uci.edu

***Abstract*—The validation of system models at the transaction-level typically relies on discrete event simulation. In order to reduce simulation time, parallel discrete event simulation (PDES) can be used to increase the simulation speed by utilizing multiple cores available on today's host PCs. However, the total order of time imposed by regular discrete event simulators becomes a bottleneck that severely limits the benefits of parallel simulation.**

In this article, we present a new out-of-order PDES technique for simulating hardware/software design models on multi-core hosts. By localizing the simulation time to individual threads and carefully handling events at different times, a system model can be simulated following a partial order of time without loss of accuracy. Subject to advanced static analysis at compile time and table-based decisions at run time, threads can be issued early, reducing the idle time of available cores. Our proposed out-of-order PDES technique shows high performance gains in simulation speed with only a small increase in compile time.

I. Introduction

The increasing complexity of embedded systems poses tremendous challenges for modeling, validation, and synthesis at the electronic system level. Moving to higher abstraction levels, System-level Description Languages (SLDLs), such as SystemC [1] and SpecC [2], allow designers to describe hardware and software components together in the same transaction-level model (TLM). To enable efficient design space exploration and automatic refinement by computer-aided design tools, a TLM specifies the functionality of the intended design using a hierarchy of connected modules with clean separation of computation and communication. Notably, a TLM also specifies any potential for parallelism and pipelining explicitly.

In this work, we address the validation of TLMs which requires accurate yet fast simulation. TLM simulation is based on discrete event (DE) semantics. Within a single process, multiple concurrent threads represent the parallelism in the design model and are executed under the coordination of a DE-based scheduler. The scheduler issues threads following a global notion of time and interprets the "zero-delay" semantics of SLDLs by use of so-called *delta-cycles* which impose a partial order on the events that happen at the same time [1].

The reference simulators for both SpecC and SystemC SLDLs issue only a single thread at any time, thereby avoiding the otherwise necessary complex synchronization of the concurrent threads. Here, however, the sequential scheduler is a serious obstacle to improving simulation performance on multi-core host PCs.

Due to the inexpensive availability of parallel processing capabilities in today's multi-core hosts, recently Parallel Discrete Event Simulation (PDES) [3] has gained attention again. Using operating system kernel threads with added synchronization, PDES issues multiple threads concurrently and runs them on the available CPU cores in parallel. In turn, simulation speed increases significantly.

In this article, we describe an advanced PDES approach, called Out-of-Order Parallel Discrete Event Simulation (OoO PDES), which improves performance even further. Without loss of accuracy, OoO PDES relaxes the global in-order event and simulation time updates in PDES so that more threads can run in parallel, resulting in higher simulator speed.

Using advanced compile-time analysis of the threads and their potential conflicts, the OoO PDES scheduler can at run-time quickly decide whether or not it is safe to issue a set of threads in parallel. Moreover, by using thread-local simulation times, the simulator can also run threads in parallel which are at different simulation cycles, while ensuring fully SLDL-compliant behavior without modification of the design model.

A. Motivation

Both SystemC and SpecC SLDLs define DE-based execution semantics with "zero-delay" delta-cycles. Concurrent threads implement the parallelism in the design model, communicate via events and shared variables, and advance simulation time by use of *wait-for-time* primitives. Parallel execution of these threads is desirable to improve the simulation performance on multi-core hosts.

While the reference simulators issue only one thread at any time, recent PDES approaches, such as [4] and [5], take advantage of the fact that threads running at the same time and delta-cycle can execute in parallel. However, such *synchronous* PDES imposes a strict order on event delivery and time advance which makes delta- and time-cycles absolute barriers for thread scheduling. Specifically, when a thread finishes its execution cycle, it has to wait until all other active threads complete their execution for the same cycle. Only then the simulator advances to the next delta or time cycle. Additionally available CPU cores are idle until all threads have reached the cycle barrier.

We note that synchronous PDES issues threads strictly *in order*, with increasing time stamps after each cycle barrier. This can severely limit the desired parallel execution.

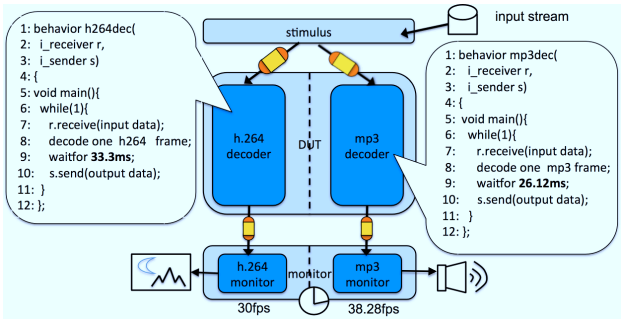
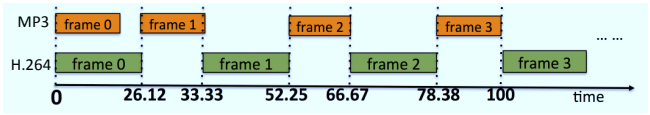


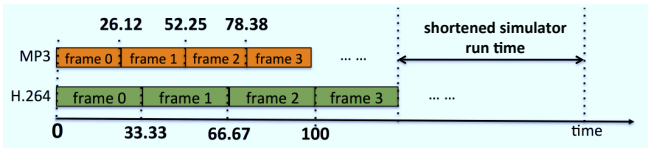
Fig. 1. High-level DVD player model with video and audio streams.

As a motivating example, Fig. 1 shows a high-level model of a DVD player which decodes a stream of H.264 video and MP3 audio data using separate decoders. Since video and audio frames are data independent, the decoders run in parallel. Both output the decoded frames according to their rate, 30 FPS for video (delay 33.3ms) and 38.28 FPS for audio (delay 26.12ms).

Unfortunately, synchronous PDES cannot exploit the parallelism in this example. Fig. 2(a) shows the thread scheduling along the time line. Except for the first scheduling step, only one thread can run at a time. Note that it is not data dependency but the global timing that prevents parallel execution.



(a) Synchronous PDES schedule



(b) Out-of-order PDES schedule

Fig. 2. Scheduling of the high-level DVD player model

In this article, we break the cycle barrier and let independent threads run *out-of-order* and in parallel. By carefully analyzing potential data and event dependencies and coordinating local time stamps for each thread, we fully maintain accuracy in simulation semantics and time. Fig. 2(b) shows the out-of-order schedule for the DVD player example. The MP3 and H.264 decoders simulate in parallel on different cores and maintain their own time stamps. As a result, we significantly reduce the simulator run time.

B. Related Work

PDES is a well-studied subject in the literature [6], [3], [7]. Two major synchronization paradigms can be distinguished, namely conservative and optimistic [3]. *Conservative* PDES typically involves dependency analysis and ensures in-order execution for dependent threads. In contrast, the *optimistic* paradigm assumes that threads are safe to execute and rolls back when this proves incorrect. Often, the temporal barriers in the model prevent effective parallelism in conservative PDES, while rollbacks in optimistic PDES are expensive in implementation and execution.

The proposed OoO PDES is conservative and can be seen as an improvement over synchronous PDES on symmetric multi-processing architectures with global simulation time and delta-cycle notion, such as [4], [8], [5]. An extended SystemC simulator described in [4] and [8] schedules multiple OS kernel threads in PDES fashion on multi-core processors. The SpecC-based approach described in [5] is very similar. However, it features a detailed synchronization protection mechanism which automatically instruments any user-defined and hierarchical channels. As such, the latter approach does not need to work around the cooperative SystemC execution semantics of the former approaches, neither does it require a specially prepared channel library.

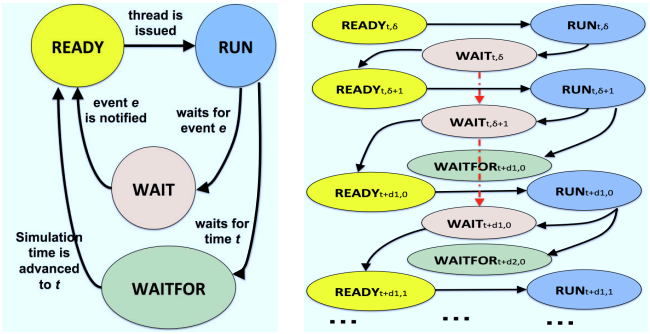
Distributed parallel simulation, such as [9] and [6], is a natural extension of PDES. Distributed simulation breaks the design model into modules, dispatches them on geographically distributed hosts, and then runs the simulation in parallel. However, model partitioning is difficult and the network speed becomes a simulation bottleneck due to the frequently needed communication.

Related work on improving simulation speed in the broader sense can be categorized into software modeling and specialized hardware approaches. Software techniques include the general idea of transaction-level modeling [10], which speeds up simulation by higher abstraction of communication, and source-level [11] or host-compiled simulation [12] which abstract the computation from the target platform. Generally, these approaches trade-off simulation speed against a loss in accuracy, for example, approximate timing due to estimated or back-annotated values.

Temporal decoupling proposed by SystemC TLM [13] also trades-off timing accuracy against simulation speed. Simulation time is incremented separately in different threads to minimize synchronization, but accuracy is reduced. Our approach also localizes simulation time to different threads, but fully maintains accurate timing.

Specialized hardware approaches include the use of Field-Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) platforms. For example, [14] emulates SystemC code on FPGA boards and [15] proposes a SystemC multi-threading model for GPU-based simulation. [16] presents a methodology to parallelize SystemC simulation across multi-core CPUs and GPUs. For such approaches, model partitioning is difficult for efficient parallel simulation on the heterogeneous simulator units.

Other simulation techniques change the infrastructure to allow multiple simulators to run in parallel and synchronize as needed. For example, the Wisconsin Wind Tunnel [17] uses a conservative time bucket synchronization scheme to synchronize simulators at a predefined interval. Another example [18] introduces a simulation backplane to handle the synchronization between wrapped simulators and analyzes the system



(a) Static states in synchronous PDES (b) Dynamic states in out-of-order PDES

Fig. 3. States and transitions of simulation threads (simplified).

to optimize the period of synchronization message transfers. Both techniques significantly speedup the simulation, however, at the cost of timing accuracy.

In contrast to the approaches above, our proposed OoO PDES does not sacrifice any accuracy in the simulation, neither does it require any special setup or hardware, nor any changes to the design model. In comparison to our work in [19] and [20], this article extends the dependency analysis at compile- and run-time. We add a detailed description of the conflict analysis for variables at different scopes (Section III-B1) and out-of-order hazards due to the localized timing (Section III-B). Moreover, we propose a novel prediction technique for OoO PDES scheduling (Section VI) that further improves the simulator speed. We show this with added experimental results for a new H.264 video encoder model (Section VII-D).

The rest of this paper is organized as follows: Section II presents the proposed OoO PDES technique and Section III introduces the needed conflict analysis at compile- and run-time. Next, Section IV describes how instance isolation can improve the analysis. The compilation algorithm is then detailed in Section V. Section VI presents a new optimization, namely scheduling with prediction. Finally, Section VII lists experimental results which show that, compared to synchronous PDES, our improved OoO PDES can further reduce the simulator run time with only a minor increase in compile time.

II. Out-of-Order Parallel Simulation

In contrast to synchronous PDES which imposes a total order on event processing and time advances, we now propose a new out-of-order simulation scheme where timing is only partially ordered. We localize the global simulation time (*time*, *delta*) for each thread and allow threads without potential data or event conflicts to run ahead of time while other working threads are still running with earlier timestamps.

A. Notations

To formally describe the OoO PDES scheduling algorithm, we introduce the following notations:

1) We define simulation time as tuple (t, δ) where $t =$ time, $\delta =$ delta-cycle. We order time stamps as follows:
   - **equal**: $(t_1, \delta_1) = (t_2, \delta_2)$, iff $t_1 = t_2$, $\delta_1 = \delta_2$
   - **before**: $(t_1, \delta_1) < (t_2, \delta_2)$, iff $t_1 < t_2$, or $t_1 = t_2$, $\delta_1 < \delta_2$
   - **after**: $(t_1, \delta_1) > (t_2, \delta_2)$, iff $t_1 > t_2$, or $t_1 = t_2$, $\delta_1 > \delta_2$
2) Each thread th has its own time (t_{th}, δ_{th}).
3) Since events can be notified multiple times and at different simulation times, we note an event e notified at (t, δ) as tuple (id_e, t_e, δ_e) and define: **EVENTS**$=\cup$**EVENTS**$_{t,\delta}$ where **EVENTS**$_{t,\delta} = \{(id_e, t_e, \delta_e) \mid t_e = t, \delta_e = \delta)\}$
4) For DE simulation, typically several sets of queued threads are defined, such as **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**}. These sets exist at all times and threads move from one to the other during simulation, as shown in Fig. 3(a). For OoO PDES, we define multiple sets with different time stamps, which we dynamically create and delete as needed, as illustrated in Fig. 3(b). Specifically, we define:
   - **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**, **JOINING**, **COMPLETE**}
   - **READY** = $\cup$**READY**$_{t,\delta}$, **READY**$_{t,\delta}$={th | th is ready to run at (t, δ)}
   - **RUN** = $\cup$**RUN**$_{t,\delta}$, **RUN**$_{t,\delta}$={th | th is running at (t, δ)}
   - **WAIT** = $\cup$**WAIT**$_{t,\delta}$, **WAIT**$_{t,\delta}$={th | th is waiting since (t, δ) for events (id_e, t_e, δ_e), where $(t_e, \delta_e) \geq (t, \delta)$}
   - **WAITFOR** = $\cup$**WAITFOR**$_{t,\delta}$, **WAITFOR**$_{t,\delta}$={th | th is waiting for simulation time advance to $(t, 0)$} Note that the delta cycle is always 0 for the **WAITFOR**$_{t,\delta}$ queues ($\delta = 0$)
   - **JOINING** = $\cup$**JOINING**$_{t,\delta}$, **JOINING**$_{t,\delta}$={th | th created child threads at (t, δ), and waits for them to complete}
   - **COMPLETE** = $\cup$**COMPLETE**$_{t,\delta}$, **COMPLETE**$_{t,\delta}$ = {th | th completed its execution at (t, δ)}

   Note that for efficiency our implementation orders these sets by increasing time stamps.
5) Initial state at the beginning of simulation:
   - $t = 0$, $\delta = 0$
   - **RUN** = **RUN**$_{0,0} = \{th_{root}\}$
   - **READY** = **WAIT** = **WAITFOR** = **COMPLETE** = **JOINING** = $\emptyset$
6) Simulation invariants:
   Let **THREADS** be the set of all existing threads. Then, at any time, the following conditions hold:
   - **THREADS** = **READY** $\cup$ **RUN** $\cup$ **WAIT** $\cup$ **WAITFOR** $\cup$ **JOINING** $\cup$ **COMPLETE**
   - $\forall$ A$_{t_1,\delta_1}$, B$_{t_2,\delta_2}$ $\in$ **QUEUES**: $A_{t_1,\delta_1} \neq B_{t_2,\delta_2} \Leftrightarrow A_{t_1,\delta_1} \cap B_{t_2,\delta_2} = \emptyset$

At any time, each thread belongs to exactly one set, and this set determines its state. Coordinated by the scheduler, threads change state by transitioning between the sets, as follows:

- **READY**$_{t,\delta}$ $\rightarrow$ **RUN**$_{t,\delta}$: thread becomes runable (is issued)
- **RUN**$_{t,\delta}$ $\rightarrow$ **WAIT**$_{t,\delta}$: thread calls **wait** for an event
- **RUN**$_{t,\delta}$ $\rightarrow$ **WAITFOR**$_{t',0}$, where $t < t' = t + delay$: thread calls **waitfor**($delay$)
- **WAIT**$_{t,\delta}$ $\rightarrow$ **READY**$_{t',\delta'}$, where $(t, \delta) < (t', \delta')$: the event that the thread is waiting for is notified; the thread becomes ready to run at (t', δ')
- **JOINING**$_{t,\delta}$ $\rightarrow$ **READY**$_{t',\delta'}$, where $(t, \delta) \leq (t', \delta')$: child threads completed and their parent becomes ready to run again
- **WAITFOR**$_{t,\delta}$ $\rightarrow$ **READY**$_{t,\delta}$, where $\delta = 0$: simulation time advances to $(t, 0)$, making one or more threads ready to run

The thread and event sets evolve during simulation as illustrated in Fig. 3(b). Whenever the sets **READY**$_{t,\delta}$ and **RUN**$_{t,\delta}$ are empty and there are no **WAIT** or **WAITFOR** queues

with earlier timestamps, the scheduler deletes **READY**$_{t,\delta}$ and **RUN**$_{t,\delta}$, as well as any expired events with the same timestamp **EVENTS**$_{t,\delta}$ (lines 8-14 in Algorithm 1).

Algorithm 1 Out-of-order PDES Algorithm

```
 1: /* trigger events */
 2: for all th ∈ WAIT do
 3:     if ∃ event (id_e, t_e, δ_e), th awaits e, and (t_e, δ_e) ≥ (t_th, δ_th) then
 4:         move th from WAIT_{t_th,δ_th} to READY_{t_e,δ_e+1}
 5:         t_th = t_e; δ_th = δ_e + 1
 6:     end if
 7: end for
 8: /* update simulation subsets */
 9: for all READY_{t,δ} and RUN_{t,δ} do
10:     if READY_{t,δ} = ∅ and RUN_{t,δ} = ∅ and WAITFOR_{t,δ} = ∅ then
11:         delete READY_{t,δ}, RUN_{t,δ}, WAITFOR_{t,δ}, EVENTS_{t,δ}
12:         merge WAIT_{t,δ} into WAIT_{next(t,δ)}; delete WAIT_{t,δ}
13:     end if
14: end for
15: /* issue qualified threads (delta cycle) */
16: for all th ∈ READY do
17:     if RUN.size < numCPUs and HasNoConflicts(th) then
18:         issue th
19:     end if
20: end for
21: /* handle wait-for-time threads */
22: for all th ∈ WAITFOR do
23:     move th from WAITFOR_{t_th,δ_th} to READY_{t_th,0}
24: end for
25: /* issue qualified threads (time advance cycle) */
26: for all th ∈ READY do
27:     if RUN.size < numCPUs and HasNoConflicts(th) then
28:         issue th
29:     end if
30: end for
31: /* if the scheduler hits this case, we have a deadlock */
32: if RUN = ∅ then
33:     report deadlock and exit
34: end if
```

B. Out-of-order PDES Scheduling Algorithm

Algorithm 1 defines the scheduling algorithm of our OoO PDES. At each scheduling step, the scheduler first evaluates notified events and wakes up corresponding threads in **WAIT**. If a thread becomes ready to run, its local time advances to (t_e, δ_e+1) where (t_e, δ_e) is the timestamp of the notified event (line 5 in Algorithm 1). After event handling, the scheduler cleans up any empty queues and expired events and issues qualified threads for the next delta-cycle (line 18). Next, any threads in **WAITFOR** are moved to the **READY** queue corresponding to their wait time and issued for execution if qualified (line 28). Finally, if no threads can run (**RUN** $= \emptyset$), the simulator reports a deadlock and quits[1].

Note that our scheduling is aggressive. The scheduler issues threads for execution as long as idle CPU cores and threads without conflicts (**HasNoConflicts**(th)) are available.

Note also that we can easily turn on/off the parallel out-of-order execution at any time by setting the **numCPUs** variable. For example, when in-order execution is needed during debugging, we set **numCPUs** $= 1$ and the algorithm will behave the same as the traditional DE simulator where only one thread is running at all times.

[1]The condition for a deadlock is the same as for a regular DE simulator.

III. Out-of-Order Conflict Analysis

The OoO scheduler depends on conservative analysis of potential conflicts among the active threads. We now describe the threads and their position in their execution, and then present the conflict analysis which we separate into static compile-time and dynamic run-time checking.

A. Thread Segments and Segment Graph

At run time, threads switch back and forth between the states of RUNNING and WAITING. When RUNNING, they execute specific *segments* of their code. To formally describe our OoO PDES conflict analysis, we introduce the following definitions:

- **Segment** seg_i: source code statements executed by a thread between two scheduling steps
- **Segment Boundary** v_i: SLDL statements which call the scheduler, i.e. *wait*, *waitfor*, *par*, etc.

Note that segments seg_i and segment boundaries v_i form a directed graph where seg_i is the segment following the boundary v_i. Every node v_i starts its segment seg_i and is followed by other nodes starting their corresponding segments. We formally define:

- **Segment Graph (SG)**: **SG=(V, E)**, where **V** = $\{v \mid v$ is a segment boundary$\}$, **E**=$\{e_{ij} \mid e_{ij}$ is the set of statements between v_i and v_j, where v_j could be reached from v_i, and $seg_i = \cup e_{ij}\}$.

Fig. 4(a) shows a simple example written in SpecC SLDL. The design contains two parallel instances `b1` and `b2` of type `B`. Both `b1` and `b2` compute the sum of a range of elements stored in a global array `array`. The work range is provided at the input ports `begin` and `end`, and the result is passed back to the parent via the output port `sum`. Finally, the `Main` behavior prints `sum1` of `b1` and `sum2` of `b2` to the screen.

From the corresponding control flow graph in Fig. 4(b), we derive the segment graph in Fig. 4(c). The graph shows five segment nodes connected by edges indicating the possible flow between the nodes. From the starting node *v0*, the control flow reaches the `par` statement (line 28) which is represented by the nodes *v1* (start) and *v2* (end) when the scheduler will be called. At *v1*, the simulator will create two threads for `b1` and `b2`. Since both are of the same type `B`, both will reach `waitfor 2` (line 12) via the same edge *v1→v3*. From there, the control flow reaches either `wait` (line 15) via *v3→v4*, or will skip the `while` loop (line 13) and complete the thread execution at the end of the `par` statement via *v3→v2*. From within the `while` loop, control either loops around *v4→v4*, or ends the loop and the thread via *v4→v2*. Finally, after *v2* the execution terminates.

Note that a code statement can be part of multiple segments. For example, line 19 belongs to both seg_3 and seg_4.

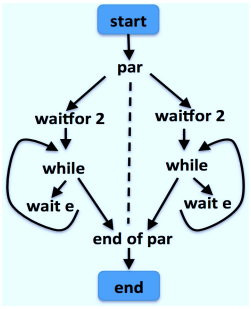
Note also that threads execute one segment in each scheduling step, and multiple threads may execute the same segments.

B. Static Conflict Analysis

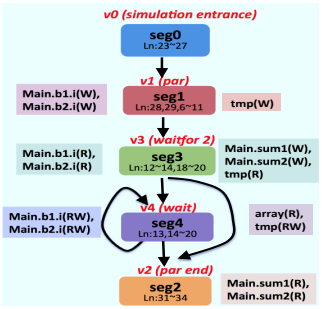
To comply with SLDL execution semantics, threads must not execute in parallel or out-of-order if the segments they are

```
1 #include <stdio.h>
2 int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
3 behavior B(in int begin,// port variable
4             in int end,  // port variable
5             out int sum) // port variable
6 {
7   int i; event e;        // member variables
8   void main(){
9     int tmp;             // stack variable
10    tmp = 0;             // tmp(W)
11    i = begin;           // i(W), begin(R)
12    waitfor 2;           // segment boundary (waitfor)
13    while(i <= end){     // i(R), end(R)
14      notify e;          // notify event e
15      wait e;            // segment boundary (wait)
16      tmp += array[i];   // array(R), tmp(RW)
17      i ++;              // i(RW)
18    }
19    sum = tmp; }         // sum(W)
20 };
21
22 behavior Main()
23 {
24   int sum1, sum2;      // member variables
25   B b1(0, 4, sum1);    // behavior instantiation
26   B b2(5, 9, sum2);    // behavior instantiation
27   int main(){
28     par{
29         b1.main();     // segment boundary (par)
30         b2.main();
31     }                  // segment boundary (par_end)
32     printf("sum 1 is :%d \n", sum1); // sum1(R)
33     printf("sum 2 is :%d \n", sum2); // sum2(R) }
34 };
```

(a) Example source code in SpecC



(b) Control Flow Graph (CFG)



(c) Segment Graph (SG)

seg	0	1	2	3	4
0	F	F	F	F	F
1	F	T	F	T	T
2	F	F	F	T	T
3	F	T	T	T	T
4	F	T	T	T	T

(d) Data conflict table (CTab)

seg	0	1	2	3	4
0	F	F	F	F	F
1	F	F	F	F	F
2	F	F	F	F	F
3	F	F	F	F	T
4	F	F	F	F	F

(e) Event notification table (NTab)

segment	0	1	2	3	4
Current Time Advance	N/A	(0:0)	(0:0)	(2:0)	(0:1)

(f) Time advance table for the current segment (CTime)

segment	0	1	2	3	4
Next Time Advance	(0:0)	(2:0)	(∞:0)	(0:0)	(0:0)

(g) Time advance table for the next segment (NTime)

Fig. 4. Simple design example with corresponding control flow graph, segment graph, conflict tables, and time advance tables.

in pose any hazard towards validity of data, event delivery, or timing.

1) Data Hazards: Data hazards are caused by parallel or out-of-order accesses to shared variables. Three cases exist, namely read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW).

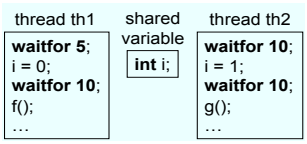


Fig. 5. Write-after-write (WAW) conflict between two parallel threads.

Fig. 5 shows a simple example of a WAW conflict where two parallel threads th_1 and th_2 write to the same variable i at different times. Simulation semantics require that th_1 executes first and sets i to 0 at time $(5, 0)$, followed by th_2 setting i to its final value 1 at time $(10, 0)$. If the simulator would issue the threads th_1 and th_2 in parallel, this would create a race condition, making the final value of i non-deterministic. Thus, we must not schedule th_1 and th_2 out-of-order. Note, however, that th_1 and th_2 can run in parallel after their second wait-for-time statement if the functions $f()$ and $g()$ are independent.

Since data hazards stem from the code in specific segments, we analyze data conflicts statically at compile-time and create a table where the scheduler can then at run-time quickly lookup any potential conflicts between active segments.

We define a **data conflict table** $CTab[N, N]$ where N is the total number of segments in the application code: $CTab[i, j] = \mathit{true}$, iff there is a potential data conflict between the segments s_i and s_j; otherwise, $CTab[i, j] = \mathit{false}$.

To build the conflict table, we compile for each segment a *variable access list* which contains all variables accessed in the segment. Each entry is a tuple *(Symbol, AccessType)* where *Symbol* is the variable and *AccessType* specifies read-only (R), write-only (W), read-write (RW), or pointer access (Ptr).

Finally, we create the conflict table $CTab[N, N]$ by comparing the access lists for each segment pair. If two segments s_i and s_j share any variable with access type (W) or (RW), or there is any shared pointer access by s_i or s_j, then we mark this as a potential conflict.

Fig. 4(d) shows the data conflict table for the example in Fig. 4(a). Here, for instance, seg_4 has a data conflict (RAW) with seg_1 since seg_4 has `i`(RW) (line 17) and seg_1 has `i`(W) (line 11) in the variable access list.

Not all variables are straightforward to analyze, however. The SLDL actually supports variables at different scopes as well as ports which are connected by port maps in the structural hierarchy of the design model.

We distinguish and handle the following cases:

- **Global variables**, e.g. `array` in line 2: This case is discussed above and can be handled directly as tuple *(Symbol, AccessType)*.
- **Local variables**, e.g. `tmp` in line 9: Local variables are stored on the function call stack and cannot be shared between different threads. Thus, they can be ignored in the access lists.
- **Instance member variables**, e.g. `i` in line 7: Since instances may be instantiated multiple times and then their variables are different, we need to distinguish them by their complete *instance path* prepended to the variable

name. For example, the actual *Symbol* used for the two instances of variable `i` is `Main.b1.i` or `Main.b2.i`, respectively.

- **Port variables**, e.g. `sum` in line 5: For ports, we need to find the actual variable mapped to the port and use that as *Symbol* together with its actual instance path. For example, `Main.sum1` is the actual variable mapped to port `Main.b1.sum`. Note that tracing ports to their connected variables requires the compiler to follow port mappings through the structural hierarchy of the design model which is possible when all used components are part of the current translation unit.
- **Pointers**: We currently do not perform pointer analysis (future work). For now, we conservatively mark all segments with pointer accesses as potential conflicts.

2) Event Hazards: Thread synchronization through event notification also poses hazards to out-of-order execution. Specifically, thread segments are dependent when one is waiting for an event notified by another.

We define an **event notification table** $NTab[N, N]$ where N is the total number of segments: $NTab[i, j] =$ *true*, iff segment s_i notifies an event that s_j is waiting for; otherwise, $NTab[i, j] =$ *false*. Note that in contrast to the data conflict table above, the event notification table is not symmetric.

Fig. 4(e) shows the event notification table for the simple example. For instance, $NTab[3, 4] =$ *true* since seg_3 notifies the event `e` (line 14) which seg_4 waits for (line 15).

Note that, in order to identify event instances, we use the same scope and port map handling for events as described above for data variables.

3) Timing Hazards: The local time for an individual thread in OoO PDES can pose a timing hazard when the thread runs too far ahead of others. To prevent this, we analyze the time advances of threads at segment boundaries. There are three cases with different increments, as listed in Table I.

TABLE I
TIME ADVANCES AT SEGMENT BOUNDARIES.

Segment boundary	Time Increment	
wait event	increment by one delta cycle	(0:1)
waitfor t	increment by time t	(t:0)
par/par_end	no time increment	(0:0)

We define two time advance tables, one for the segment a thread is currently in, and one for the next segment(s) that a thread can reach in the following scheduling step.

The **current time advance table** $CTime[N]$ lists the time increment that a thread will experience when it enters the given segment. For the example in Fig. 4(f) for instance, the *waitfor* 2 (line 12) at the beginning of seg_3 sets $CTime[3] = (2 : 0)$.

The **next time advance table** $NTime[N]$ lists the time increment that a thread will incur when it leaves the given and enters the next segment. Since there may be more than one next segment, we list in the table the minimum of the time advances which is the earliest time the thread can become active again. Formally:
$NTime[i] = min\{CTime[j], \forall seg_j$ which follow $seg_i\}$.

For example, Fig. 4(g) lists $NTime[3] =$ (0:0) since seg_3 is followed by seg_4 (increment (0:1)) and seg_2 (increment (0:0).

TABLE II
EXAMPLES FOR DIRECT AND INDIRECT TIMING HAZARDS.

Situation	th_1	th_2	Hazard?
Direct Timing Hazard	(10:2)	(10:0), next segment at (10:1)	yes
	(10:2)	(10:0), next segment at (12:0)	no
Indirect Timing Hazard	(10:2)	(10:0), wakes th_3 at (10:1)	yes
	(10:2)	(10:1), wakes th_3 at (10:2)	no

There is two types of timing hazards, namely direct and indirect ones. For a candidate thread th_1 to be issued, a *direct timing hazard* exists when another thread th_2, that is safe to run, resumes its execution at a time earlier than the local time of th_1. In this case, the future of th_2 is unknown[2] and could potentially affect th_1. Thus, it is not safe to issue th_1.

Table II shows an example where th_1 is considered for execution at time (10:2). If there is a thread th_2 with local time (10:0) whose next segment runs at time (10:1), i.e. before th_1, then the execution of th_1 is not safe. However, if we know from the time advance tables that th_2 will resume its execution later at (12:0), no timing hazard exists with respect to th_2.

An *indirect timing hazard* exists if a third thread th_3 can wake up earlier than th_1 due to an event notified by th_2. Again, Table II shows an example. If th_2 at time (10:0) potentially wakes a thread th_3 so that th_3 runs in the next delta cycle (10:1), i.e. earlier than th_1, then it is not safe to issue th_1.

C. Dynamic Conflict Detection

With the above analysis performed at compile time, the generated tables are passed to the simulator so that it can make quick and safe scheduling decisions at run time by using table lookups. Our compiler also instruments the design model such that the current segment ID is passed to the scheduler as an additional argument whenever a thread executes scheduling statements, such as *wait* and *wait-for-time*.

At run-time, the scheduler calls a function **HasNoConflicts**(th) to determine whether or not it can issue a thread th early. As shown in Algorithm 2, **HasNoConflicts**(th) checks for potential conflicts with all concurrent threads in the **RUN** and **READY** queues with an earlier time than th. For each concurrent thread, function **Conflict**(th, th_2) checks for any data, timing, and event hazards. Note that these checks can be performed in constant time (O(1)) due to the table lookups.

IV. THE EFFECT OF INSTANCE ISOLATION

The static analysis described in Section III is sometimes overly conservative and generates *false conflicts* which prevent the scheduler from issuing threads that actually do not pose any hazard. In Fig. 4 for example, the two threads for $b1$ and $b2$ are executing in parallel in segment seg_3 which contains write (W) accesses to variables `Main.sum1` and `Main.sum2`. Consequently, the above analysis reports a data conflict between the two threads th_{b1} and th_{b1} preventing them from executing in parallel. In reality, however, the execution

[2]We propose to predict the future of such threads in Section VI.

Algorithm 2 Conflict Detection in Scheduler

```
1:  bool HasNoConflicts(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 conflict with th2) then
14:     return true end if
15:   if (th2 may enter another segment before th) then
16:     return true end if
17:   if (th2 may wake up another thread to run before th) then
18:     return true end if
19:   return false
20: }
```

is safe when th_{b1} and th_{b2} are both in seg_3 because instance `b1` will only access `Main.sum1` and `b2` only `Main.sum2`. Thus, the conflict does not really exist.

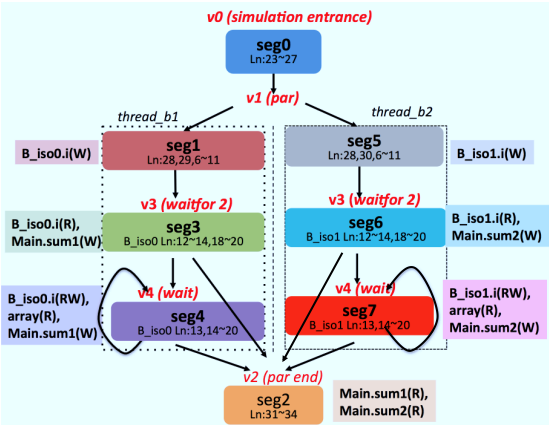
Looking closer at this false conflict, we see that both `b1` and `b2` share the same definition, i.e. the same code segments. Because of this sharing, we cannot determine that only thread th_{b1} accesses `Main.sum1` but not th_{b2} (and vice versa).

```
3  behavior B_iso0(...)      behavior B_iso1(...)
4  {                         {
     // same body as B          // same body as B
20 };                        };
21
22 behavior Main()
23 {
24   int sum1, sum2;
25   B_iso0 b1(0, 4, sum1);
26   B_iso1 b2(5, 9, sum2);
27   int main(){
       ...
34 };
```

(a) Source code with isolated instances

seg	0	1	2	3	4	5	6	7
0	F	F	F	F	F	F	F	F
1	F	T	F	T	T	F	F	F
2	F	F	F	T	T	F	T	T
3	F	T	T	T	T	F	F	F
4	F	T	T	T	T	F	F	F
5	F	F	F	F	F	T	T	T
6	F	F	T	F	F	T	T	T
7	F	F	T	F	F	T	T	T

(b) Conflict table after isolation



(c) Segment graph after isolation

Fig. 6. Simple design example with isolated instances.

Following this, we can resolve the false conflict by having separate behavior definitions `B` for the instances `b1` and `b2`, such as `B_iso0` and `B_iso1` in Fig. 6(a). Here, two different segments seg_3 and seg_6 start from the corresponding `waitfor 2` in `B_iso0` and `B_iso1`, respectively. Now seg_3 writes to `Main.sum1`, seg_6 writes to `Main.sum2`, and there is no conflict. Consequently, the scheduler can safely execute th_{b1} in seg_3 in parallel to th_{b2} in seg_6. The same argument holds for seg_4 and seg_7, as indicated in the conflict table in Fig. 6(b).

We will use the term *instance isolation* [20] to describe the model modification demonstrated here. *Isolating* an instance means to create a unique copy of its class definition (behavior or channel) so that each instance has its own code segments.

A. Instance Isolation without Code Duplication

Instance isolation essentially creates additional scheduling statements (i.e. *par*, *wait*, and *wait-for-time*) for specific instances. Different segments are then generated for these statements and the variable access information is separated.

As described above, isolation textually creates an additional class with a different name (e.g. *B_iso1* in Fig. 6(a)), but all statements remain the same as in the original definition (e.g. `B` in Fig. 4(a)). This code duplication eases the understanding of isolation, but is not desirable for an implementation due to its waste of resources.

In the following section, we propose a compiler optimization that uses the original statements without duplication, but still creates separate segments for different instances and attaches the variable access information accordingly. Thus, the size of the design model and the program segment in the binary executable will not increase.

To achieve this, we distinguish instances by unique identifiers (i.e. *instID*) which the compiler maintains and passes to the simulator. At run time, the simulator can then use the current instance identifier to distinguish the segments even though they execute the same program code. In Fig. 6(a) for example, seg_3 of `b1` and seg_5 of `b2` originate from the same statement `waitfor 2` in line 12 (see also Fig. 4(a)).

Also, when processing a function call in a segment, we need to analyze the function body to obtain its variable access information since these variables affect the same segment. Moreover, we need to analyze function definitions also to connect the segment graph correctly if additional segments are started due to segment boundary statements inside the function body. This analysis could grow exponentially with the size of the design if there are frequent deep function calls. We avoid this by *caching* the information obtained during the function analysis, so that we can reuse this data the next time the same function is called. The time complexity of our compile-time analysis is therefore practically linear to the size of the code.

We present the detailed algorithm for the static code analysis, which automatically isolates the instances in the design "on-the-fly" without duplicated source code and with only minimal compile time increase, in the next section.

V. Static Conflict Analysis in the Compiler

At compile time, we use static analysis of the application source code to determine whether or not any conflicts exist

Algorithm 3 Build the Segment Graph

```
 1: newSegList = BuildSegmentGraph(currSegList, stmnt)
 2: {
 3: switch (stmnt.type) do
 4: case STMNT_COMPOUND:
 5:   newL = currSegList
 6:   for all subStmnt ∈ Stmnt do
 7:     newL = BuildSegmentGraph(newL, subStmnt)
 8:   end for
 9: case STMNT_IF_ELSE:
10:   ExtendAccess(stmnt.conditionVar, currSegList)
11:   tmp1 = BuildSegmentGraph(currSegList, subIfStmnt)
12:   tmp2 = BuildSegmentGraph(currSegList, subElseStmnt)
13:   newL = tmp1 ⋃ tmp2
14: case STMNT_WHILE:
15:   ExtendAccess(stmnt.conditionVar, currSegList)
16:   helperSeg = new Segment
17:   tmpL = new SegmentList; tmpL.add(helperSeg)
18:   tmp1 = BuildSegmentGraph(tmpL, subWhileStmnt)
19:   if helperSeg ∈ tmp1
20:     then remove helperSeg from tmp1 end if
21:   for all Segment s ∈ tmp1⋃ currSegList do
22:     s.nextSegments ⋃= helperSeg.nextSegments
23:   end for
24:   newL = currSegList ⋃ tmp1; delete helperSeg
25: case STMNT_PAR:
26:   newSeg = new Segment; totalSegments ++
27:   newEndSeg = new Segment; totalSegments ++
28:   for all Segment s ∈ currSegList do
29:     s.nextSegments.add(newSeg) end for
30:   tmpL = new SegmentList; tmpL.add(newSeg)
31:   for all subStmnt ∈ stmnt do
32:     BuildSegmentGraph(tmpL, subStmnt)
33:     for all Segment s ∈ tmpL do
34:       s.nextSegments.add(newEndSeg) end for
35:   end for
36:   newL= new SegmentList; newL.add(newEndSeg)
37: case STMNT_WAIT:
38: case STMNT_WAITFOR:
39:   newSeg = new Segment; totalSegments++
40:   for all Segment s ∈ currSegList do
41:     s.nextSegments.add(newSeg); end for
42:   newL = new SegmentList; newL.add(newSeg)
43: case STMNT_EXPRESSION:
44:   if stmnt is a function call f() then
45:     newL = BuildFunctionSegmentGraph(currSegList, fct)
46:   else
47:     ExtendAccess(stmnt.expression, currSegList)
48:     newL = currSegList
49:   end if
50: case ...: /* other statements omitted for brevity */
51: end switch
52: return newL;
53: }
```

between the segments. Overall, the compiler traverses the application's control flow graph following all branches, function calls, and thread creation points, and recursively builds the corresponding segment graph. The segment graph is then used to build the access lists and desired conflict tables.

To present the algorithm in detail, we need to introduce a few definitions:

- **cacheMultiInfo**: a Boolean flag at each function for caching multiple sets of information for different instances; *true* if new segments are created or interface methods[3] are called in this function; *false* otherwise (default).
  Note that, if *cacheMultiInfo* is *false*, the cached information is the same for all instances that call the function.
- **cachedInfo**: cached information, as follows:
  - **instID**: instance identifier, e.g. `Main.b1`.
  - **dummyInSeg**: a dummy segment as the initial segment of this set of cached information.
  - **carryThrough**: a Boolean flag; *true* when the input segment carries through the function and is part of the output segments; *false* otherwise. For example for `Main.b1`, `B.main.carrythrough` = *false* since seg_1 is connected to seg_3 and will not carry through `B.main`.
  - **outputSegments**: segments (without the input segment) that will be the output after analyzing this function. For example for `Main.b1`, `B.main.outputSegments`=$\{seg_3, seg_4\}$.
  - **segAccessLists**: list of segment access lists. The segments here are a subset of the global segments in the design. This list only contains the segments that are accessed by *instID.fct*.

During the analysis, when a statement is processed, there is always an input segment list and an output segment list. For example, for the `while` loop (Fig. 4(a), line 13), the input segment list is $\{seg_3\}$ and the output segment list is $\{seg_3, seg_4\}$. To start, we create an initial segment (i.e. seg_0) for the design as the input segment of the first statement in the program entrance function, i.e. `Main.main()`.

A. Algorithm for Static Conflict Analysis

Our optimized algorithm for the static code analysis consists of four phases, as follows:

- **Input:** Design model (e.g. from file `design.sc`).
- **Output:** Segment graph and segment conflict tables.
- **Phase 1:** Use Algorithm 3 to create the *global* segment graph, where Algorithm 4 lists the details of function *BuildFunctionSegmentGraph()* at line 45.
  If no function needs to be cached for multiple instances, the complexity of this phase is $O(n)$ where n is the number of statements in the design.
- **Phase 2:** Use Algorithm 5 to build a *local* segment graph with segment access lists for each function. Here, we only add variables accessed in this function to the segment access lists. We do not follow function calls in this phase.
  The complexity of this phase is $O(n_s)$ where n_s is the number of statements in the function definition.
  Fig. 7(a) illustrates this for the example in Fig. 6. We do not follow function calls to `B.main` from `Main.main` but connect a *dummyInSeg* node instead. We also create two sets of `main` function cached information, shown in Fig. 7(b) and Fig. 7(c), for the two instances of `B` since segment boundary nodes are created when calling `B.main`.
  Note that we do not know yet the instance path of the member variables in this phase. Therefore, we use port variable `sum` instead of its real mapping `Main.sum1` and `Main.sum2` here.

[3]Interface method definitions differ for different types of instances. Details are omitted here for brevity.

Algorithm 4 Code analysis for out-of-order PDES, Phase 1: BuildFunctionSegmentGraph(currSegList, fct)

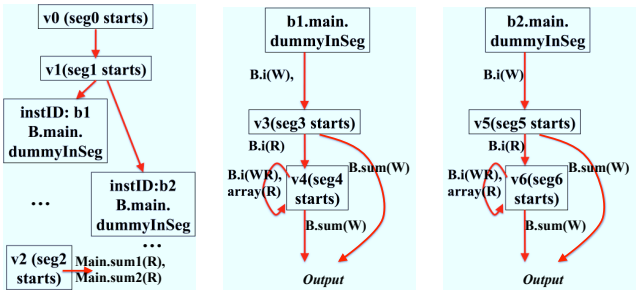
```
1:  if fct is first called then
2:    BuildSegmentGraph(currSegList, fct.topstmnt);
3:    if new segment nodes are created in fct then
4:      set fct.cacheMultiInfo = true;
5:      cache function information with current instID;
6:    else
7:      cache function information without instID; endif
8:  else /*fct has already been analyzed*/
9:    if fct.cacheMultiInfo = false then
10:     /*no segments are created by calling this function*/
11:     use the cached information of fct;
12:   else /*new segments are created by calling this function*/
13:     if current instID is cached then
14:       use the cached information of fct.cacheinfo[instID];
15:     else
16:       BuildSegmentGraph(currSegList, fct.topstmnt);
17:       cache function information with current instID; endif
18:   endif
19: endif
```

Algorithm 5 Code analysis for out-of-order PDES, Phase 2

```
1:  for all function fct do
2:    Traverse the control flow of fct.
3:    Create and maintain a local segment list localSegments.
4:    Use fct.dummyInSeg as the initial input segment.
5:    For each statement, add variables with their access type into
      proper segments.
6:    for all function calls inst.fct or fct do
7:      Do not follow the function calls. Just register
        inst.fct.dummyInSeg or fct.dummyInSeg as the input
        segments of the current statement and indicate that inst.fct
        or fct is called in the input segments.
8:      Add the output segments of the function call to localSeg-
        ments and use the output segments as the input of the next
        statement in the current function.
9:    end for
10: end for
```

- **Phase 3:** Build the *complete* segment access lists for each function. Here, we propagate function calls and add all accessed variables to the cached segment access lists cascaded with proper instance paths.
  For our example, `b1` is cascaded to the instance paths of the member variables accessed in segment `B.main.dummyInSeg` for instance `b1` when analyzing `Main.main` (if necessary)[4]. We also use the function caching technique here to reduce the complexity of the analysis. The complexity of this phase is $O(n_g)$ where n_g is the size of the segment graph for each function.
- **Phase 4:** Collect the access lists for each segment in `Main.main` and add them to the global segment access lists. Since `Main.main` is the program entry (or root function), all the segments are in its local segments and the member variables have complete cascaded instance paths in the segment access lists. The actual mapping of port variables can now be found according to their instance path. The complexity of this phase is $O(n_l)$ where n_l is the size of the local segment access list of `Main.main`.

[4]Regular member variables accessed in different instances will not cause data hazards. Only port variables and interface member variables need the instance path for tracing the real mapping later.



(a) Main.main (b) B.main (Main.b1) (c) B.main (Main.b2)

Fig. 7. Function-local segment graphs and variable access lists for the example in Fig. 6.

In summary, the algorithm generates precise segment conflict information using implicit instance isolation. The overall complexity is, for practical purposes, linear to the size of the analyzed design.

As a limitation, we currently do not support the analysis of recursive function calls (future work).

VI. Out-of-order PDES Scheduling with Predictions

To be safe and compliant with SLDL semantics, our out-of-order scheduling is conservative in conflict checking. So far, we have only considered *current* potential conflicts among the thread segments, as well as potential conflicts in the next step ahead. This can disqualify some threads from being issued out of the order that do not pose any real hazards when we analyze their behavior *in the future*.

For example, let's look again at the first example of a direct timing hazard in Table II (page 6). Here, the conservative scheduler does not let thread th_1 run in parallel with th_2 because th_2 enters another segment before the current time of th_1. Assuming no information about this future segment of th_2, the scheduler makes the safe decision to hold th_2 back. However, if the next segment(s) of th_2 and the corresponding potential conflicts would be known, the scheduler could *predict* whether or not th_2 will have any conflicts with th_1 in the future (until the time of th_1). If not, it would still be safe to issue th_2 at that scheduling step.

Again, we face a *false conflict*. The key to reduce such cases here is to provide more information to the scheduler about the future segments for each thread. In other words, if the scheduler can predict the future thread state, it can eliminate more false conflicts.

Following this thought, we find that the segment graph actually provides a complete static picture of the potential execution behavior for each thread in the model. This makes it possible to predict the future segments and their conflicts for a thread at any scheduling step.

We have extended our implementation in this regard to look ahead one more step. In addition to the conflict tables defined in Section III-A, which provide the variable access information for the current thread segment and the time cycle information of the next thread segment, we build at compile-time two more tables. These tables provide the variable access information and the time advance information for the next *two* segments for a given thread. As a result, the extended scheduler can therefore look one more step ahead and make better decisions.

Table V shows experimental results that confirm this idea. With prediction of two steps ahead, the simulation of our H.264 encoder model gains 42.4% speedup.

Generally, this type of prediction can be extended to predict n steps ahead ($n \geq 1$). We will explore this in future work.

VII. Experiments and Results

We have implemented the proposed out-of-order parallel simulator in a SpecC[5]-based system design environment [21] and conducted experiments on four multi-media systems that we built in-house based on standard reference source code.

To demonstrate the benefits of our out-of-order PDES, we compare the compiler and simulator run times with the traditional single-threaded reference and a regular parallel implementation [5] without out-of-order scheduling.

For our experiments, we use 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[6]. Each CPU contains 6 parallel cores, each of which supports 2 hyper-threads per core. Thus, in total the server hardware supports up to 24 threads running in parallel.

TABLE III
Experimental results for the abstract DVD player example.

Simulator:	Single-thread reference	Multi-core	
		Synch. PDES	OoO PDES
Compile time	1.54s	1.53s / +0.7%	1.63s / **-5.5%**
Simulator time	12.75s	12.74s / +0.1%	5.85s / **+117.9%**
# segments N		62	
total conflicts in $CTab[N,N]$		168/3844 (4.4%)	
# threads issued		1008	
# threads issued out-of-order		**722 (71.63%)**	

A. An Abstract Model of a DVD Player

Our first experiment uses a DVD player model similar to the model discussed in Section I-A, where a H.264 video and a MP3 audio stream are decoded in parallel. However, this model features four parallel slice decoders which decode separate slices in a H.264 frame simultaneously. Specifically, the H.264 stimulus reads new frames from the input stream and dispatches its slices to the four slice decoders. A synchronizer block completes the decoding of each frame and triggers the stimulus to send the next one. The blocks in the model communicate via double-handshake channels.

[5]Due to its similarity, our results are equally applicable to SystemC.

[6]To ensure consistent timing measurements, we have disabled the dynamic frequency scaling and turbo mode of the processors.

According to profiling results, the workload ratio between decoding one H.264 frame with 704x566 pixels and one 44.1kHz MP3 frame is about 30:1. Further, about 70% of the decoding time is spent in the slice decoders (17.5% per unit).

Table III shows the statistics and measurements for this model. Note that the conflict table is very sparse, allowing 71.63% of the threads to be issued out-of-order. While the synchronous PDES cannot gain performance due to in-order time barriers and synchronization overheads, our out-of-order simulator shows more than twice the simulation speed.

TABLE IV
Experimental results for JPEG and H.264 examples.

JPEG Image Encoder				
TLM abstraction:	spec	arch	sched	net
# segments N	54	51	54	55
Total conflicts in $CTab[N,N]$	208/2916 (7.1%)	208/2601 (8.0%)	208/2916 (7.1%)	219/3025 (7.2%)
# threads issued	274	273	274	4861
# threads issued out-of-order	**178 (65.0%)**	**178 (65.2%)**	**171 (62.4%)**	**1976 (40.7%)**
H.264 Video Decoder				
TLM abstraction:	spec	arch	sched	net
# segments N	72	69	72	74
Total conflicts in $CTab[N,N]$	439/5184 (8.47%)	443/4761 (9.30%)	443/5184 (8.55%)	410/5476 (7.49%)
# threads issued	97176	97175	97177	97181
# threads issued out-of-order	**21856 (22.49%)**	**21845 (22.48%)**	**21781 (22.41%)**	**21807 (22.44%)**

B. A JPEG Encoder Model

Our second experiment uses a JPEG image encoder model. The stimulus reads a BMP color image with 3216x2136 pixels and performs color-space conversion from RGB to YCbCr. Since encoding of the three color components (Y, Cb, Cr) is independent, our JPEG encoder performs the DCT, quantization and zigzag modules for the colors in parallel, followed by a sequential Huffman encoder at the end. The JPEG monitor collects the encoded data and stores it in the output file.

To show the increased simulation speed also for models at different abstraction levels, we have created four models (*specification*, *architecture mapped*, *OS scheduling refined*, *network linkage refined*) with increasing amount of implementation detail, down to a network model with detailed bus transactions. Table IV lists the experimental results and shows that, for the JPEG encoder, about half of all threads can be issued out-of-order.

Table V shows the corresponding compiler and simulator run times. While the compile time increases similar to the synchronous PDES compiler, the simulation speed improves by about 150%, almost 5 times the gain of the synchronous parallel simulator. Moreover, with the scheduling prediction feature, the simulation speed improves further to gain almost 200%. In other words, for the JPEG encoder, our simulator is almost 3 times as fast as the reference implementation.

C. A Detailed H.264 Decoder Model

Our third experiment simulates a complex parallel video decoder based on the H.264/AVC standard [22]. While this

TABLE V
EXPERIMENTAL RESULTS FOR FOUR STANDARD MULTI-MEDIA APPLICATION EXAMPLES.

Simulator:		Single-thread reference		Multi-core					
				Synch. PDES		OoO PDES		OoO PDES with prediction	
		Compile time [sec]	Simulator time [sec]	Compile time [sec] / Speedup	Simulator time [sec] / Speedup	Compile time [sec] / Speedup	Simulator time [sec] / Speedup	Compile time [sec] / Speedup	Simulator time [sec] / Speedup
DVD player		1.54	12.75	1.53 / +0.7%	12.74 / +0.1%	1.63 / -5.5%	5.85 / **+117.9%**	1.70 / -9.4%	5.84 / **+118.3%**
JPEG Encoder	spec	4.34	1.67	4.09 / +6.1%	1.28 / +30.5%	4.39 / -1.1%	0.64 / **+160.9%**	4.44 / -2.3%	0.58 / **+187.9%**
	arch	4.85	1.70	4.95 / -2.0%	1.32 / +28.8%	5.03 / -3.6%	0.67 / **+153.7%**	4.89 / -0.8%	0.57 / **+198.2%**
	sched	4.83	1.69	5.00 / -3.4%	1.29 / +31.0%	5.06 / -4.5%	0.69 / **+144.9%**	5.74 / -9.2%	0.59 / **+186.4%**
	net	5.15	2.74	5.74 / -10.3%	2.06 / +33.0%	5.55 / -7.2%	1.14 / **+140.4%**	5.76 / -10.6%	0.93 / **+194.6%**
H.264 Decoder	spec	15.13	123.88	15.41 / -1.8%	128.85 / -3.9%	15.36 / -1.5%	82.13 / **+50.8%**	15.68 / -3.5%	82.72 / **+49.8%**
	arch	15.19	125.76	15.48 / -1.9%	128.67 / -2.3%	15.47 / -1.8%	81.17 / **+54.9%**	15.71 / -3.3%	82.42 / **+52.6%**
	sched	15.25	124.03	15.68 / -2.7%	130.61 / -5.0%	16.05 / -5.0%	82.56 / **+50.2%**	16.46 / -7.4%	82.59 / **+50.2%**
	net	15.89	126.20	15.98 / -0.6%	128.25 / -1.6%	16.63 / -4.4%	81.13 / **+55.6%**	16.54 / -3.6%	82.55 / **+52.9%**
H.264 Encoder		24.69	2732.36	26.12 / -5.5%	2720.83 / +0.4%	29.75 / -17.0%	2725.25 / **+0.3%**	30.61 / -19.3%	1918.89 / **+42.4%**

model is similar at the highest level to the video part of the abstract DVD player, it contains many more blocks at lower levels which implement the complete H.264 reference application consisting of about 40,000 lines of code. Internally, each slice decoder consists of complex H.264 decoder functions including entropy decoding, inverse quantization and transformation, motion compensation, and intra-prediction. For our simulation, we use a video stream of 1079 frames, 1280x720 pixels per frame, each with 4 slices of equal size.

Our simulation results for this industrial-size design are listed in the lower half of Table IV and Table V, again for four models at different abstraction levels, including a network model with detailed bus transactions.

While the synchronous PDES actually loses simulation speed, our proposed simulator shows more than 50% gain since about a quarter of the threads can be issued out-of-order (see Table IV). Even for a model with this large complexity, the increase of compilation time due to static conflict analysis is well below 10%.

D. H.264/AVC Video Encoder with Parallel Motion Search

Our fourth application is a parallelized video encoder based on the H.264/AVC standard. Intra- and inter-frame prediction are applied to encode an image according to the type of the current frame. During inter-frame prediction, the current image is compared to the reference frames in the decoded picture buffer and the corresponding error for each reference image is obtained.

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. Our test stream is a video of 95 frames with 176x144 pixels per frame. The number of B-slices between the I-slices or P-slices is 4. That is, among every five consecutive frames, four frames need inter-frame prediction.

Table V shows that synchronous PDES and OoO PDES without prediction can hardly achieve any significant speedup. However, the proposed prediction feature of OoO PDES achieves simulation acceleration with a speed up of 42.4%.

VIII. CONCLUSIONS AND FUTURE WORK

High simulator performance is critical for the efficient validation of transaction-level design models. In this article, we have presented a novel out-of-order scheduling technique for multi-core parallel simulation of system-level design models with hardware and software components. Our approach breaks the simulation-cycle barrier of traditional simulation by localizing the simulation time for parallel threads, carefully delivering notified events, and handling a dynamically managed set of simulation queues. Potential data conflicts between parallel threads are prevented by conservative and predictive compile-time analysis based on the segment graph of the application. Using fast conflict table lookup, our out-of-order scheduler can quickly make decisions at run-time and issue more parallel threads than synchronous PDES, resulting in significantly higher simulation speed on multi-core hosts.

Our experimental results show clearly that, with only a small increase in compile time, our simulator is significantly faster than the traditional single-threaded reference implementation, as well as a synchronous PDES implementation.

Our out-of-order PDES technique fully maintains SLDL simulation semantics and is applicable, without loss of accuracy, to C-based system-level models at any abstraction level.

In future work, we plan to further extend and optimize the static conflict analysis and prediction technique.

ACKNOWLEDGMENT

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer, 2002.

[2] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*. Kluwer, 2000.

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

[4] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," in *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pp. 80–87, 2009.

[5] 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.

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

[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] 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.

[9] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele, "Scalably Distributed SystemC Simulation for Embedded Applications," in *International Symposium on Industrial Embedded Systems, 2008. SIES 2008.*, pp. 271–274, June 2008.

[10] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, October 2003.

[11] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *Proceedings of the Design Automation Conference (DAC)*, 2011.

[12] A. Gerstlauer, "Host-Compiled Simulation of Multi-Core Platforms," in *Proceedings of the International Symposium on Rapid System Prototyping (RSP), Washington, DC, June 2010*.

[13] "SystemC TLM-2.0." http://www.accellera.org/downloads/standards/systemc/tlm.

[14] S. Sirowy, C. Huang, and F. Vahid, "Online SystemC Emulation Acceleration," in *Proceedings of the Design Automation Conference (DAC)*, 2010.

[15] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: A fast SystemC simulator on GPUs," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2010.

[16] 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.

[17] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, M. H. D. Wood, S. Huss-Lederman, and J. Larus, "Wisconsin Wind Tunnel II: a fast, portable parallel architecture simulator," *IEEE Concurrency*, vol. 8, pp. 12–20, Oct-Dec 2000.

[18] D. Yun, S. Kim, and S. Ha, "A Parallel Simulation Technique for Multicore Embedded Systems and Its Performance Analysis," *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, vol. 31, pp. 121–131, Jan 2012.

[19] 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.

[20] W. Chen and R. Dömer, "An Optimizing Compiler for Out-of-Order Parallel ESL Simulation Exploiting Instance Isolation," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[21] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski, "System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 647953, p. 13 pages, 2008.

[22] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, pp. 560 –576, july 2003.