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

Weiwei Chen, *Member, IEEE,* Xu Han, Che-Wei Chang, Guantao Liu, and Rainer Dömer, *Member, IEEE*

***Abstract*—The validation of system models at the transaction-level typically relies on discrete event (DE) simulation. In order to reduce simulation time, parallel discrete event simulation (PDES) can be used by utilizing multiple cores available on today's host PCs. However, the total order of time imposed by regular DE simulators becomes a bottleneck that severely limits the benefits of parallel simulation. In this paper, we present a new out-of-order (OoO) PDES technique for simulating transaction-level models on multicore 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 OoO PDES technique shows high performance gains in simulation speed with only a small increase in compile time. Using six embedded application examples, we also show the speed trade-off for multicore PDES based on different multithreading libraries.**

***Index Terms*—Parallel discrete event simulation (PDES), system-level description languages (SLDLs), system-level design and validation, transaction level modeling.**

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, 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 paper, 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 simulation speed.

Due to the inexpensive availability of parallel processing capabilities in today's multicore hosts, recently parallel DE simulation (PDES) [3] has gained attention again. Using operating system (OS) 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 paper, we describe an advanced PDES approach, called out-of-order PDES (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 multicore hosts.

While the reference simulators issue only one thread at a 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, this 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 cycle. Only then the simulator advances to the

Manuscript received November 15, 2013; revised April 21, 2014; accepted July 15, 2014. Date of current version November 18, 2014. This work was supported by the National Science Foundation (NSF) under Research Grant NSF Award 0747523. This paper was recommended by Associate Editor S. Parameswaran.

The authors are with the Center for Embedded Computer Systems, University of California, Irvine, Irvine, CA 92697 USA (e-mail: weiweic@uci.edu; hanx@uci.edu; cheweic@uci.edu; guantaol@uci.edu; doemer@uci.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2014.2356469

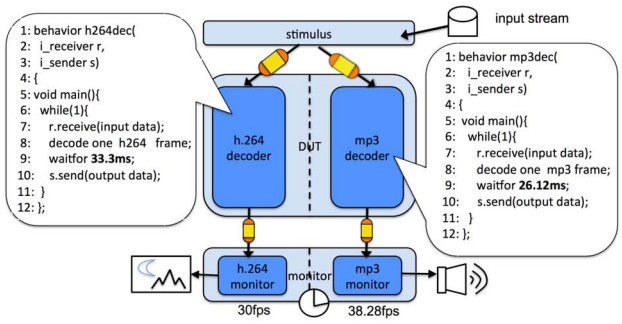


Fig. 1. High-level DVD player model with video and audio streams.

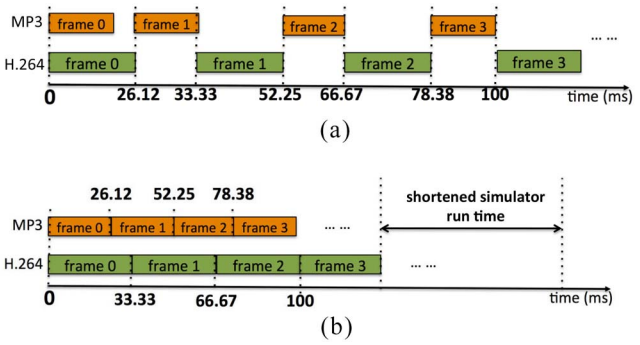


Fig. 2. Scheduling of the high-level DVD player model. (a) Synchronous PDES schedule. (b) OoO PDES schedule.

next delta or time cycle. 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.

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 frames/s for video (delay 33.3 ms) and 38.28 frames/s for audio (delay 26.12 ms).

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.

In this paper, we break the cycle barrier and let independent threads run OoO 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 OoO 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 [3], [6], and [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 multiprocessing (SMP) architectures with global simulation time and delta-cycle notion, such as [4], [5], and [8]. An extended SystemC simulator described in [4] and [8] schedules multiple OS kernel threads in PDES fashion on multicore 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 [6] and [9], is a natural extension of PDES. Distributed simulation breaks the design model into modules on geographically distributed hosts and then runs the simulation in parallel. However, model partitioning is difficult and network speed becomes a bottleneck due to frequently needed communication.

Related work on improving simulator speed in the broader sense can be categorized into software modeling and specialized hardware approaches. Software techniques include the general idea of TLM [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. In contrast, OoO PDES 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 multithreading model for GPU-based simulation. Reference [16] presents a methodology to parallelize SystemC simulation across multicore CPUs and GPUs. For such approaches, model partitioning is difficult on the heterogeneous simulator units.

Other simulation techniques change the simulation 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. This technique significantly speedup the simulation at the cost of timing accuracy. Another example [18] introduces a simulation backplane to handle the synchronization between wrapped simulators and analyzes the system to optimize the period of synchronization message transfers. It sacrifices the timing accuracy for interrupt services for speed benefits of parallel simulation without causing any harm on the functional correctness of the design.

In contrast to the approaches above, the 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 [19]–[21], this paper extends the dependency analysis at compile- and runtime. We add a detailed description of the conflict analysis for variables at different scopes and OoO hazards due to the localized timing (Section III-B). We also integrate a novel prediction technique (Section V) and a hybrid multithreading library (Section VI), which further improve the simulator speed. We show this with added experimental results for a new H.264 video encoder model and two highly parallel graphics applications (Section VII).

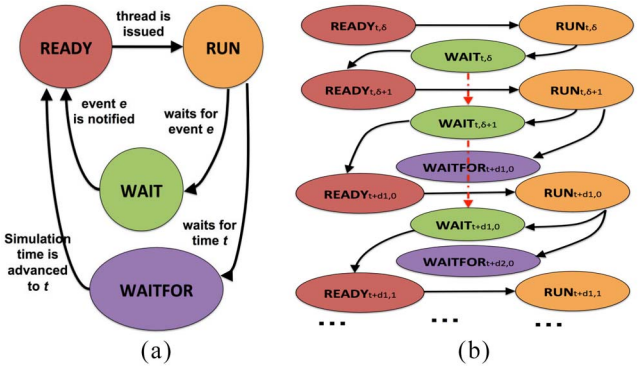


Fig. 3. States and transitions of simulation threads (simplified). (a) Static states in synchronous PDES. (b) Dynamic states in OoO PDES.

II. OoO Parallel Simulation

In contrast to synchronous PDES, which imposes a total order on event processing and time advances, we now propose OoO PDES where timing is only partially ordered. We localize the global simulation time for each thread and allow threads without potential data or event conflicts to run ahead of time, maximizing multicore utilization.

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, δ = delta-cycle. We order time stamps as follows.
   a) *Equal:* $(t_1, \delta_1) = (t_2, \delta_2)$, iff $t_1 = t_2$, $\delta_1 = \delta_2$.
   b) *Before:* $(t_1, \delta_1) < (t_2, \delta_2)$, iff $t_1 < t_2$, or $t_1 = t_2$, $\delta_1 < \delta_2$.
   c) *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-based 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 the following.
   a) QUEUES = {READY, RUN, WAIT, WAITFOR, JOINING, COMPLETE}.
   b) READY = $\cup$READY$_{t,\delta}$, READY$_{t,\delta}$ = $\{th \mid th$ is ready to run at $(t, \delta)\}$.
   c) RUN = $\cup$RUN$_{t,\delta}$, RUN$_{t,\delta}$ = $\{th \mid th$ is running at $(t, \delta)\}$.
   d) WAIT = $\cup$WAIT$_{t,\delta}$, WAIT$_{t,\delta}$ = $\{th \mid th$ is waiting since (t, δ) for events (id_e, t_e, δ_e), where $(t_e, \delta_e) \geq (t, \delta)\}$.
   e) WAITFOR = $\cup$WAITFOR$_{t,\delta}$, WAITFOR$_{t,\delta}$ = $\{th \mid th$ is waiting for simulation time advance to $(t, 0)\}$ Note that δ is always 0 for the WAITFOR$_{t,\delta}$ queues.
   f) JOINING = $\cup$JOINING$_{t,\delta}$, JOINING$_{t,\delta}$ = $\{th \mid th$ created child threads at (t, δ) and waits for them to complete$\}$.
   g) COMPLETE = $\cup$COMPLETE$_{t,\delta}$, COMPLETE$_{t,\delta}$ = $\{th \mid th$ completed its execution at $(t, \delta)\}$.
   Note that for efficiency our implementation orders these sets by increasing time stamps.
5) Initial state at the beginning of simulation.
   a) $t = 0$, $\delta = 0$.
   b) RUN = RUN$_{0,0}$ = $\{th_{\text{root}}\}$.
   c) READY = READY$_{0,0}$ = WAIT = WAIT$_{0,0}$ = WAITFOR = WAITFOR$_{0,0}$ = COMPLETE = COMPLETE$_{0,0}$ = JOINING = JOINING$_{0,0}$ = $\emptyset$.
6) Simulation invariants.
   Let THREADS be the set of all existing threads. Then, at any time, the following conditions hold.
   a) THREADS = READY $\cup$ RUN $\cup$ WAIT $\cup$ WAITFOR $\cup$ JOINING $\cup$ COMPLETE.
   b) $\forall$ A, B $\in$ QUEUES, $A, B \neq \emptyset$: $A \neq B \Leftrightarrow A \cap B = \emptyset$.

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.

1) READY$_{t,\delta}$ $\rightarrow$ RUN$_{t,\delta}$: The thread becomes runable (is issued).
2) RUN$_{t,\delta}$ $\rightarrow$ WAIT$_{t,\delta}$: The thread calls wait for an event.
3) RUN$_{t,\delta}$ $\rightarrow$ WAITFOR$_{t',0}$, where $t < t' = t + delay$: The thread calls waitfortime(delay).
4) RUN$_{t,\delta}$ $\rightarrow$ JOINING$_{t,\delta}$: The thread creates its child threads and waits for them to complete.
5) RUN$_{t,\delta}$ $\rightarrow$ COMPLETE$_{t,\delta}$: The thread finishes the work and completes.
6) 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').
7) JOINING$_{t,\delta}$ $\rightarrow$ READY$_{t',\delta'}$, where $(t, \delta) \leq (t', \delta')$: Child threads completed and their parent becomes ready to run again.
8) 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

Algorithm 1 OoO 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
```

empty and there are no WAIT or WAITFOR queues with earlier timestamps, the scheduler deletes $\text{READY}_{t,\delta}$ and $\text{RUN}_{t,\delta}$, as well as any expired events with the same timestamp $\text{EVENTS}_{t,\delta}$ (lines 8–14 in Algorithm 1).

B. OoO 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 $(t_e, \delta_e + 1)$ 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 OoO 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. OoO 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 (SG)

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.

1) *Segment* seg_i*:* Source code statements executed by a thread between two scheduling steps.
2) *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 the following.

1) *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 $\text{seg}_i = \cup e_{ij}\}$.

Fig. 4(a) shows a simple example written in SpecC SLDL. The design contains two parallel instances, `b1` of type `B1` and `b2` of type `B2`. Both `b1` and `b2` contain loops with computation, synchronization, and simulation time advances. Finally, the `Main` behavior prints the value of a variable to the screen.

From the corresponding CFG in Fig. 4(b), we derive the SG in Fig. 4(c). The graph shows nine 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 19) where the scheduler is called to create two threads for `b1` and `b2`. Two new segments, represented by the nodes *v1* and *v5*, are created for `b1` and `b2`. Instance `b1` first reaches `waitfor 1` (line 8) via *v1*→*v2*, or skips the `while` loop (line 7) to reach `waitfor 3` (line 12) via *v1*→*v4*. While in the loop (line 7), the execution reaches `wait e` (line 9) via *v2*→*v3* after `waitfor 1` (line 8), and then `waitfor 3` (line 12) via *v3*→*v4* when it exits the loop.

Similarly, instance `b2` reaches `waitfor 2` (line 8) via *v5*→*v6* and `waitfor 4` (line 12) via *v6*→*v7* after the loop. Its execution completes at the end of the `par` statement via *v7*→*v8*. Finally, after *v8* the execution terminates. Note that a code statement can be part of multiple segments. For example, line 7 for `B1` belongs to both seg_1 and seg_3. Note also that threads execute one segment in each scheduling step.

B. Static Conflict Analysis

To comply with SLDL execution semantics, threads must not execute in parallel or OoO if their segments pose any hazard toward validity of data, event delivery, or timing.

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

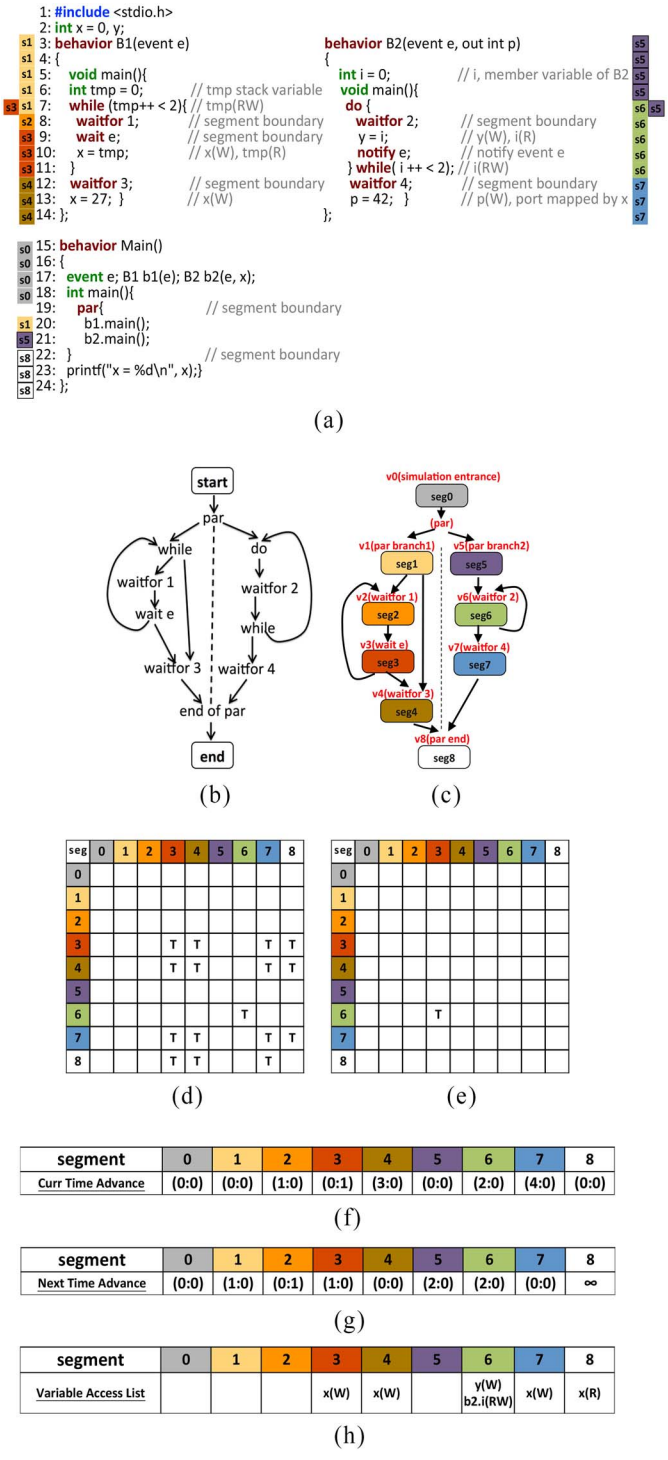


Fig. 4. Simple design example with corresponding control flow graph (CFG), SG, conflict tables (CTs), and time advance tables. (a) Example source code in SpecC. (b) CFG. (c) SG. (d) Data CT. (e) Event notification table (NT). Time advance table for the (f) current segment (CTime) and (g) next segment (NTime). (h) Segment variable access lists.

Fig. 5 shows a simple example of a WAW conflict where two parallel threads th_1 and th_2 start at the same time but 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

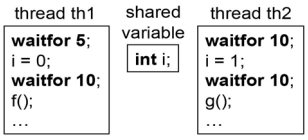


Fig. 5. WAW conflict between two parallel threads.

nondeterministic. Thus, we must not schedule th_1 and th_2 OoO. 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 CT $CT[N, N]$ where N is the total number of segments in the application code: $CT[i, j]$ = true, iff there is a potential data conflict between the segments seg_i and seg_j; otherwise, $CT[i, j]$ = false.

To build the CT, 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 CT $CT[N, N]$ by comparing the access lists for each segment pair. If two segments seg_i and seg_j share any variable with access type (W) or (RW), or there is any Ptr by seg_i or seg_j, then we mark this as a potential conflict.

Fig. 4(d) shows the data CT for the example in Fig. 4(a). Here, for instance, seg_3 has a data conflict (WAW) with seg_7, since, seg_3 writes `x` (line 10) and seg_7 writes `x` (line 13, port `p` is mapped to `x`).

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.

We distinguish and handle the following cases.

1) *Global Variables, e.g., `x, y` in Line 2:* This case is discussed above and can be handled directly as tuple (Symbol, AccessType).
2) *Local Variables, e.g., `tmp` in Line 6 for Behavior `B1`:* Local variables are stored on the stack and cannot be shared between different threads. Thus, they can be ignored in the access lists.
3) *Instance Member Variables, e.g., `i` in Line 5 for Behavior `B2`:* Since, classes can 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 instance of `B2`'s variable `i` is `Main.b2.i`.[2]
4) *Port Variables, e.g., `p` in Line 3:* 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, `x` is the actual variable mapped to port `Main.b2.p`. Note that tracing ports to their connected variables requires the compiler to follow port mappings through the structural hierarchy of the design model

[2]In SpecC, there is always only one instance of the `Main` behavior. Thus, we can omit `Main` from the instance path.

TABLE I
TIME ADVANCES AT SEGMENT BOUNDARIES

Segment boundary	Time Increment		Add to (t', δ')
wait event	increment by one delta cycle	(0:1)	$(t', \delta' + 1)$
waitfor t	increment by time t	(t:0)	$(t' + t, 0)$
par/par_end	no time increment	(0:0)	(t', δ')

TABLE II
EXAMPLES FOR DIRECT AND INDIRECT TIMING HAZARDS

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

which is possible when all used components are part of the current translation unit.

5) *Pointers:* We currently do not perform pointer analysis (future work). For now, we conservatively mark all segments with Ptrs as potential conflicts.

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

We define an event NT $NT[N, N]$ where N is the total number of segments: $NT[i, j]$ = true, iff segment seg_i notifies an event that seg_j is waiting for; otherwise, $NT[i, j]$ = false. Note that in contrast to the data CT, the event NT is not symmetric.

Fig. 4(e) shows the event NT for the simple example. For instance, $NT[6, 3]$ = true since seg_6 notifies the event e (line 10) which seg_3 waits for (line 9).

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.

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 8) at the beginning of seg_6 sets CTime[6] = (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

$$\text{NTime}[i] = \min\{\text{CTime}[j], \forall \text{seg}_j \text{ which follow } \text{seg}_i\}.$$

For example, Fig. 4(g) lists NTime[1] = (1:0), since, seg_1 is followed by seg_2 [increment (1:0)] and seg_4 [increment (3:0)].

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[3] 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.

Algorithm 2 Conflict Detection in Scheduler

```
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: }
```

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. 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 between the segments. Overall, the compiler traverses the application's CFG following all branches, function calls, and thread creation points, and recursively builds the corresponding SG. The SG is then used to build the access lists and desired CTs.

[3]We predict the future of such threads in Section V.

Algorithm 3 Build the SG

```
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: }
```

Note that a behavior can be instantiated multiple times. Instance isolation is needed so as to create different segments starting from the the same segment boundary for these instances separately.[4]

To present the algorithm for building a SG in detail, we need to introduce a few definitions.

1) *cacheMultiInfo:* A Boolean flag at each function showing the need to cache information for different instances; true if new segments are created or interface methods are called in this function; false otherwise.
   Note that, if cacheMultiInfo is false, the cached information is the same for all instances that call the function.
2) *cachedInfo:* Cached information, as follows.

[4]The effect of instance isolation is discussed in detail in [21].

Algorithm 4 Code Analysis for OoO 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
```

   a) *instID:* Instance identifier, e.g., `Main.b1`.
   b) *dummyInSeg:* A dummy segment as the initial segment of this cached information.
   c) *carryThrough:* A Boolean flag; true when the input segment carries through the function and is part of the output segments; false otherwise. For `Main.b1`, `B1.main.carrythrough` = false since seg_1 is connected to both seg_2 and seg_4, and therefore will not carry through `B1.main`.
   d) *outputSegments:* Segments (without the input segment) that will be the output after analyzing this function. For example for `Main.b1`, `B1.main.outputSegments` = $\{\text{seg}_4\}$.
   e) *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 `B1 while` loop [Fig. 4(a), line 7–11], the input segment list is $\{\text{seg}_1\}$ and the output segment list is $\{\text{seg}_3\}$. 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, i.e., `Main.main()`.

A. Algorithm for Static Conflict Analysis

The static code analysis algorithm consists of four phases.

1) *Input:* Design model (e.g., from file `design.sc`).
2) *Output:* SG and segment CTs.
3) *Phase 1:* Use Algorithm 3 to create the global SG, 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.
4) *Phase 2:* Use Algorithm 5 to build a local SG 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.

Algorithm 5 Code Analysis for OoO PDES, Phase 2

1: **for all** functions *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 localSegments and use the output segments as the input of the next statement in the current function.
9: **end for**
10: **end for**

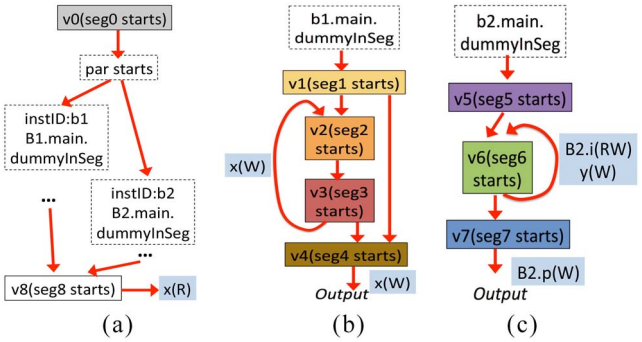


Fig. 6. Function-local SGs and variable access lists for the example in Fig. 4. (a) Main.main. (b) B.main (Main.b1). (c) B.main (Main.b2).

The complexity of this phase is $O(n_s)$ where n_s is the number of statements in the function definition.

Fig. 6(a) illustrates this for the example in Fig. 4. We do not follow function calls to `B1.main` from `Main.main` but connect a *dummyInSeg* node instead. We then create two sets of `main` function cached information for the instances of `B1` and `B2`, respectively, shown in Fig. 6(b) and (c), since segment boundary nodes are created when calling `B1.main` and `B2.main`.

Note that we do not know yet the instance path of the member variables in this phase. Therefore, we use port variable `p` instead of its real mapping `x` here.

5) *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 `B1.main.dummyInSeg` for instance `b1` when analyzing `Main.main` (if necessary).[5] 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 SG for each function.

6) *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, i.e., (`x` is the mapping of port `p`).

The complexity of this phase is $O(n_l)$ where n_l is the size of the local segment access list of `Main.main`.

[5]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.

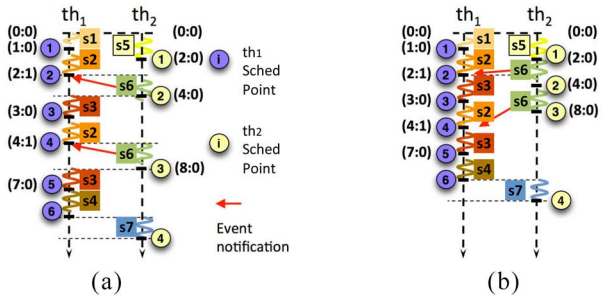


Fig. 7. OoO PDES scheduling. (a) Original scheduling. (b) Optimized scheduling.

In summary, the algorithm generates precise segment conflict information. 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).

V. Optimized OoO PDES With Predictions

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

A. State Prediction to Avoid False Conflicts

OoO scheduling is often prevented because of the unknown future behavior of the threads. Fig. 7(a) shows the scheduling of thread execution for the example in Fig. 4. 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.

An example of a false conflict detected at run time is shown in Fig. 7(a), when th_2 finishes its execution in seg_5 and hits the scheduling point th_2.(1) At that time, th_1 is running in seg_2. The current time is (1:0) for th_1 and (0:0) for th_2. As shown in Fig. 4(g), 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 2 will return true at line 16. The scheduler therefore cannot issue th_2 OoO at scheduling point th_2.(1).

However, this is a false conflict for OoO thread issuing. Although, th_1 may run into its 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)

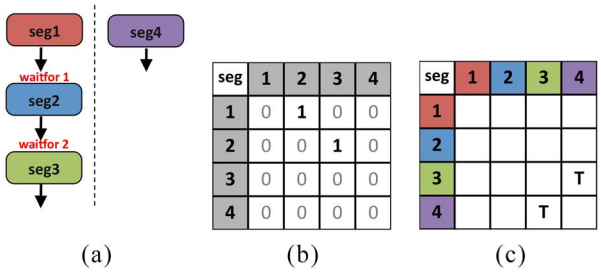


Fig. 8. Partial SG with adjacency matrix and CT. (a) SG. (b) Segment adjacency matrix. (c) Data CT.

after finishing seg_3. Thus, it is actually safe to issue th_2 OoO at scheduling point $th_2.①$ since th_2's time is not after (2:0).

We observe that, 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 in parallel with th_1 instead of holding it back for the next scheduling step.

This observation motivates the idea of optimizing OoO PDES by looking ahead of the current simulation state. With prediction information, as shown in Fig. 7(b), th_2 can be issued at both scheduling points $th_2.①$ and $th_2.②$. The simulator run time can thus be shortened by prediction.

B. Static Prediction Analysis

To provide the OoO scheduler with prediction information, we need the knowledge of future thread status. The OoO PDES scheduler can check the future status of threads if we provide future segment information. Consequently, we extend the data structures defined in Section III-B.

1) Data Hazards Prediction:

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

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

b) *Predictive Data CT (CT_n[N,N]):*

$$CT_n[i,j] = \begin{cases} true \text{ if } \text{seg}_i \text{ has a potential data conflict} \\ \qquad \text{with } \text{seg}_j \text{ within n scheduling steps} \\ false \text{ otherwise.} \end{cases}$$

Here, CT_0[N,N] is the same as data CT CT[N,N]. However, CT_n is asymmetric for $n > 0$.

Fig. 8(a) and (b) shows a partial SG and its adjacency matrix. The Data CT is shown in Fig. 8(c) where a data conflict exist between seg_3 and seg_4.

The Data CTs with 0, 1, and 2 prediction steps are shown in Fig. 9(a)–(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. 9(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. 9(c).

As shown in [20], the Data CT 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

$$CT_0[N,N] = CT[N,N] \quad (1)$$

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

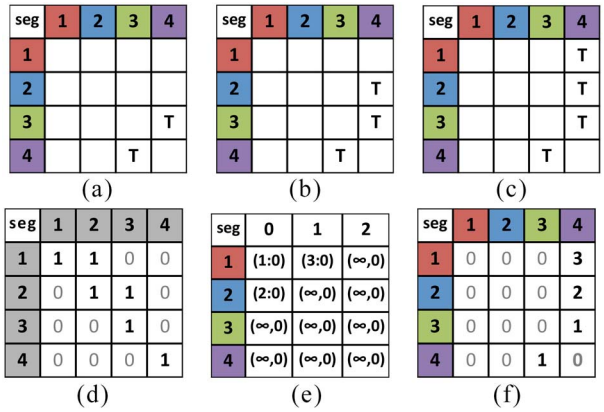


Fig. 9. Data structures for optimized OoO PDES scheduling. (a) Data CT w 0 prediction step (CT_0). (b) Data CT w 1 prediction step (CT_1). (c) Data CT w 2 prediction steps (CT_2). (d) Modified segment adjacency matrix. (e) Time advance table w 0, 1, 2 prediction steps. (f) Combined Data CT.

Here, A'[N,N] is the modified adjacency matrix [see Fig. 9(d)] with 1 s on the diagonal so as to preserve the conflicts from the previous data conflict prediction tables. Note that more conflicts are added to the CTs as the number of prediction steps increases, up to a maximum limit. As shown in [20], the number of prediction CTs for each design is limited. The maximum number of predictions is at most the length of the longest path in the SG.

We observe that CT_m contains all the conflicts from CT_0 to CT_{m-1} (m>0). Thus, we propose the following.

a) *Predictive Combined CT (CCT[N,N]):*

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

As shown in Fig. 9(f), the number of prediction steps is stored in CCT (instead of Boolean values).

2) Time Hazards Prediction:

a) *Predictive Time Advance Table* (NTime_n[N]):
$\text{NTime}_n[i] = \min\{\text{thread time advance after } n+1 \text{ scheduling steps from } \text{seg}_i\}$. Here, $\text{NTime}_0 = \text{NTime}$.

Fig. 9(e) shows the time advance table with predictions NTime_n for the example in Fig. 8. If a thread is now running in seg_1, it will advance by at least (3:0) after two scheduling steps. Therefore, $\text{NTime}_1[1] = (3{:}0)$.

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

a) *Predictive Event NT (NTP[N, N]):*

$$NTP[i,j] = \begin{cases} (t_\triangle, \delta_\triangle) \text{ if a thread in } \text{seg}_i \text{ may wake up} \\ \qquad \text{a thread in } \text{seg}_j \text{ with least} \\ \qquad \text{time advance of } (t_\triangle, \delta_\triangle) \\ (\infty, 0) \text{ if a thread in } \text{seg}_i \text{ will never} \\ \qquad \text{wake up another thread in } \text{seg}_j. \end{cases}$$

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. We predict the minimum time advance between each thread segment pair considering both direct or indirect event notifications.

Algorithm 6 Conflict Detection Using the Combined Prediction Table for M Prediction 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 ≥ 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 + NTime_m[th_2.seg]<th.timestamp) then
8:        return true;
9:    end if
10:   else if (M < M_FP)
11:     /*hazards may happen after M scheduling steps*/
12:     if(th_2.timestamp + NTime_M[th_2.seg]<th.timestamp) then
13:       return true; end if
14:   endif
15:   /*check event hazards*/
16:   for all th_w ∈ WAIT do
17:     if(NTP[th_2.seg, th_w.seg] + th_2.timestamp< th.timestamp) then
18:       /*th_w may wake up before th*/
19:       check data and time hazards between th_w and th; endif
20:   end for
21:   return false;
22: }
```

C. Conflict Detection for OoO PDES With Predictions

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

Algorithm 6 shows the conflict detection algorithm using predition for M steps ($M \geq 0$). Note that for $M = 0$, this matches the original OoO PDES conflict detection Algorithm 2.

Assume that th_1 and th_2 are two threads in the simulation of a model whose SG is Fig. 8(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 2 will return true because th_2 is possible to enter seg_2 with timestamp of (2:0) that is before th_1. Thus, the scheduler cannot issue th_1 OoO.

However, Conflict(th_1) in Algorithm 6 will return false when $M = 1$ or 2. With prediction information, the scheduler knows 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 time (3:0). Thus, it is actually safe to issue th_1 OoO at the current scheduling step. We conclude, the prediction information allows to eliminate false conflicts.

VI. Hybrid Multithreading Infrastructure

The OoO PDES scheduler aims at the most effective utilization of the available hardware on the multicore host. This requires an efficient multithreading infrastructure using user-level and kernel-level threads. Kernel-level threads maintained by the OS are needed to map parallel working threads onto the multiple cores of the simulation host. However, OS kernel threads carry a heavy overhead in system-calls for creation, deletion, and context switching. In contrast, user-level multithreading libraries, such as QuickThreads [22], incur very little overhead in cooperative multithreading, but can utilize only a single core.

In order to maximally exploit the parallelism of multicore machines and minimize the overhead for multiSthreading, we propose a HybridThreads [23] infrastructure which runs user-level threads (QuickThreads) on top of the kernel-level threads (PosixThreads). Based on HybridThreads, our OoO simulator benefits from both advantages, multiScore scalability and low system overhead.

Fig. 10 compares the simulator infrastructure using the three multithreading approaches. The traditional reference simulator uses the QuickThreads library [Fig. 10(a)] and runs multiple user-level threads on a single CPU core in cooperative fashion. Here, only one thread is actively running at any time. The same sequential scheduling can be implemented using kernel-level PosixThreads, as shown in Fig. 10(b).

In contrast, Fig. 10(c) shows a parallel simulator built on PosixThreads. Multiple threads can run on multiple CPU cores under synchronization by the PDES scheduler. The thread-to-core mapping and context switching is organized by the OS.

Fig. 10(d) shows our proposed HybridThreads approach which creates one PosixThread per available CPU core. Each thread is created at the beginning of the simulation and mapped to its core (fixed affiliation) until the end. On top of these kernel threads, user-level QuickThreads are created as needed by the application and scheduled on top of and across the kernel threads. Thus, these hybrid threads can run in parallel on the available CPU cores. If more than one user-level thread is mapped to a kernel thread, our HybridThreads library runs these in cooperative fashion, just as regular QuickThreads.

Internally, the HybridThreads library only updates a threads' stack with a new execution context during initialization. When a context switch occurs, the current register values and program counter are saved on its stack. The underlying kernel thread then switches to the stack of the next user-level thread. Thus, HybridThreads effectively runs multiple QuickThreads in parallel across the available CPU cores. The incurred overhead due to thread synchronization and resource sharing at both user and kernel levels is small and compensated by the performance improvement through truly parallel execution.

Note that HybridThreads essentially behave the same as QuickThreads when there is only one core available. On the other hand, on a multiScore host where a core is available for each thread, the HybridThreads library behaves practically just as PosixThreads. As such, HybridThreads make it possible to switch among different threading schemes dynamically.

VII. Experiments and Results

In this section, we present experimental results of using the proposed OoO simulator on real-world embedded applications and also evaluate the performance of the state prediction and the HybridThreads library. For our experiments, we use a SMP capable server running 64-bit Fedora 12 Linux. The SMP hardware specifically consists of 2 Intel Xeon X5650 processors running at 2.67 GHz[6] with six parallel cores each. Thus, in total the server hardware supports up to 12 threads running in parallel.

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

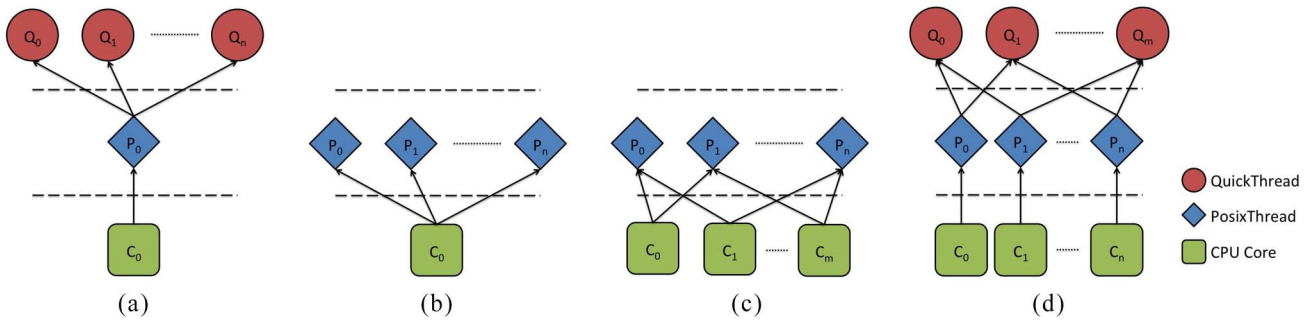


Fig. 10. User-level and kernel-level multithreading models in DE simulators. (a) Seq. DES using QuickThreads. (b) Seq. DES using PosixThreads. (c) PDES using PosixThreads. (d) PDES using HybridThreads.

TABLE III
EXPERIMENTAL RESULTS FOR THE ABSTRACT DVD PLAYER EXAMPLE (USING POSIXTHREADS)

Simulator:	Single-thread reference	Multi-core	
		Synch. PDES	OoO PDES
Compile time	0.7s	0.82s / 85.4%	1.28s / **54.7%**
Simulator time	34.75s	34.6s / 100.4%	15.92s / **217.3%**
# segments N		62	
total conflicts in $CT[N, N]$		168/3844 (4.4%)	
# threads issued		48930	
# threads issued out-of-order		**36263 (74.11%)**	

TABLE IV
STATISTICS FOR JPEG, H.264, FIBO, AND MANDELBROT EXAMPLES

JPEG Image Encoder				
TLM abstraction:	spec	arch	sched	net
# segments N	54	51	54	55
Total conflicts in $CT[N, N]$	208/2916 (7.1%)	208/2601 (8.0%)	208/2916 (7.1%)	219/3025 (7.2%)
# threads issued	274	273	274	4848
# threads issued out-of-order	**220 (80.3%)**	**217 (79.5%)**	**220 (80.3%)**	**4062 (83.79%)**
H.264 Video Decoder				
TLM abstraction:	spec	arch	sched	net
# segments N	72	69	72	74
Total conflicts in $CT[N, N]$	463/5184 (8.93%)	467/4761 (9.81%)	467/5184 (9.00%)	434/5476 (7.93%)
# threads issued	97176	97175	97177	97181
# threads issued out-of-order	**24806 (25.53%)**	**24810 (25.53%)**	**24842 (25.56%)**	**24847 (25.57%)**

Applications:	**Fibo_Timed**	**Mandelbrot**	**H.264 Encoder**	
# segments N	320	89	256	
Total conflicts in $CT[N, N]$	0/1024000 (0%)	77/7921 (0.97%)	2049/65536 (3.13%)	
# threads issued	127	7408	7463530	
# threads issued out-of-order	**49 (38.58%)**	**510 (6.88%)**	**5383237 (72.13%)**	

A. Evaluation of OoO PDES

We have implemented our parallel simulator in a SpecC[7]-based system design environment [24] and conducted experiments on six multimedia applications that we built in-house based on standard reference source code. All benchmark examples are tested to produce correct results.

We compare the compiler and simulator run times with the traditional single-threaded reference and a synchronous parallel implementation without OoO scheduling [5]. Tables III, VI, and VII show the speedup compared to the sequential simulator in the left column. Table V shows the cumulative speedup over different multithreading libraries.

1) Abstract Model of 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. According to profiling results, the workload ratio between decoding one H.264 frame with 704×566 pixels and one 44.1 kHz 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 CT is very sparse, allowing 74.11% of the threads to be issued OoO. While the synchronous PDES cannot gain performance due to in-order time barriers and synchronization overhead, our OoO simulator shows more than twice the simulation speed.

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

2) JPEG Encoder Model: Our second experiment uses a JPEG image encoder. The stimulus reads a BMP color image with 3216×2136 pixels and performs color-space conversion from RGB to YCbCr. Since, encoding of the three color components (Y, Cb, and 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 demonstrate models at different abstraction levels, we have created four models (specification, architecture mapped, scheduling refined, and network 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 about 80% of all threads can be issued OoO.

Tables V and VI show the corresponding compiler and simulator run times. While the compile time increases about 10%, the OoO simulation speed improves by more than 260% over the synchronous simulator. Moreover, for the JPEG encoder, our simulator is at least 2 times as fast as the reference implementation using QuickThreads, which clearly shows the benefit of parallelization.

3) Detailed H.264 Decoder Model: Our third experiment simulates a complex parallel video decoder based on the

TABLE V
EXPERIMENTAL RESULTS FOR SIMULATING SIX EMBEDDED APPLICATION EXAMPLES (WITH CUMULATIVE SPEEDUP)

Applications		Simulator								
		Single-thread			Multi-core					
		Sequential (Quick)	Sequential (Posix)		Synch. PDES (Posix)		OoO PDES (Posix)		OoO PDES (Hybrid)	
		sim[sec]	sim[sec]	speedup	sim[sec]	+speedup	sim[sec]	+speedup	sim[sec]	+speedup
DVD player		17.63	34.75	50.7%	34.60	100.4%	**15.92**	217.3%	**15.92**	100.0%
JPEG Encoder	spec	2.01	3.81	52.8%	2.90	131.4%	1.10	263.6%	**1.09**	100.9%
	arch	2.43	3.57	68.1%	2.85	125.3%	**1.09**	261.5%	**1.09**	100.0%
	sched	2.18	3.65	59.7%	2.86	127.6%	1.10	260.0%	**1.09**	100.9%
	net	2.82	4.72	59.7%	3.88	121.6%	1.32	293.9%	**1.29**	102.3%
H.264 Decoder	spec	**117.63**	246.85	47.7%	247.54	99.7%	142.87	173.3%	143.15	99.8%
	arch	**119.37**	249.36	47.9%	249.86	99.8%	142.20	175.7%	145.60	97.7%
	sched	**118.47**	251.48	47.1%	250.21	100.5%	141.30	177.1%	143.40	98.5%
	net	**119.29**	247.01	48.29%	251.01	98.41%	146.09	171.8%	147.72	98.9%
Fibo_Timed		14.72	21.08	69.8%	4.34	485.7%	2.19	198.2%	**1.83**	119.7%
Mandelbrot		345.37	664.94	51.9%	52.81	1259.1%	50.01	105.6%	**34.29**	145.8%
H.264 Encoder		**1348.55**	2764.25	48.8%	2752.80	100.4%	1428.40	192.7%	1378.30	103.6%

H.264/AVC standard [25]. While this model is at the highest level similar 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 intraprediction. For our simulation, we use a video stream of 1079 frames, 1280 × 720 pixels per frame, each with four slices of equal size. The experimental results for this industrial-size design are listed in Tables IV–VI, again for four models at different abstraction levels, including a network model with detailed bus transactions.

While the synchronous PDES shows almost no improvement in simulation speed, our proposed simulator shows more than 70% gain when using PosixThreads since about a quarter of the threads can be issued OoO (Table IV). Even with this large complex model, the increase of compilation time (Table VI) due to static conflict analysis is below 12%.

4) Parallel Timed Fibonacci Calculation: Our fourth application Fibo_Timed calculates the Fibonacci series in parallel and recursive fashion. Recall that a Fibonacci number is defined as the sum of the previous two numbers, $fib(n) = fib(n-1) + fib(n-2)$, and the first two numbers are $fib(0) = 0$ and $fib(1) = 1$. Our Fibonacci model parallelizes the Fibonacci calculation by letting two units compute the two previous numbers in parallel. This parallel decomposition continues up to a user-specified depth limit (in our case 5), from where on the classic recursive calculation method is used.

This application uses shared variables to communicate the input and calculated output values between the units, as well as a few counters to keep track of the actual number of parallel threads (for statistical purposes). Thus, the threads are not fully independent from each other. Also, the computational load is not evenly distributed among the instances due to the fact that the number of calculations increases by a factor of approximately 1.618 (the golden ratio) for every next number.

Our model also has timing information back-annotated using wait-for-time statements at each leaf block for the recursive calculation. The time delay of a unit is determined by its computational load, i.e., $T_{fib}(n) = 1.618 * T_{fib}(n-1)$.

For this highly parallel application, synchronous PDES shows almost 500% speedup and our OoO simulator doubles this when using PosixThreads. The use of HybridThreads gains another 20% speedup on top of this.

The short compilation time (Table VI) increases 8% for the synchronous PDES, and about 3× for OoO PDES. However, the absolute compile time of only 2 s is negligible.

5) Mandelbrot Graphics Renderer: As representative of a highly parallel and computation intensive graphics application, we built a renderer model for visualization of Mandelbrot images [26] with 512 parallel slices. Here, synchronous PDES shows more than 12 × speedup and our OoO simulator shows an extra 5% gain with PosixThreads. Notably, the use of HybridThreads pays off with another 45% speedup.

6) H.264/AVC Video Encoder With Parallel Motion Search: As a highly-complex multimedia application, we have parallelized a video encoder based on the H.264/AVC standard. Intra and interframe prediction are applied to encode an image according to the type of the current frame. During interframe 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 this paper, 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 176 × 144 pixels per frame. There are five bi-directionally predicted slices (B-slices) between the intracoded slices (I-slices) or forward predicted slices (P-slices). Thus, among every five consecutive frames, four frames need interframe prediction.

Table V shows that when using the same PosixThread library, the synchronous PDES can hardly achieve any speedup while the OoO PDES accelerates the simulation by almost 2×. While the HybridThread approach adds another 3.6% speedup, none of the parallel approaches can beat the sequential QuickThreads simulator. For this application, the available parallelism is too limited (a large part of computation is sequential) to make up for the parallel simulation overhead.

B. Evaluation of the Hybrid Multithreading Approach

Overall, Table V shows quite different performance for the multithreading libraries Quick-, Posix-, and HybridThreads. A significant drop down to about 50% occurs from the user-level QuickThreads to kernel-level PosixThreads. As indicated by the cumulative speedup, this high cost of utilizing multiple

TABLE VI
EXPERIMENTAL RESULTS FOR COMPILING SIX APPLICATION EXAMPLES

Applications		Simulator				
		Sequential	Synch. PDES		OoO PDES	
		comp[sec]	comp[sec]	speedup	comp[sec]	speedup
DVD player		0.70	0.82	85.4%	1.28	54.7%
JPEG Encoder	spec	3.43	3.16	108.5%	3.56	96.3%
	arch	3.59	3.57	100.6%	3.86	93.0%
	sched	3.54	3.52	100.6%	3.90	90.8%
	net	3.87	3.97	97.5%	4.54	85.2%
H.264 Decoder	spec	6.60	6.78	97.4%	7.46	88.5%
	arch	7.00	7.14	98.0%	7.72	90.7%
	sched	6.95	7.08	98.2%	7.75	89.7%
	net	7.22	7.29	99.0%	8.17	88.4%
Fibo_Timed		0.62	0.67	92.5%	2.00	31.0%
Mandelbrot		1.89	1.85	102.2%	7.86	23.5%
H.264 Encoder		21.18	21.12	100.3%	36.50	58.0%

TABLE VII
COMPARISON OF MULTITHREADING APPROACHES USING MANDELBROT WITH INCREASING NUMBER OF PARALLEL SLICES

Parallel Slices	Simulator				
	Seq. (Quick)	OoO PDES (Posix)		OoO PDES (Hybrid)	
	sim[sec]	sim[sec]	speedup	sim[sec]	+speedup
1	**345.15**	352.34	98.0%	357.82	98.5%
2	345.12	213.35	161.8%	**213.27**	100.0%
4	345.21	125.80	274.4%	**121.95**	103.2%
8	345.11	71.55	482.3%	**68.82**	104.0%
16	345.11	52.22	660.9%	**48.38**	107.9%
32	345.15	46.30	745.5%	**40.88**	113.3%
64	345.30	46.17	747.9%	**36.86**	125.3%
128	345.17	47.58	725.5%	**35.74**	133.1%
256	345.40	48.37	714.1%	**34.78**	139.1%
512	345.37	50.01	690.6%	**34.29**	145.8%

CPU cores is generally recovered by the parallel simulators and the HybridThreads approach. While for examples with limited application parallelism, i.e., the H.264 codecs, QuickThreads remains the fastest simulator, the HybridThreads approach is the winner for the other four applications, as indicated by bold numbers (fastest simulations) in Table V.

The righmost column compares HybridThreads directly against PosixThreads for OoO PDES. Both perform equally well in most cases, but HybridThreads can exploit its quick context switch advantage for the highly parallel applications Fibo_Timed and Mandelbrot, adding 20% to 45% speedup.

Table VII shows the difference between the multithreading approaches using the Mandelbrot example with an increasing number of parallel slices. While QuickThreads performs the same for all cases, PosixThreads gains speed with increased parallelism until there are significantly more threads than parallel cores available, when performance decreases again. HybridThreads, in contrast, shows steady performance gains over PosixThreads and 10 times faster than QuickThreads.

C. Evaluation of State Prediction

The effect of scheduler state prediction (Section V) is demonstrated in Table VIII which shows the simulation speed and compile times of the H.264 decoder models for increasing number of prediction steps. Overall, more prediction steps achieve higher simulator speed. On a closer look, one can also observe significant speedup jumps after 4 and 8 prediction steps for the spec model, and after 3 and 7 steps for the refined models (which exhibit a reorganized hierarchy). For all these models, simulation speed converges at 8 to 9 prediction steps.

Note that even for such large design models, the increase in compile time due to the static prediction analysis is negligible. The maximum number of predictions, i.e., the length of the longest path in the SG [20], can guarantee the most efficient OoO simulation at almost no cost.

TABLE VIII
EFFECT OF STATE PREDICTION ON THE H.264 DECODER

Prediction Steps	Simulator run time [sec]				Compile time [sec]			
	spec	arch	sched	net	spec	arch	sched	net
1	240.04	245.47	248.41	246.57	6.35	6.23	7.75	6.67
2	241.46	245.90	249.88	248.33	6.02	6.05	7.94	6.58
3	240.19	251.00	247.68	248.05	7.49	6.06	6.19	8.15
4	247.79	181.80	182.04	180.76	7.40	6.11	7.85	6.71
5	178.93	179.94	179.08	178.55	7.32	7.59	7.93	6.63
6	178.31	179.60	182.00	180.09	7.38	7.56	7.92	8.45
7	182.50	179.70	180.14	179.11	6.30	7.80	7.92	8.25
8	171.02	153.46	143.24	148.01	6.19	7.67	6.41	6.61
9	142.02	153.62	146.76	148.12	6.12	7.65	7.84	6.63
10	144.93	153.44	147.58	149.55	7.50	6.26	7.71	6.59
11	144.12	151.08	147.33	148.02	6.27	7.60	6.52	6.58
12	145.02	153.06	146.46	148.88	7.43	6.32	6.22	8.09

VIII. CONCLUSION

Highest simulator performance is critical for the efficient validation of transaction-level design models. In this paper, we have presented a novel OoO scheduling technique for multicore parallel simulation of system-level models. 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 a SG of the application. Using fast CT lookups, our OoO scheduler can quickly make decisions at run-time and issue more parallel threads than synchronous PDES. At the same time, we make use of a hybrid multithreading infrastructure that combines the benefits of multicore kernel-level threads and low-overhead user-level threads for truly parallel execution with fast context-switching.

Our experimental results show that, with only a small increase in compile time, our simulator is significantly faster than the traditional sequential reference implementation, as well as a synchronous PDES implementation. Notably, our OoO 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 (i.e., add pointer analysis) and look into additional methods to further improve the simulation speed (e.g., many-core target platforms).

ACKNOWLEDGMENT

The authors would like to thank the National Science Foundation (NSF) for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Boston, MA, USA: Kluwer, 2002.

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

[3] R. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, pp. 30–53, Oct. 1990.

[4] P. Ezudheen, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC kernel for fast hardware simulation on SMP machines," in *Proc. ACM/IEEE/SCS 23rd Workshop Princ. Adv. Distrib. Simulat. (PADS)*, Lake Placid, NY, USA, 2009, pp. 80–87.

[5] W. Chen, X. Han, and R. Dömer, "Multi-core simulation of transaction level models using the system-on-chip environment," *IEEE Des. Test. Comput.*, vol. 28, no. 3, pp. 20–31, May/Jun. 2011.

[6] K. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 5, pp. 440–452, Sep. 1979.

[7] D. Nicol and P. Heidelberger, "Parallel execution for serial simulators," *ACM Trans. Model. Comput. Simulat.*, vol. 6, pp. 210–242, Jul. 1996.

[8] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous parallel SystemC simulation on multi-core host architectures," in *Proc. Int. Conf. Hardware/Softw. Codesign Syst. Syn.*, Scottsdale, AZ, USA, 2010, pp. 241–246.

[9] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele, "Scalably distributed SystemC simulation for embedded applications," in *Proc. Int. Symp. Ind. Embedded Syst. (SIES)*, Le Grande Motte, France, Jun. 2008, pp. 271–274.

[10] L. Cai and D. Gajski, "Transaction level modeling: An overview," in *Proc. Int. Conf. Hardware/Softw. Codesign Syst. Syn.*, Newport Beach, CA, USA, Oct. 2003, pp. 19–24.

[11] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate source-level simulation of software timing considering complex code optimizations," in *Proc. Design Autom. Conf. (DAC)*, New York, NY, USA, 2011, pp. 486–491.

[12] A. Gerstlauer, "Host-compiled simulation of multi-core platforms," in *Proc. Int. Symp. Rapid Syst. Prototyping (RSP)*, Washington, DC, USA, Jun. 2010, pp. 1–6.

[13] *Systemc TLM-2.0*, IEEE Standard 1666-2011, 2011.

[14] S. Sirowy, C. Huang, and F. Vahid, "Online SystemC emulation acceleration," in *Proc. Design Autom. Conf. (DAC)*, Anaheim, CA, USA, 2010, pp. 30–35.

[15] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: A fast SystemC simulator on GPUs," in *Proc. Asia South Pacific Design Autom. Conf. (ASPDAC)*, Taipei, Taiwan, 2010, pp. 149–154.

[16] R. Sinha, A. Prakash, and H. D. Patel, "Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs," in *Proc. Asia South Pacific Design Autom. Conf. (ASPDAC)*, Sydney, NSW, Australia, 2012, pp. 455–460.

[17] S. Mukherjee *et al*., "Wisconsin wind tunnel II: A fast, portable parallel architecture simulator," *IEEE Concurrency*, vol. 8, no. 4, 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 1, pp. 121–131, Jan. 2012.

[19] W. Chen, X. Han, and R. Dömer, "Out-of-order parallel simulation for ESL design," in *Proc. Design Autom. Test Europe (DATE) Conf.*, Dresden, Germany, 2012, pp. 141–146.

[20] W. Chen and R. Dömer, "Optimized out-of-order parallel discrete event simulation using predictions," in *Proc. Design Autom. Test Europe (DATE) Conf.*, Grenoble, France, 2013, pp. 3–8.

[21] W. Chen and R. Dömer, "An optimizing compiler for out-of-order parallel ESL simulation exploiting instance isolation," in *Proc. Asia South Pacific Design Autom. Conf. (ASPDAC)*, Sydney, NSW, Australia, 2012, pp. 461–466.

[22] D. Keppel, "Tools and techniques for building fast portable threads packages," Dept. Comput. Sci. Eng., Univ. Washington, Seattle, WA, USA, Tech. Rep. UWCSE 93-05-06, May 1993.

[23] G. Liu and R. Dömer, "A hybrid thread library built for efficient electronic system level simulation," Center Embedded Comput. Syst., Univ. California, Irvine, CA, USA, Tech. Rep. CECS-13-11, Oct. 2013.

[24] R. Dömer *et al*., "System-on-chip environment: A SpecC-based framework for heterogeneous MPSoC design," *EURASIP J. Embedded Syst.*, vol. 2008, Jan. 2008, Art. ID 647953.

[25] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 560–576, Jul. 2003.

[26] B. Mandelbrot, "Fractal aspects of the iteration of $z- > \lambda z(1-z)$ for complex λ and z," in *Non-Linear Dynamics (New York)*, vol. 357, Robert H. G. Helleman, Ed. New York, NY, USA: Ann. NY. Acad. Sci., 1979, pp. 249–259.

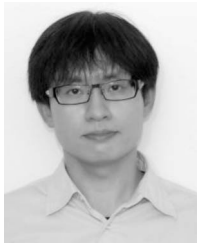
Weiwei Chen (S'08–M'14) received the B.S. and M.S. degrees in computer science and engineering from Shanghai Jiao Tong University, Shanghai, China, and the Ph.D. degree in electrical engineering and computer science from the University of California, Irvine, Irvine, CA, USA, in 2013.

She is currently a Researcher with Qualcomm Research Silicon Valley, Santa Clara, CA, USA. Her current research interests include system-level design and validation and parallel computing.

Dr. Chen was the recipient of the 2013 Outstanding Dissertation Award by European Design Automation Association.

Xu Han receiced the M.S. degree in electrical engineering from the Royal Institute of Technology, Stockholm, Sweden, and the Ph.D. degree in electrical engineering and computer science from the University of California, Irvine, Irvine, CA, USA, in 2013.

He is currently a Senior Engineer with Qualcomm Innovation Center, San Diego, CA, USA. His current research interests include system-level modeling and recoding of embedded system models.



Che-Wei Chang received the M.S. degrees in electrical engineering and computer science and engineering from Cheng Kung University, Tainan City, Taiwan, and the University of California, Irvine, Irvine, CA, USA, in 2011. He is currently pursuing the Ph.D. degree in electrical engineering and computer science with the University of California, Irvine.

His current research interests include system-level modeling and formal verification.

Guantao Liu received the M.S. degree in computer science and engineering from the University of California, Irvine, Irvine, CA, USA, in 2013, where he is currently pursuing the Ph.D. degree in electrical engineering and computer science.

His current research interests include system-level design, parallel discrete event simulation, and many-core computer architectures.

Rainer Dömer (S'96–M'00) received the Ph.D. degree in information and computer science from the University of Dortmund, Dortmund, Germany, in 2000.

He is currently an Associate Professor with the Electrical Engineering and Computer Science Department, the University of California, Irvine, Irvine, CA, USA, where he is a faculty member of the Center for Embedded Computer Systems. His current research interests include system-level design and methodologies, embedded computer systems, specification and modeling languages, and parallel discrete event simulation.