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 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 transaction-level 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.

Using six embedded application examples, we also show the speed trade-off for multi-core PDES based on different multi-threading libraries.

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 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 simulation speed.

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 runtime 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 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 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 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.

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.

Distribution parallel simulation, such as [9] and [6], 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 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. 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 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 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 speeds up 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 our work in [19], [20] and [21], 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 and out-of-order hazards due to the localized timing (Section III-B). We also integrate a novel prediction technique (Section V) of-order hazards due to the localized timing (Section III-B).

We introduce the following notations:

1. **PDES** Simulation
2. **T**ime
3. **E**vents
4. **Q**uues
5. **T**hreads

### 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 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 multi-core utilization.

#### A. Notations

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

1. We define simulation time as tuple \((t, \delta)\) where \(t = \text{time}, \delta = \text{delta-cycle}. We order time stamps as follows:
   - **equal:** \((t_1, \delta_1) = (t_2, \delta_2), \text{iff } t_1 = t_2, \delta_1 = \delta_2\)
   - **before:** \((t_1, \delta_1) < (t_2, \delta_2), \text{iff } t_1 < t_2, \text{ or } t_1 = t_2, \delta_1 < \delta_2\)
   - **after:** \((t_1, \delta_1) > (t_2, \delta_2), \text{iff } t_1 > t_2, \text{ or } t_1 = t_2, \delta_1 > \delta_2\)
2. Each thread has its own time \((t_{th}, \delta_{th})\).
3. Since events can be notified multiple times and at different simulation times, we note an event \(e\) notified at \((t, \delta)\) as tuple \((id_e, t_e, \delta_e)\) and define: \(EVENTS = \{\text{EVENTS}_{t, \delta}\) where \(\text{EVENTS}_{t, \delta} = \{\{id_e, t_e, \delta_e\} | t_e = t, \delta_e = \delta\}\)
4. For DE-based simulation, typically several sets of queued threads are defined, such as \(\text{QUEUES} = \{\text{READY, RUN, WAIT, WAITFOR}\). 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** = \{READY\}<sub>\(t, \delta\), READY\(t, \delta\) = \{\text{th } \mid \text{ th is ready to run at } (t, \delta)\}\)
- **RUN** = \{RUN\}<sub>\(t, \delta\), RUN\(t, \delta\) = \{\text{th } \mid \text{ th is running at } (t, \delta)\}\)
- **WAIT** = \{WAIT\}<sub>\(t, \delta\), WAIT\(t, \delta\) = \{\text{th } \mid \text{ th is waiting since } (t, \delta)\}\)
- **WAITFOR** = \{WAIT\_<sub>FOR\><sub>\(t, \delta\), WAIT\_<sub>FOR\><sub>\(t, \delta\) = \{\text{th } \mid \text{ th is waiting for simulation time advance to } (t, 0)\}\)
- **JOINING** = \{JOINING\_<sub>\(t, \delta\), JOINING\_<sub>\(t, \delta\) = \{\text{th } \mid \text{ th created child threads at } (t, \delta)\}\)
- **COMPLETE** = \{COMPLETE\_<sub>\(t, \delta\), COMPLETE\_<sub>\(t, \delta\) = \{\text{th } \mid \text{ 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:
   - \(t = 0, \delta = 0\)
   - **RUN** = RUN\(_{0,0}\) = \{\text{th_{root}}\}
   - **READY** = READY\(_{0,0}\) = \{\text{WAIT} = WAIT\(_{0,0}\) = WAITFOR = WAITFOR\(_{0,0}\) = COMPLETE = COMPLETE\(_{0,0}\) = JOINING = JOINING\(_{0,0}\) = 0\}

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\_<sub>\(t, \delta\) → RUN\_<sub>\(t, \delta\):** the thread becomes runnable (is issued)
- **RUN\_<sub>\(t, \delta\) → WAIT\_<sub>\(t, \delta\):** the thread calls **wait** for an event
- **RUN\_<sub>\(t, \delta\) → WAIT\_<sub>FOR\><sub>\(t, \delta\), where } t < t' = t + delay:** the thread calls **wait**time\(_{delay}\)
- **RUN\_<sub>\(t, \delta\) → JOINING\_<sub>\(t, \delta\):** the thread creates its child threads and waits for them to complete
- **RUN\_<sub>\(t, \delta\) → COMPLETE\_<sub>\(t, \delta\):** the thread finishes the work and completes
- **WAIT\_<sub>\(t, \delta\) → READY\_<sub>\(t', \delta':** where } (t, \delta) < (t', \delta'):** the event that the thread is waiting for is notified; the thread becomes ready to run at \((t', \delta')\)
- **JOINING\_<sub>\(t, \delta\) → READY\_<sub>\(t', \delta':** where } (t, \delta) ≤ (t', \delta'):** child threads completed and their parent becomes ready to run again
- **WAIT\_<sub>FOR\><sub>\(t, \delta\) → READY\_<sub>\(t', \delta':** where } (t, \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\_<sub>\(t, \delta\) and RUN\_<sub>\(t, \delta\) are empty and there are no WAIT or WAITFOR queues with earlier timestamps, the scheduler deletes READY\_<sub>\(t, \delta\) and RUN\_<sub>\(t, \delta\), as well as any expired events with the same timestamp EVENTS\_<sub>\(t, \delta\) (lines 8-14 in Algorithm 1).

#### 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 \((t_e, \delta_e + 1)\) where \((t_e, \delta_e)\) 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
Algorithm 1 Out-of-order PDES Algorithm

1: /* trigger events */
2: for all th ∈ WAIT do
3:   if ∃ event (ide, te, δte), th awaits e, and (te, δte) ≥ (tth, δth) then
4:     move th from WAIT tth,δth to READY tth,δth + 1
5:     tth = te; δth = δte + 1
6:   end if
7: end for
8: /* update simulation subsets */
9: for all READYt,δ and RUNt,δ do
10:   if READYt,δ = ∅ and RUNt,δ = ∅ and WAITFORt,δ = ∅ then
11:     delete READYt,δ, RUNt,δ, WAITFORt,δ, EVENTSt,δ
12:     merge WAITt,δ into WAITnext(t,δ); delete WAITt,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 WAITFORt,δth to READY t,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

qualified (line 28). Finally, if no threads can run (RUN = ∅), the simulator reports a deadlock and quits.\footnote{The condition for a deadlock is the same as for a regular DE simulator.}

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.

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 $V = \{v | v$ is a segment boundary\}, $E = \{e_{ij} | e_{ij}$ 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, $b1$ 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 control flow graph in Fig. 4(b), we derive the segment graph 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 $v1 \rightarrow v2$, or skips the while loop (line 7) to reach waitfor 3 (line 12) via $v1 \rightarrow v4$. While in the loop (line 7), the execution reaches wait e (line 9) via $v2 \rightarrow v3$ after waitfor 1 (line 8), and then waitfor 3 (line 12) via $v3 \rightarrow v4$ when it exits the loop.

Similarly, instance $b2$ reaches waitfor 2 (line 8) via $v5 \rightarrow v6$ and waitfor 4 (line 12) via $v6 \rightarrow v7$ after the loop. Its execution completes at the end of the par statement via $v7 \rightarrow v8$. 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 out-of-order if their segments 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 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 $i$ 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 \( CT[N, N] \) 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 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 \( CT[N, N] \) 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 pointer access by \( seg_i \) or \( seg_j \), 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, \( seg_3 \) 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:

- **Global variables**, e.g. \( x, y \) in line 2: This case is discussed above and can be handled directly as tuple \((Symbol, AccessType)\).
- **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.
- **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 \( Main.b2.i \).
- **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 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 a notification that has been issued by another thread. We define an event notification table \( NT[N, N] \) where \( N \) is the total number of segments: \( NT[i, j] = true \), iff segment

---

**Fig. 5.** Write-after-write (WAW) conflict between two parallel threads.
segₙ notifies an event that segₙ is waiting for; otherwise, NT[i,j] = false. Note that in contrast to the data conflict table, the event notification table is not symmetric.

Fig. 4(e) shows the event notification table for the simple example. For instance, NT[6,3] = true since seg₉ notifies the event e (line 10) which seg₃ 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.

TABLE I
TIME ADVANCES AT SEGMENT BOUNDARIES.

<table>
<thead>
<tr>
<th>Segment boundary</th>
<th>Time Increment</th>
<th>Add to (t, δ')</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait event</td>
<td>increment by one delta cycle (0:1)</td>
<td>(t', δ' + 1)</td>
</tr>
<tr>
<td>waitfor t</td>
<td>increment by time t (t:0)</td>
<td>(t', t + t, 0)</td>
</tr>
<tr>
<td>par</td>
<td>par_end</td>
<td>no time increment (0:0)</td>
</tr>
</tbody>
</table>

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 seg₆ sets CTime[6] = (2 : 0).

The next time advance table NT ime[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:

\[ NT ime[i] = \min\{CTime[j] \mid \forall segₗ which follow segᵢ \} \]

For example, Fig. 4(g) lists NT ime[1] = (1:0) since seg₁ is followed by seg₂ (increment (1:0)) and seg₄ (increment (3:0)).

TABLE II
EXAMPLES FOR DIRECT AND INDIRECT TIMING HAZARDS.

<table>
<thead>
<tr>
<th>Situation</th>
<th>th₁</th>
<th>th₂</th>
<th>Hazard?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>(10:2)</td>
<td>(10:0), next segment at (10:1)</td>
<td>yes</td>
</tr>
<tr>
<td>Timing Hazard</td>
<td>(10:2)</td>
<td>(10:0), next segment at (12:0)</td>
<td>no</td>
</tr>
<tr>
<td>Indirect</td>
<td>(10:2)</td>
<td>(10:0), wakes th₃ at (10:1)</td>
<td>yes</td>
</tr>
<tr>
<td>Timing Hazard</td>
<td>(10:2)</td>
<td>(10:1), wakes th₃ at (10:2)</td>
<td>no</td>
</tr>
</tbody>
</table>

There are two types of timing hazards, namely direct and indirect ones. For a candidate thread th₁ to be issued, a direct timing hazard exists when another thread th₂, that is safe to run, resumes its execution at a time earlier than the local time of th₁. In this case, the future of th₂ is unknown and could potentially affect th₁. Thus, it is not safe to issue th₁.

Table II shows an example where th₁ is considered for execution at time (10:2). If there is a thread th₂ with local time (10:0) whose next segment runs at time (10:1), i.e. before th₁, then the execution of th₁ is not safe. However, if we know from the time advance tables that th₂ will resume its execution later at (12:0), no timing hazard exists with respect to th₂.

An indirect timing hazard exists if a third thread th₃ can wake up earlier than th₁ due to an event notified by th₂. Again, Table II shows an example. If th₂ at time (10:0) potentially wakes a thread th₃ so that th₃ runs in the next delta cycle (10:1), i.e. earlier than th₁, then it is not safe to issue th₁.

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, th₂) 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.

Algorithm 2 Conflict Detection in Scheduler

1: bool NoConflicts(Thread th) 2: { 3: for all th₂ ∈ RUN ∪ READY, 4: if ((th₂,t,th₂,δ) ∈ THREATEN) do 5: if (Conflict(th, th₂)) then return false end if 6: end for 7: return true 8: } 9: }

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 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.

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.

To present the algorithm for building a segment graph in detail, we need to introduce a few definitions:

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

4The effect of Instance Isolation is discussed in detail in [21]
Algorithm 3 Build the Segment Graph

1: newSegList = BuildSegmentGraph(currSegList, stmt)
2: {  
3:     case STMNT_COMPOUND:
4:         newL = currSegList
5:         for all subStmnt ∈ subStmnt do
6:             newL = BuildSegmentGraph(newL, subStmnt)
7:         end for
8:         case STMNT_IF_ELSE:
9:             tmpL = new SegmentList; tmpL.add(tmp1)
10:        tmp2 = BuildSegmentGraph(currSegList, subElseStmnt)
11:        tmp1 = BuildSegmentGraph(currSegList, subIfStmnt)
12:        newL = currSegList
13:        ExtendAccess(stmnt.conditionVar, currSegList)
14:        newL = tmp1 ⋃ tmp2
15:        ExtendAccess(stmnt.expression, currSegList)
16:        case STMNT_WHILE:
17:            tmpL = new SegmentList; tmpL.add(helperSeg)
18:        newL = BuildSegmentGraph(currSegList, fct.topstmnt);
19:        BuildSegmentGraph(tmpL, subWhileStmnt)
20:        if helperSeg ∈ tmp1
21:            then remove helperSeg from tmp1
22:        for all Segment s ∈ tmp1 ⋃ currSegList do
23:            s.nextSegments = helperSeg.nextSegments
24:        end for
25:        case STMNT_FAR:
26:            newSeg = new Segment; totalSegments ++
27:            newEndSeg = new Segment; totalSegments ++
28:        for all Segment s ∈ currSegList do
29:            s.nextSegments.add(newSeg)
30:        end for
31:        case STMNT_EXPR:
32:            newL = BuildSegmentGraph(tmpL, subStmt)
33:        for all Segment s ∈ tmpL do
34:            s.nextSegments.add(newEndSeg)
35:        end for
36:        case STMNT_WAIT:
37:            newL = new SegmentList; newL.add(newEndSeg)
38:        case STMNT_WAITFOR:
39:            newSeg = new Segment; totalSegments ++
40:        for all Segment s ∈ currSegList do
41:            s.nextSegments.add(newSeg)
42:        end for
43:        case STMNT_EXPRESSION:
44:            if stmt 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: }

- **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.

- **cachedInfo**: cached information, as follows:
  - **instID**: instance identifier, e.g. `Main.b1`
  - **dummyInSeg**: a dummy segment as the initial segment of this 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 `Main.b1`, `B1.main.carrythrough = false` since `seg1` is connected to both `seg2` and `seg4`, and therefore will not carry through `B1.main`.

- **outputSegments**: segments (without the input segment) that will be the output after analyzing this function. For example for `Main.b1`, `B1.main.outputSegments={seg1}`.
- **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 `{seg1}` and the output segment list is `{seg3}`. To start, we create an initial segment (i.e. `seg0`) 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.

- **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.

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

- **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. 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
Algorithm 5 Code analysis for out-of-order 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 segment graphs and variable access lists for the example in Fig. 4.

sets of main function cached information for the instances of B1 and B2 respectively, shown in Fig. 6(b) and Fig. 6(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.

• 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). 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, 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.

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 out-of-order 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

Out-of-order 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 $t_1$ and $t_2$ 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 $t_2$ finishes its execution in $seg_5$ and hits the scheduling point $th_2.1$. At that time, $t_1$ is running in $seg_2$. The current time is (1:0) for $t_1$ and (0:0) for $t_2$. 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 $t_1$ to enter the next segment, i.e. $seg_3$, is (1:1), and for $t_2$ is (2:0). Since $t_1$ may run into its next segment ($seg_3$) with an earlier timestamp (1:1) than $t_2$ (2:0), the Conflict() in Algorithm 2 will return true at line 16. The scheduler therefore cannot issue $t_2$ out-of-order at scheduling point $th_2.1$.

However, this is a false conflict for out-of-order thread issuing. Although $t_1$ may run into its next segment ($seg_3$) earlier than $t_2$, there are no data conflicts between $t_1$’s next segment $seg_3$ and $t_2$’s current segment $seg_5$. Moreover, the next time advance of $seg_3$ is (1:0). So $t_1$ will start a new segment no earlier than (2:0) after finishing $seg_3$. Thus, it is actually safe to issue $t_2$ out-of-order at scheduling point $th_2.1$ since $t_2$’s time is not after (2:0).

We observe that, if the scheduler knows what will happen with $t_1$ in more than one scheduling step ahead of scheduling point $th_2.1$, it can issue $t_2$ in parallel with $t_1$ instead of holding it back for the next scheduling step.

This observation motivates the idea of optimizing OoPDES by looking ahead of the current simulation state. With prediction information, as shown in Fig. 7(b), $t_2$ can be issued at both scheduling points $th_2.1$ and $th_2.2$. The simulator run time can thus be shortened by prediction.
is limited. The maximum number of predictions is at most the number of prediction conflict tables for each design. More conflicts are added to the conflict tables as the number of steps increases. Note that we extend the data structures defined in Section III-B:

1) Data hazards prediction:
- Segment Adjacency Matrix \( A[N,N] \):

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

- Predictive Data Conflict Table \( CT_n[N,N] \):

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

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

Fig. 8(a) and (b) show a partial segment graph and its Adjacency Matrix. The Data Conflict Table is shown in Fig. 8(c) where a data conflict exist between seg_3 and seg_4.

The Data Conflict Tables with 0, 1 and 2 prediction steps are shown in Fig. 9(a), (b) and (c), respectively. Since seg_2 is followed by seg_3 and seg_3 has a conflict with seg_4, a thread in seg_2 has a conflict with a thread who is in seg_4 after one scheduling step. Thus, \( CT_1[2,4] \) is true in Fig. 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 Conflict Table with \( n \) prediction steps can be built recursively by using Boolean matrix multiplication. Basically, if seg_i is followed by seg_j, and seg_j has a data conflict with seg_k within the next \( n-1 \) prediction steps, then seg_i has a data conflict with seg_k within the next \( n \) prediction steps. Formally,

\[
CT_0[N,N] = CT[N,N]
\]

\[
CT_n[N,N] = A'[N,N] \ast CT_{n-1}[N,N], \quad \text{where} \ n > 0
\]

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

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

- Predictive Combined Conflict Table (CCT\([N,N]\)):

\[
CCT[i,j] = \begin{cases} 
\min \{k \mid CT_k[i,j] = \text{true}\} & \text{if seg}_i \text{ has a conflict with seg}_j; \\
0 & \text{otherwise.}
\end{cases}
\]

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

\[
\begin{array}{c|cccc|c|cccc|c}
\hline
\text{seg}_{1} & \text{seg}_{2} & \text{seg}_{3} & \text{seg}_{4} \\
\hline
1 & \times & \times & & \\
2 & \times & \times & & \\
3 & \times & \times & & \\
4 & \times & \times & & \\
\hline
\end{array}
\]

2) Time Hazards Prediction:
- Predictive Time Advance Table (\( NTime_n[N] \)):

\[
NTime_n[i] = \min \{\text{thread time advance after n + 1 scheduling steps from seg}_i\}.
\]

Here, \( NTime_0 = NTime \).

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, \( NTime_1 = (3:0) \).

3) Event Hazards Prediction: We need also prediction information for event notifications to prevent event hazards.
- Predictive Event Notification Table (\( NTP[N,N] \)):

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

Here, we have table entries of time advances. Note that a thread can wake up another thread directly or indirectly via other threads. For instance, \( th_1 \) wakes up \( th_2 \), and \( th_2 \) then wakes up \( th_3 \) through event delivery. In this case, \( th_1 \) wakes up \( th_2 \) directly and \( th_3 \) indirectly. We predict the minimum time advance between each thread segment pair considering both direct or indirect event notifications.
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 out-of-order.

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

\begin{algorithm}
1: \textbf{bool} Conflict(Thread \( th_1 \), Thread \( th_2 \))
2: \{ 
3: \text{//check the combined prediction table for data and time hazards\} 
4: \text{m = }CT[\text{th}_2\text{-seg}, \text{th}_\text{seg}]\cdot 1; 
5: \text{if}(m \geq b) \text{then} /*There are data conflicts within M scheduling steps*/ 
6: \text{if}(\text{th}_\text{seg} \text{may enter into a segment before th and cause data hazards*/} 
7: \text{if}(\text{th}_2\text{-timestamp} + \text{NT ime}(\text{th}_2\text{-seg}) < \text{th}.\text{timestamp}) \text{then} 
8: \text{return true; } 
9: \text{end if} 
10: \text{else if } (M < M_P)/*hazards may happen after M scheduling steps*/ 
11: \text{if}(\text{th}_2\text{-timestamp} + NT ime(\text{th}_2\text{-seg}) < \text{th}.\text{timestamp}) \text{then} 
12: \text{return true; end if} 
13: \text{endif} 
14: \text{for all th}_w \in \text{WAIT do} 
15: \text{if}(\text{th}_w\text{-timestamp} + \text{NT ime}(\text{th}_w\text{-seg}) < \text{th}.\text{timestamp}) \text{then} 
16: \text{return true; end if} 
18: \text{endif} 
20: \text{return false; } 
22: \}
\end{algorithm}

Assume that \( th_1 \) and \( th_2 \) are two threads in the simulation of a model whose Segment Graph is Fig. 8(a). \( th_1 \) is ready to run in \( seg_1 \) with timestamp \( 3:0 \), and \( th_2 \) is still running in \( seg_2 \) with timestamp \( 1:0 \).

Conflicts\((th_1)\) in Algorithm 2 will return \text{true} because \( th_2 \) is possible to enter \( seg_2 \) with timestamp \( 2:0 \) that is before \( th_1 \). Thus, the scheduler cannot issue \( th_1 \) out-of-order.

However, Conflict\((th_1)\) in Algorithm 6 will return \text{false} when \( M = 1 \) or 2. With prediction information, the scheduler knows that \( th_1 \) (in \( seg_2 \)) 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 \( th_1 \) out-of-order at the current scheduling step. We conclude, the prediction information allows to eliminate false conflicts.

VI. HYBRID MULTI-THREADING INFRASTRUCTURE

The out-of-order PDES scheduler aims at the most effective utilization of the available hardware on the multi-core host. This requires an efficient multi-threading infrastructure using user-level and kernel-level threads. Kernel-level threads maintained by the operating system (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 multi-threading libraries, such as QuickThreads [22], incur very little overhead in cooperative multi-threading, but can utilize only a single core.

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

Fig. 10 compares the simulator infrastructure using the three multi-threading 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 Posix thread 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-level 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 multi-core 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 out-of-order 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 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\(^6\) with 6 parallel cores each. Thus, in total the server hardware supports up to 12 threads running in parallel.

A. Evaluation of Out-of-Order PDES

We have implemented our parallel simulator in a SpecC\(^7\)-based system design environment [24] and conducted experiments on six multi-media 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 out-of-order 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 multi-threading libraries.

![Diagram of DES using QuickThreads](a)

![Diagram of DES using PosixThreads](b)

![Diagram of PDES using PosixThreads](c)

![Diagram of PDES using HybridThreads](d)

Fig. 10. User-level and kernel-level multi-threading models in discrete event simulators.

Table III shows the statistics and measurements for this model. Note that the conflict table is very sparse, allowing 74.11% of the threads to be issued out-of-order. While the synchronous PDES cannot gain performance due to in-order time barriers and synchronization overhead, our out-of-order simulator shows more than twice the simulation speed.

| Applications | Spec C | Arch | Scheduler | Net
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG Image Encoder</td>
<td>JPEG, H.264, Fibo, Mandelbrot</td>
<td>320</td>
<td>89</td>
<td>256</td>
</tr>
<tr>
<td>Total conflicts in CT ([N, N])</td>
<td>0/024000</td>
<td>0/017921</td>
<td>0/063536</td>
<td>0/063536</td>
</tr>
<tr>
<td># threads issued</td>
<td>0/024000</td>
<td>0/017921</td>
<td>0/063536</td>
<td>0/063536</td>
</tr>
<tr>
<td># threads issued out-of-order (38.58%)</td>
<td>0/0010</td>
<td>0/0010</td>
<td>0/0010</td>
<td>0/0010</td>
</tr>
</tbody>
</table>

1) 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. 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).

\(^6\)To ensure consistent timing measurements, we have disabled the dynamic frequency scaling and turbo mode of the processors.

\(^7\)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 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 demonstrate models at different abstraction levels, we have created four models (specification, architecture mapped, scheduling refined, 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 out-of-order.

Table VI and V show the corresponding compiler and simulator run times. While the compile time increases about 10%, the out-of-order 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 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 intra-prediction. For our simulation, we use a video stream of 1079 frames, 1280x720 pixels per frame, each with 4 slices of equal size. The experimental results for this industrial-size design are listed in Table IV, V and 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 out-of-order (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. \( T_{fib}(n) = 1.618 \cdot T_{fib}(n-1) \).

For this highly parallel application, synchronous PDES shows almost 500% speedup and our out-of-order 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 3x for OoO PDES. However, the absolute compile time of only 2 seconds 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 12x speedup and our out-of-order 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 multi-media application, we have parallelized a 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. There are 5 bi-directionally predicted slices (B-slices) between the intra-coded slices (I-slices) or forward predicted slices (P-slices). Thus, among every five consecutive frames, four frames need inter-frame prediction.

Table V shows that when using the same PosixThread library, the synchronous PDES can hardly achieve any speedup while the out-of-order PDES accelerates the simulation by almost 2x. 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 Multi-Threading Approach

Overall, Table V shows quite different performance for the multi-threading 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 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 rightmost 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 multi-threading 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 Segment Graph [20], can guarantee the most efficient out-of-order simulation at almost no cost.

**VIII. Conclusions and Future Work**

Highest 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 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 segment graph of the application. Using fast conflict table lookups, our out-of-order 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 multi-threading infrastructure that combines the benefits of multi-core 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 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 (i.e. add pointer analysis) and look into additional methods to further improve the simulation speed (e.g. many-core target platforms).

**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