Out-of-Order Parallel Simulation for ESL Design

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

***Abstract*—At the Electronic System Level (ESL), design validation often relies on discrete event (DE) simulation. Recently, parallel simulators have been proposed which increase simulation speed by using multiple cores available on today's PCs. However, the total order of time in DE simulation is a bottleneck that severely limits the benefits of parallel simulation. This paper presents a new out-of-order simulator for multi-core parallel DE simulation of hardware/software designs at any abstraction level. By localizing the simulation time and carefully handling events at different times, a system model can be simulated following a partial order of time. Subject to automatic static data analysis at compile time and table-based decisions at run time, threads can be issued early which reduces the idle time of available cores. Our experiments show high performance gains in simulation speed with only a small increase of compile time.**

I. Introduction

ESL design models specified in System-level Description Languages (SLDLs), such as SystemC [8] and SpecC [7], are usually validated using simulation. The simulator is a regular discrete event (DE) simulator. Within a single process, multiple concurrent threads emulate the parallelism in the design model. Typically, the multi-threading model is cooperative (i.e. non-preemptive), which simplifies the communication through events and variables in shared memory. Recent works [10], [11], [2] aim to utilize the parallel computation resources available in multi-core CPUs that are common in today's host PCs. Here, an extended simulation kernel uses OS kernel threads and additional synchronization for running multiple threads in parallel on the available cores. However, the number of threads that can run in parallel at each scheduling step is often very limited. The inner loops for delta-cycle and simulation time update in DE simulation severely limit the usable parallelism.

In this work, we relax the global in-order event and timing update based on compile-time automatic static analysis of the threads and their potential conflicts. Using the analysis results, our extended simulation kernel can then at run-time quickly decide whether or not any conflicts between candidate threads exist. If not, it issues threads early (with local timestamps). In turn, parallelism is maximized and simulation speed increases.

In other words, we extend parallel ESL simulation by aggressive out-of-order execution for higher simulation speed while maintaining all SLDL semantics and accurate timing.

After a brief discussion of related work, Section II motivates our idea using a simple DVD player example. Section III presents our out-of-order parallel DE simulation in detail and Section IV shows its higher simulation speed in experiments.

978-3-9810801-8-6/DATE12/©2012 EDAA

A. Related Work

Parallel Discrete Event Simulation (PDES) is a well-studied subject [1], [6], [9]. Two major synchronization paradigms exist, namely conservative and optimistic [6]. *Conservative* PDES ensures in-order event execution. In contrast, the *optimistic* paradigm assumes that every event is safe when executed 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.

C-based SLDLs use DE simulation driven by events and simulation time advances. To interpret "zero-delay" semantics of SLDLs, the notion of *delta-cycles* imposes a partial order on the events that happen at the same time [8]. PDES with delta-cycle notion has been also been explored. For example, [10], [11], [3], [2] extend the SystemC and SpecC kernels respectively to allow parallel simulation on multi-core processors. [10], [11], [2] apply PDES to SystemC and SpecC targeting symmetric multi-processing (SMP) architectures by using conservative synchronization. However, as an obstacle, the global simulation time is shared by all threads.

II. Motivation

While the reference simulators for both SystemC and SpecC are single-threaded, parallel approaches like [10], [2] take advantage of the fact that several threads running at the same simulation time and delta-cycle can be issued in parallel. However, even these PDES approaches impose a total order on event delivery and time advance which makes delta- and time-cycles absolute barriers for thread execution. More specifically, when a thread finishes its execution for a cycle, it has to wait until all other active threads complete their execution for the same cycle. Only then the simulator advances to the next *delta* or *time* cycle. Additionally available CPU cores are idle until all threads have reached the cycle barrier.

As a motivating example, Fig. 1 shows a high-level model of a DVD player which decodes a stream with 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, regular PDES approaches cannot exploit the parallelism in this example. Fig. 2(a) shows the thread scheduling along the time line. Except for the very first scheduling step, only one thread can run at any time. Note that it is not data dependency but only the global timing that prevents parallel execution in the simulator.

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

III. Out-of-Order Parallel Simulation

Regular DE simulation imposes a total order on event processing and time advancing, reducing the potential for parallel execution. We now propose a new out-of-order simulation scheme where timing is only partially ordered. We localize the global simulation time (*time*, *delta*) for each thread and allow threads without potential data or event conflicts to run ahead of time while other working threads are still running with earlier timestamps. To avoid any read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) hazards on shared variables, we use static analysis to detect potentially conflicting code segments.

A. Definitions

To formally describe our out-of-order PDES algorithm, we introduce the following notations:

1) We define simulation time as tuple (t, δ) where t =time, δ =delta-cycle, and order time stamps as follows:
   - **equal**: $(t_1, \delta_1) = (t_2, \delta_2)$, iff $t_1 = t_2$, $\delta_1 = \delta_2$
   - **before**: $(t_1, \delta_1) < (t_2, \delta_2)$, iff $t_1 < t_2$, or $t_1 = t_2$, $\delta_1 < \delta_2$
   - **after**: $(t_1, \delta_1) > (t_2, \delta_2)$, iff $t_1 > t_2$, or $t_1 = t_2$, $\delta_1 > \delta_2$
2) Each thread th has its own time (t_{th}, δ_{th}).
3) Since events can be notified multiple times and at different simulation times, we note an event e notified at (t, δ) as tuple (id_e, t_e, δ_e) and define: **EVENTS**= $\cup$**EVENTS**$_{t,\delta}$ where **EVENTS**$_{t,\delta}$ = $\{(id_e, t_e, \delta_e) \mid t_e = t, \delta_e = \delta)\}$
4) For regular DE simulation, typically several sets of queued threads are defined, such as **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**}. These sets exist at all times and threads move from one to the other during simulation, as shown in Fig. 3(a).

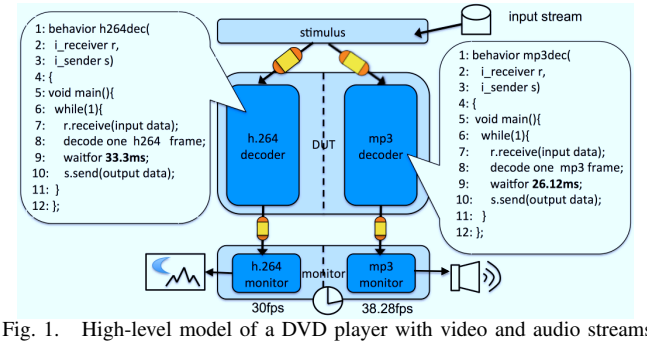
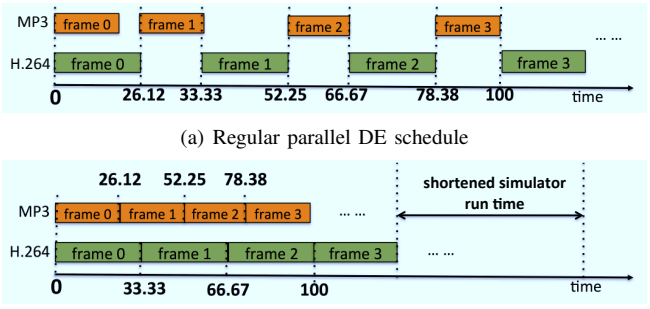


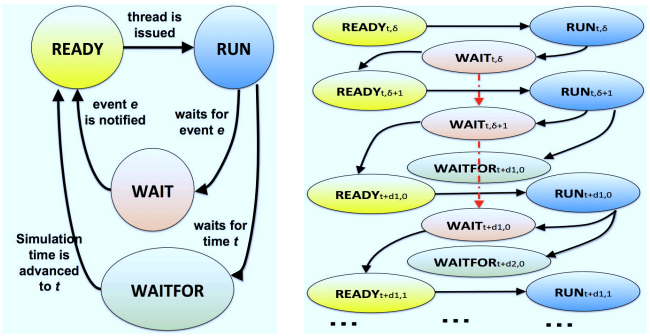
Fig. 1. High-level model of a DVD player with video and audio streams.



(a) Regular parallel DE schedule

(b) Out-of-order parallel DE schedule

Fig. 2. DE scheduling of the high-level DVD player model



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

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

Now, for our out-of-order PDES, we define multiple sets with different time stamps, which we dynamically create and delete as needed, as illustrated in Fig. 3(b).

Specifically, we define:

- **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**, **JOINING**, **COMPLETE**}
- **READY** = $\cup$**READY**$_{t,\delta}$, **READY**$_{t,\delta}$={th | th is ready to run at (t, δ)}
- **RUN** = $\cup$**RUN**$_{t,\delta}$, **RUN**$_{t,\delta}$={th | th is running at (t, δ)}
- **WAIT** = $\cup$**WAIT**$_{t,\delta}$, **WAIT**$_{t,\delta}$={th | th is waiting since (t, δ) for events (id_e, t_e, δ_e), where $(t_e, \delta_e) \geq (t, \delta)$}
- **WAITFOR** = $\cup$**WAITFOR**$_{t,\delta}$, **WAITFOR**$_{t,\delta}$={th | th is waiting for simulation time advance to $(t, 0)$}
- **JOINING** = $\cup$**JOINING**$_{t,\delta}$, **JOINING**$_{t,\delta}$={th | th created child threads at (t, δ), and waits for them to complete}
- **COMPLETE** = $\cup$**COMPLETE**$_{t,\delta}$, **COMPLETE**$_{t,\delta}$ = {th | th completed its execution at (t, δ)}

Note that our implementation orders theses sets by increasing time stamps for efficiency.

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

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

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

The thread and event sets evolve during simulation as illustrated in Fig. 3. Whenever the sets **READY**$_{t,\delta}$ and **RUN**$_{t,\delta}$ are empty and there are no **WAIT** or **WAITFOR** queues with earlier timestamps, the scheduler deletes **READY**$_{t,\delta}$ and **RUN**$_{t,\delta}$, as well as any events with the same timestamp **EVENTS**$_{t,\delta}$.

B. Out-of-order Scheduling Algorithm

Algorithm 1 shows the scheduling algorithm of our out-of-order parallel DE simulator. 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 waiting time and issued for execution if qualified (line 28). Finally, if no thread 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 any 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.

C. Static Conflict Analysis at Compile-Time

We use static analysis of the application code to determine whether or not a thread is qualified to run early/out-of-order. In particular, we have to prevent parallel data access to shared variables, namely read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW).

Fig. 4 shows a simple example of a WAW conflict where two threads th_1 and th_2 write to the same variable i at different times. Simulation semantics require that th_1 executes first and sets i to 0 at time $(5,0)$, followed by th_2 setting i to its final value 1 at time $(10,0)$. Now, if our simulator would issue the threads th_1 and th_2 out-of-order, we 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 (which we will refer to as a second *segment*) if the functions $f()$ and $g()$ are independent.

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

1) Thread Segments and Segment Graph: Switching back and forth between **RUNNING** and **WAITING**, threads execute different segments of their code. Notably, two threads with shared variables typically conflict only in few segments of their execution. Thus, we can refine our thread conflict analysis by using the following definitions:

Algorithm 1 Out-of-order PDES Algorithm

```
 1: /* trigger events */
 2: for all th ∈ WAIT do
 3:     if ∃ event (id_e, t_e, δ_e), th awaits e, and (t_e, δ_e) ≥ (t_th, δ_th) then
 4:         move th from WAIT_{t_th,δ_th} to READY_{t_e,δ_e+1}
 5:         t_th = t_e; δ_th = δ_e + 1
 6:     end if
 7: end for
 8: /* clean up 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
```

- **Segment** s_i: code portion executed by a thread between two scheduling steps.
- **Segment Boundary** v_i: SLDL statements which call the scheduler, i.e. *wait*, *waitfor*, *par*. Note that segments s_i and segment boundaries v_i form a directed graph. s_i is the segment followed by segment boundary v_i. v_i can be followed by multiple segment boundaries, and s_i can be composed of multiple code portions.
- **Segment Graph (SG)**: **SG=(V, E)**, where **V** = $\{v \mid v$ is a segment boundary$\}$, **E**=$\{e_{ij} \mid e_{ij}$ is the code portion between v_i and v_j, where v_j is reached after $v_i\}$.
- **Segment Conflict Table (CTab[N, N])**: CTab[i, j] = $false$, iff there is no data conflict between the segments s_i and s_j; otherwise, CTab[i, j] = $true$. N is the total number of segments in the application code.

Fig. 5 shows a simple source code example and its Control Flow Graph (CFG). From the CFG, we derive the Segment Graph (SG) by converting every scheduler call (i.e. *wait*,

Algorithm 2 Build the Segment Graph

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

waitfor) into nodes and all possible flows of control into edges. Here segment node 3 corresponds to the *wait e2* statement. From there, control reaches either node 4 (*wait e3*) through blocks e, g, h or node 5 (*wait e4*) through blocks e, g, j.

For the general case, our compiler uses Algorithm 2 to traverse an application's CFG following all branches, function calls and threads, and recursively build the corresponding SG.

2) Computing the Segment Conflict Table: Based on the SG, we can easily compute a table of conflicting segments.

First, 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).

For example, a statement $a = a + b$ creates an access list $\{a$(RW), b(R)$\}$.

Our compiler computes the variable access lists for each segment during the generation of the SG (line 43 in Algorithm 2, *ExtendAccess()*). Note that we currently do not perform any pointer analysis (future work). Instead, we conservatively mark all segments with pointer accesses (Ptr) as conflicting. However, we do follow port mappings through the structural hierarchy of the design model and store the actual target variables in the access list.

Finally, we create the segment conflict table $CTab[N, N]$ by comparing the access lists for each segment pair. If two segments s_i and s_j share any variable with access type (W) or (RW), or there is any pointer access by s_i or s_j, then we mark this as a conflict: $CTab[i, j] = CTab[j, i] = true$. Otherwise, there is no conflict: $CTab[i, j] = CTab[j, i] = false$.

Algorithm 3 Scheduling Conflict Detection

```
 1: bool HasNoConflicts(Thread th)
 2: {
 3:   for all th2 ∈ RUN where (th2.t, th2.δ) ≠ (th.t, th.δ) do
 4:     if (Conflict(th, th2)) then return false end if
 5:   end for
 6:   for all th2 ∈ READY where (th2.t, th2.δ) < (th.t, th.δ) do
 7:     if (Conflict(th, th2)) then return false end if
 8:   end for
 9:   for all th2 ∈ WAIT where (th2.t, th2.δ) < (th.t, th.δ) do
10:     if (Conflict(th, th2)) then return false end if
11:   end for
12:   return true
13: }
14: bool Conflict(Thread th1, Thread th2)
15: {
16:   if (th2 may enter another segment before (th1.t, th1.δ)) then
17:     return true end if
18:   if (CTab[th.segID, th2.segID]) then
19:     return true end if
20:   return false
21: }
```

3) Scheduling Conflict Detection: While the segment graph and conflict table are built at compile time, the simulator needs to check at run-time whether an available thread at a particular segment can be issued out-of-order, i.e. without conflict. To do this efficiently, we use a table-lookup in $CTab[i, j]$ and only run our out-of-order scheduler when a CPU core is idle.

In order to provide the scheduler with the next segment a given thread is about to execute, our compiler instruments the SLDL code such that the segment ID is passed to the scheduler as an additional argument when the thread executes a *wait*, *waitfor*, or other scheduling statement. At run-time, the scheduler then calls the **HasNoConflicts**(th) function to determine whether or not to issue the thread th early. As shown in Algorithm 3, the **HasNoConflicts**(th) function checks for potential conflicts with all parallel running threads (in **RUN**), as well as all waiting threads in the **READY** and **WAIT** queues with an earlier time stamp than th. Note that each check can be performed in constant time (O(1)) due to the table-lookup in function **Conflict**(th_1, th_2).

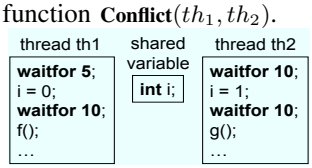


Fig. 4. Write-after-write (WAW) conflict between two parallel threads.

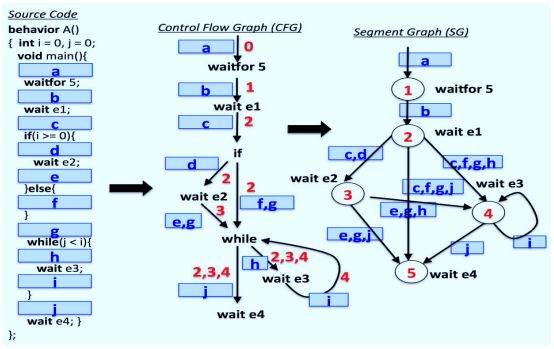


Fig. 5. Converting SLDL source code to control flow and segment graph.

IV. Experiments and Results

We have implemented the proposed out-of-order parallel simulator in a SpecC[2]-based system design environment [5] and conducted experiments on three multi-media applications shown in Fig. 6. To demonstrate the benefits of our out-of-order PDES, we compare the compiler and simulator run times with the traditional single-threaded reference and a regular parallel implementation [2] without out-of-order scheduling. All experiments have been performed on the same host PC with a 4-core CPU (Intel$^{(R)}$ Core$^{(TM)}$2 Quad) at 3.0 GHz.

TABLE I
Experimental results for the abstract DVD player example

Simulator:	Single-thread reference	Multi-core	
		Regular PDES	Out-of-order PDES
compile time	0.71s	0.88s / -19.3%	0.96s / **-26.0%**
simulator time	7.82s	7.98s / -2.0%	3.89s / **+101.0%**
#segments N		50	
total conflicts in $CTab[N,N]$		160/2500 (6.4%)	
#threads issued		1008	
#threads issued out-of-order		**791 (78.47%)**	

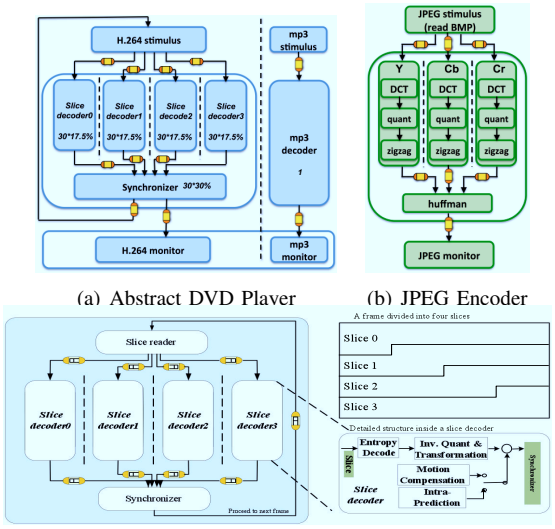
TABLE II
Out-of-order PDES statistics for JPEG and H.264 examples

JPEG Image Encoder				
TLM abstraction:	spec	arch	sched	net
#segments N	42	40	42	43
total conflicts in $CTab[N,N]$	199/1764 (11.3%)	184/1600 (11.5%)	199/1764 (11.3%)	212/1849 (11.5%)
#threads issued	271	268	268	4861
#threads issued out-of-order	**176 (64.9%)**	**173 (64.5%)**	**176 (65.7%)**	**2310 (47.5%)**
H.264 Video Decoder				
TLM abstraction:	spec	arch	sched	net
#segments N	67	67	69	70
total conflicts in $CTab[N,N]$	512/4489 (11.4%)	518/4489 (11.5%)	518/4761 (10.88%)	518/4900 (10.57%)
#threads issued	923017	921732	937329	1151318
#threads issued out-of-order	**179581 (19.46%)**	**176500 (19.15%)**	**177276 (18.91%)**	**317591 (27.58%)**

A. An Abstract Model of a DVD Player

Our first experiment uses the DVD player model shown in Fig. 6(a). Similar to the model discussed in Section II, 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.

[2]Due to its similarity, our results are equally applicable to SystemC [4].



(a) Abstract DVD Player (b) JPEG Encoder

(c) Parallel Video Decoder based on H.264/AVC standard [12]

Fig. 6. Example design models for out-of-order PDES experiments.

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. The resulting workload of the major blocks is shown in the diagram.

Table I shows the statistics and measurements for this model. Note that the conflict table is very sparse, allowing 78.47% of the threads to be issued out-of-order. While the regular PDES loses performance due to in-order time barriers and synchronization overheads, our out-of-order simulator shows twice the simulation speed.

B. A JPEG Encoder Model

Our second experiment uses the JPEG image encoder model shown in Fig. 6(b). The stimulus reads a BMP color image with 3216x2136 pixels and performs color-space conversion from RGB to YCbCr. Since encoding of the three color components (Y, Cb, Cr) is independent, our JPEG encoder performs the DCT, quantization and zigzag modules for the colors in parallel, followed by a sequential Huffman encoder at the end. The JPEG monitor collects the encoded data and stores it in the output file.

To show the increased simulation speed also for models at different abstraction levels, we have created four models (*spec*, *arch*, *sched*, *net*) with increasing amount of implementation detail, down to a network model with detailed bus transactions. Table II lists the PDES statistics and shows that, for the JPEG encoder, about half or more of all threads can be issued out-of-order. Table III shows the corresponding compiler and simulator run times. While the compile time increases similar to the regular parallel compiler, the simulation speed improves by about 138%, more than 5 times the gain of the regular parallel simulator.

TABLE III
EXPERIMENTAL RESULTS FOR THE JPEG IMAGE ENCODER AND THE H.264 VIDEO DECODER EXAMPLES

Simulator:		Single-thread reference		Multi-core			
				Regular parallel		Out-of-Order parallel	
		compile time [sec]	simulator time [sec]	compile time [sec] / speedup	simulator time [sec] / speedup	compile time [sec] / speedup	simulator time [sec] / speedup
JPEG Encoder	spec	0.80	2.23	1.10 / -27.3%	1.84 / +21.2%	1.13 / -29.2%	0.93 / **+139.8%**
	arch	1.09	2.23	1.35 / -20.0%	1.80 / +23.9%	1.37 / -21.2%	0.93 / **+140.0%**
	sched	1.14	2.24	1.41 / -19.9%	1.83 / +22.4%	1.43 / -21.0%	0.92 / **+143.5%**
	net	1.34	2.90	1.59 / -17.6%	2.33 / +24.5%	1.63 / -19.6%	1.26 / **+130.1%**
H.264 Decoder	spec	12.35	97.16	13.91 / -11.2%	97.33 / -0.2%	18.13 / -31.9%	60.33 / **+61.1%**
	arch	11.97	97.81	12.72 / -5.9%	99.93 / -2.1%	18.46 / -35.2%	60.77 / **+61.0%**
	sched	18.18	100.20	18.84 / -3.5%	100.18 / +0.0%	24.80 / -26.7%	60.96 / **+64.4%**
	net	18.57	111.07	19.52 / -4.9%	106.14 / +4.6%	26.06 / -28.7%	66.25 / **+67.7%**

C. A Detailed H.264 Decoder Model

Our third experiment simulates a complex parallel video decoder based on the H.264/AVC standard [12]. Fig. 6(c) shows a high-level block diagram of our model. While this model is similar at the highest level to the video part of the abstract DVD player, it contains many more blocks at lower levels which implement the complete H.264 reference application consisting of about 40,000 lines of code. Internally, each slice decoder consists of complex H.264 decoder functions entropy decoding, inverse quantization and transformation, motion compensation, and intra-prediction. For our simulation, we use a video stream of 1079 frames and 1280x720 pixels per frame, each with 4 slices of equal size.

Our simulation results for this industrial-size design are listed in the lower half of Table II and Table III, again for four models at different abstraction levels, including a network model with detailed bus transactions. Due to the large complexity of the models, the compile time increases by up to 35.2%[3]. This, however, is insignificant when compared to the much longer simulator run times.

While the regular parallel simulator shows almost no improvement in simulation speed, our proposed simulator shows more than 60% gain since many of the threads can be issued out-of-order (see Table II).

V. CONCLUSIONS AND FUTURE WORK

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

Experimental results show that, with only a small increase in compile time, our simulator is significantly faster than the traditional single-threaded reference implementation, as well as a regular multi-core parallel simulator.

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 will optimize the static code analysis and look into additional methods to further improve the simulation speed.

[3]We are aware of several opportunities for optimizing the static analysis at compile-time and will address this in future work.

ACKNOWLEDGMENT

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

REFERENCES

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

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

[3] B. Chopard, P. Combes, and J. Zory. A Conservative Approach to SystemC Parallelization. In *International Conference on Computational Science (4)*, pages 653–660, 2006.

[4] R. Dömer, W. Chen, X. Han, and A. Gerstlauer. Multi-Core Parallel Simulation of System-Level Description Languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 311–316, 2011.

[5] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski. System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008(647953):13 pages, 2008.

[6] R. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, Oct 1990.

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

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

[9] D. Nicol and P. Heidelberger. Parallel Execution for Serial Simulators. *ACM Transactions on Modeling and Computer Simulation*, 6(3):210–242, July 1996.

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

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

[12] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560 –576, july 2003.