Advances in Parallel Discrete Event Simulation for Electronic System-Level Design

Weiwei Chen, Xu Han, Che-Wei Chang, and Rainer Dömer
University of California

Editors' notes:
The authors target the speeding up of parallel discrete event simulations in transaction-level models.

—Rasit Onur Topaloglu, IBM, and Beven Baas, University of California, Davis

The large complexity of modern embedded systems with their heterogeneous components, complex interconnects, and sophisticated functionality poses challenges to system validation and debugging. At the Electronic System-Level (ESL), accurate yet fast simulation is key to enabling effective and efficient model validation and implementation. This paper presents and compares several simulation techniques for designs described in System-Level Description Languages (SLDLs). In particular, the well-known approach of Parallel Discrete Event Simulation (PDES) [1] has recently gained attention again due to the inexpensive availability of parallel processing in today's multi-core CPU hosts. PDES holds the promise to map the explicit parallelism described in SLDL models efficiently onto the parallel cores available on the simulation host. As such, it can exploit the available parallelism and significantly reduce the simulation time.

Digital Object Identifier 10.1109/MDT.2012.2226015
Date of publication: 23 October 2012; date of current version: 11 April 2013.

Related work

The validation of ESL models is typically based on Discrete Event (DE) simulation which is driven by events and simulation time advances. Most ESL design frameworks today still rely on synchronous discrete event simulators which issue only a single thread at any time to avoid the complex synchronization of the concurrent threads. As such, the simulator kernel becomes an obstacle to improving simulation performance on multi-core hosts.

Distributed Parallel Simulation [2], [3] breaks a model into modules, dispatches them on geographically distributed simulation hosts, and then runs the simulation in parallel. However, model partitioning is difficult and the network speed becomes a bottleneck due to the frequently needed communication.

Specialized hardware including Field-Programmable Gate Array (FPGA) [4] and Graphics Processing Units (GPU) [5] can also boost simulation speed. The methodology presented in [6] parallelizes SystemC simulation across multicore CPUs and GPUs but the model needs to be partitioned on the heterogeneous simulator units.

Other techniques run multiple simulators in parallel and synchronize them. The Wisconsin Wind Tunnel [7] uses a conservative time bucket synchronization scheme to synchronize simulators

 Copublished by the IEEE CEDA, IEEE CASS, IEEE SSCS, and TTTC

at a predefined interval. In [8], a simulation backplane handles the synchronization between wrapped simulators and the system optimizes the period of the synchronization message transfer. Both techniques significantly speedup the simulation at the cost of timing accuracy.

PDES research on SLDL simulation provides a general approach for parallel simulation of ESL models. An extension of the SystemC kernel [9], [10] actually allows parallel execution on multicore processors. The modified simulator kernel issues multiple OS kernel threads in parallel and synchronizes them in each scheduling step. The SpecC-based approach in [11] is similar. However, a synchronization protection mechanism automatically instruments communication channels. There is no need to work around the cooperative SystemC execution semantics, nor for a specially prepared channel library.

SLDL DE simulation uses the notion of *delta-cycles* which interpret the "zero-delay" semantics of SLDLs and impose a partial order on the events that happen at the same time. Synchronous PDES approaches including [9], [11] impose a total order on simulation advances which makes delta and time cycles absolute *simulation cycle barriers* for thread execution. When a thread finishes its execution for a cycle, it has to wait until all other active threads complete the same cycle. Only then the simulator advances to the next delta or time cycle. Available CPU cores are idle until all threads have reached the barrier.

To address this limitation, *out-of-order PDES* [12] breaks the simulation cycle barrier and aggressively issues multiple threads in parallel even if they are in different cycles. This keeps the available CPU cores in the host as busy as possible. In contrast to synchronous PDES, timing is only partially ordered in out-of-order PDES.

In comparison to our work in [11], [12], we review and compare the major PDES approaches here. We highlight the advanced out-of-order PDES and provide results for a new highly parallel benchmark example (fibo_timed) and additional embedded applications for image, video and audio processing which compare synchronous and out-of-order PDES.

Parallel discrete event simulation

DE simulation creates threads for the explicit parallelism in the model (e.g. *par* and *pipe* statements in SpecC, and *SC_THREADS* in SystemC).[1] A scheduler manages the threads by use of queues, such as **READY**, which contains all those that are ready to execute, and **WAIT**, which contains threads waiting for events. Threads switch between **READY** and **WAIT** during simulation subject to event notification and time advances. Events are delivered in an inner loop called *delta-cycle* and simulation time advances in an outer loop *time-cycle*.

PDES approaches differ in the way threads are scheduled and, in particular, whether or not threads are allowed to run in parallel. A simple example can illustrate this. Figure 1a shows a high-level model of a DVD player which decodes the MP3 audio and H.264 video frames of the media stream using separate decoders. The decoders work in parallel and output the decoded frames according to their rate, 30 FPS for video (delay 33.3 ms) and 38.28 FPS for audio (delay 26.12 ms).

Traditional DE simulation executes threads sequentially, only one at any time, and when running at the same simulated time, i.e. within a delta-cycle, the choice of the next thread to run is non-deterministic (by definition). For the DVD player, this schedule is shown in Figure 1c.

In contrast, PDES approaches improve simulator performance by executing suitable threads in parallel on a multi-core host. Figure 1d shows the scheduling under synchronous and Figure 1e under out-of-order PDES. While synchronous PDES parallelizes only threads running at the same simulated time, i.e. only the very first frame at time 0, out-of-order PDES localizes the simulation time and executes independent threads in parallel *out-of-order*. For the DVD model, this results in significantly reduced simulator run-time.

Synchronous PDES

Figure 2a shows the control flow of the synchronous PDES scheduler. In each cycle, it picks multiple threads from the **READY** queue and runs them in parallel. In particular, the loop on the left side of the graph moves threads from **READY** to **RUN** as long as processor cores are available.

[1]With the exception of different requirements for protection of communication and synchronization between concurrent threads, as outlined in [11], this section applies equally to both SystemC and SpecC SLDLs.

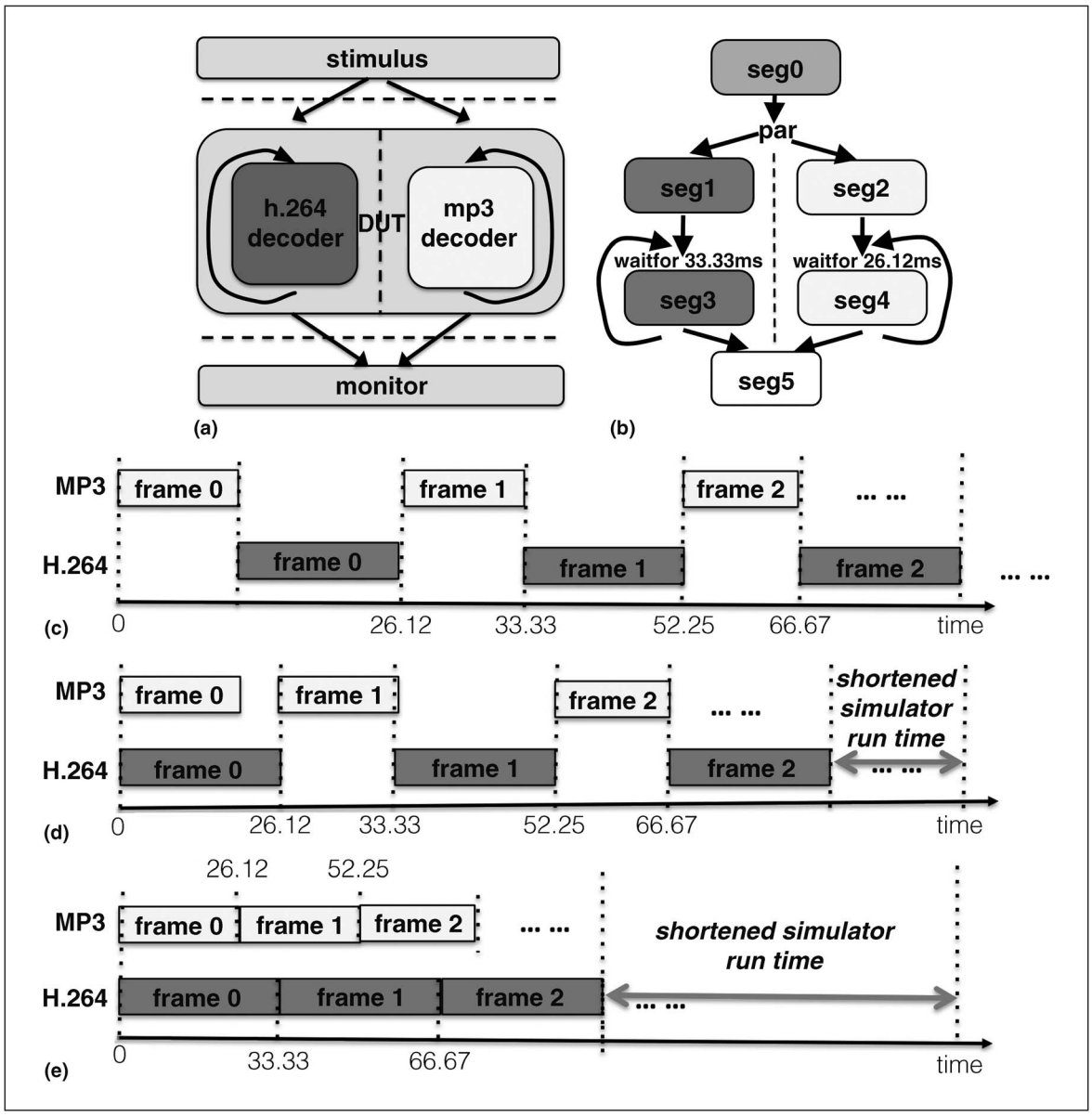

Figure 1. High-level DVD player example. (a) Model structure; (b) segment graph; (c) traditional DE simulation schedule; (d) synchronous PDES schedule; (e) out-of-order PDES schedule.

Explicit synchronization is required for running multiple threads safely in parallel. The simulator data structures, including thread queues and event lists, and shared variables in communication channels must be properly protected by locks for mutually exclusive access by the concurrent threads.

Note that synchronous PDES only parallelizes threads running in the same delta-cycle and the global simulation time advances only when no threads are running. CPU cores are idle when there are not enough threads in the same cycle or the workloads of the parallel threads are imbalanced.

Out-of-order PDES

Figure 2b shows the more aggressive algorithm of out-of-order PDES which issues threads that are independent early without waiting for global time advance. In other words, out-of-order PDES advances simulation cycles in a partial order using thread-local timing. Each thread processes its simulation cycles

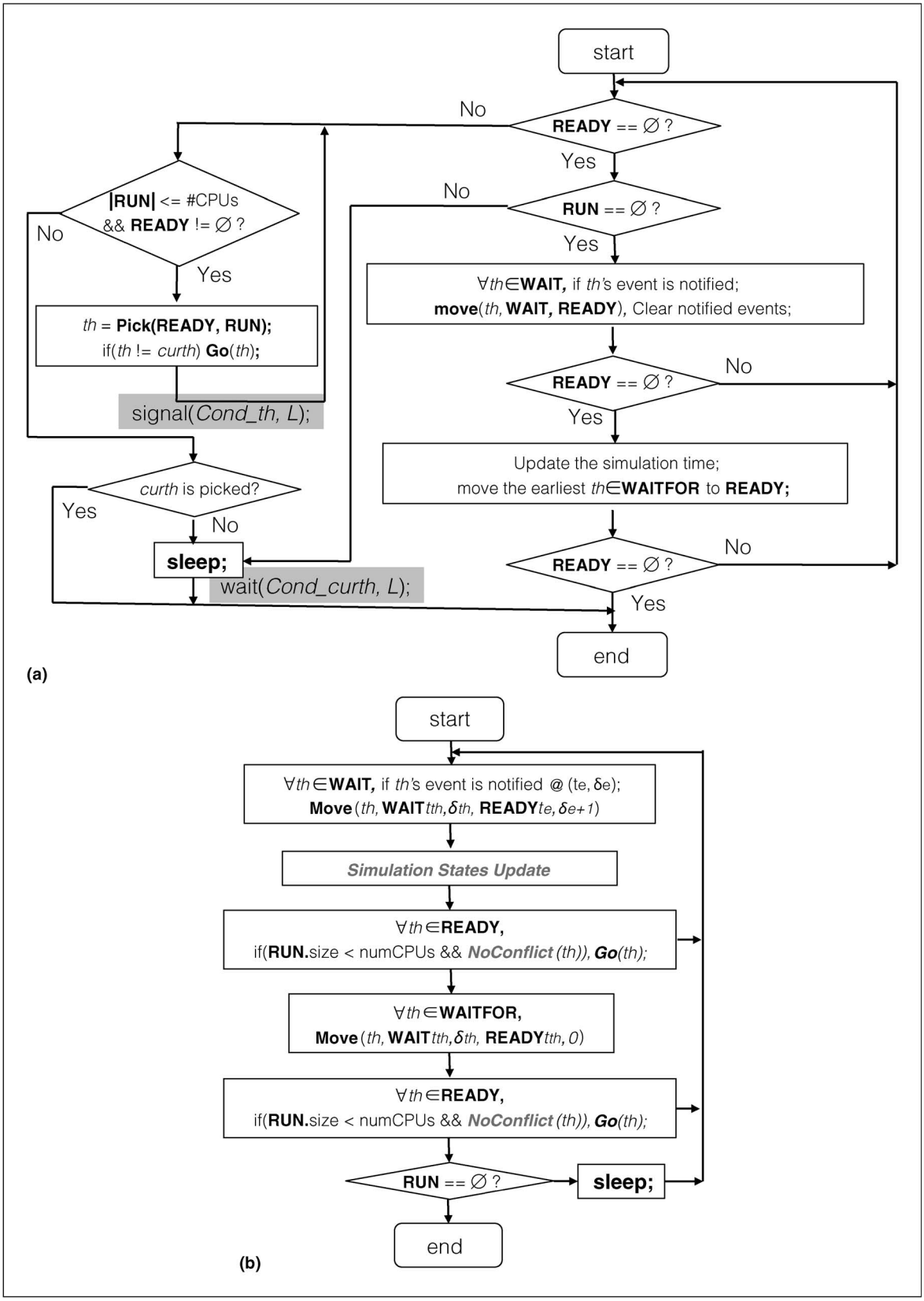

Figure 2. PDES algorithms. (a) Synchronous PDES scheduler. (b) Out-of-order PDES scheduler.

Table 1 Comparison of traditional, synchronous, and out-of-order PDES approaches.

	Traditional DE simulation	Synchronous PDES, e.g. [9], [11]	Out-of-Order PDES, e.g. [12]
Simulation Time	One global time tuple (t, δ) shared by every thread and event		Local time for each thread *th* as tuple (t_{th}, δ_{th}). A total order of time is defined with the following relations: **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 $t_1 = t_2, \delta_1 > \delta_2$.
Event Description	Events are identified by their *id*s, i.e, event (*id*).		A timestamp is added to identify every event, i.e. event (*id, t, δ*).
Simulation Thread Sets	**READY**, **RUN**, **WAIT**, **WAITFOR**, **JOINING**, **COMPLETE**		Threads are organized as subsets with the same timestamp (t_{th}, δ_{th}). Thread sets are the union of these subsets, i.e, **READY** = ∪**READY**$_{t,\delta}$, **RUN** = ∪**RUN**$_{t,\delta}$, **WAIT** = ∪**WAIT**$_{t,\delta}$, **WAITFOR** = ∪**WAITFOR**$_{t,\delta}$ ($\delta = 0$), where the subsets are ordered in increasing order of time (t,δ).
Threading Model	User-level or OS kernel-level		***OS kernel-level***
Run Time Scheduling	Event delivery in-order in delta-cycle loop. Time advance in-order in outer loop.		Event delivery out-of-order if no conflicts exist. Time advance out-of-order if no conflicts exist.
	Only one thread is active at one time. ***No parallelism***. ***No SMP utilization***.	Threads at same cycle run ***in parallel***. ***Limited parallelism***. ***Inefficient SMP utilization***.	Threads at same cycle or with no conflicts run ***in parallel.*** ***More parallelism.*** ***Efficient SMP utilization.***
Compile Time Analysis	No synchronization protection needed.	***Need synchronization protection for shared resources***, e,g, any user-defined and hierarchical channels	
	No conflict analysis needed.		Static conflict analysis derives Segment Graph (SG) from CFG, analyzes variable and event accesses, passes conflict table to scheduler. ***Compile time increases.***

as soon as possible subject only to dependencies on other threads [12].

While simulation time is localized to each thread, SLDL execution semantics are fully preserved because potential data and event hazards are conservatively analyzed at compile-time and checked at run-time. This is in contrast to *temporal decoupling* in SystemC TLM which trades off simulation speed against accuracy. Temporal decoupling allows threads to run ahead of the global simulation time without checking of dependencies and thus can lead to execution inconsistent with the standard semantics.

The conservative out-of-order PDES is also different from *speculative multithreading* techniques which are optimistic but have to *roll-back* in case the speculation turns out to be incorrect. Note that roll-backs are costly in the sense that either special hardware or complex software is needed to preserve the simulation semantics.

Out-of-order PDES uses static model analysis at compile-time to meet the standard simulation semantics. Using table-lookups at run-time, the scheduler then can make quick and safe decisions about issuing threads in parallel.

During simulation, threads call the scheduler at the end of every cycle so that the scheduler can decide and issue the threads for execution in the next cycle. We define the portion of code executed by a thread between two scheduling steps as a ***Segment*** (*seg*), and a ***Segment Boundary*** is defined by SLDL statements which call the scheduler, such as *wait* and *par*.

Together, segment boundaries (vertices) and segments (edges) form a directed graph, called ***Segment Graph***, which can be derived from the control flow graph of the model. As such, the segment graph shows the possible order of execution of the segments in the model.

Figure 1b shows the segment graph of the DVD example. Simulation starts at segment seg_0 and then creates two parallel threads for the two decoders in seg_1 and seg_2. Segments seg_3 and seg_4, respectively, follow after the segment boundaries created by the *wait-for-time* statements reflecting the frame delays.

In the DVD example, the audio and video frames are data-independent, so there are no conflicts between the segments. In general, however, a table of potential data and event conflicts among the segments is calculated by the compiler and passed to the simulator for checking at run-time [12]. Figure 2b lists the conflict table lookup (*NoConflict(th)*) by the scheduler which avoids any possible data and event hazards. Note that each conflict check can be performed in constant time ($O(1)$).

Table 1 compares out-of-order PDES in detail against the traditional DE simulation and synchronous PDES.

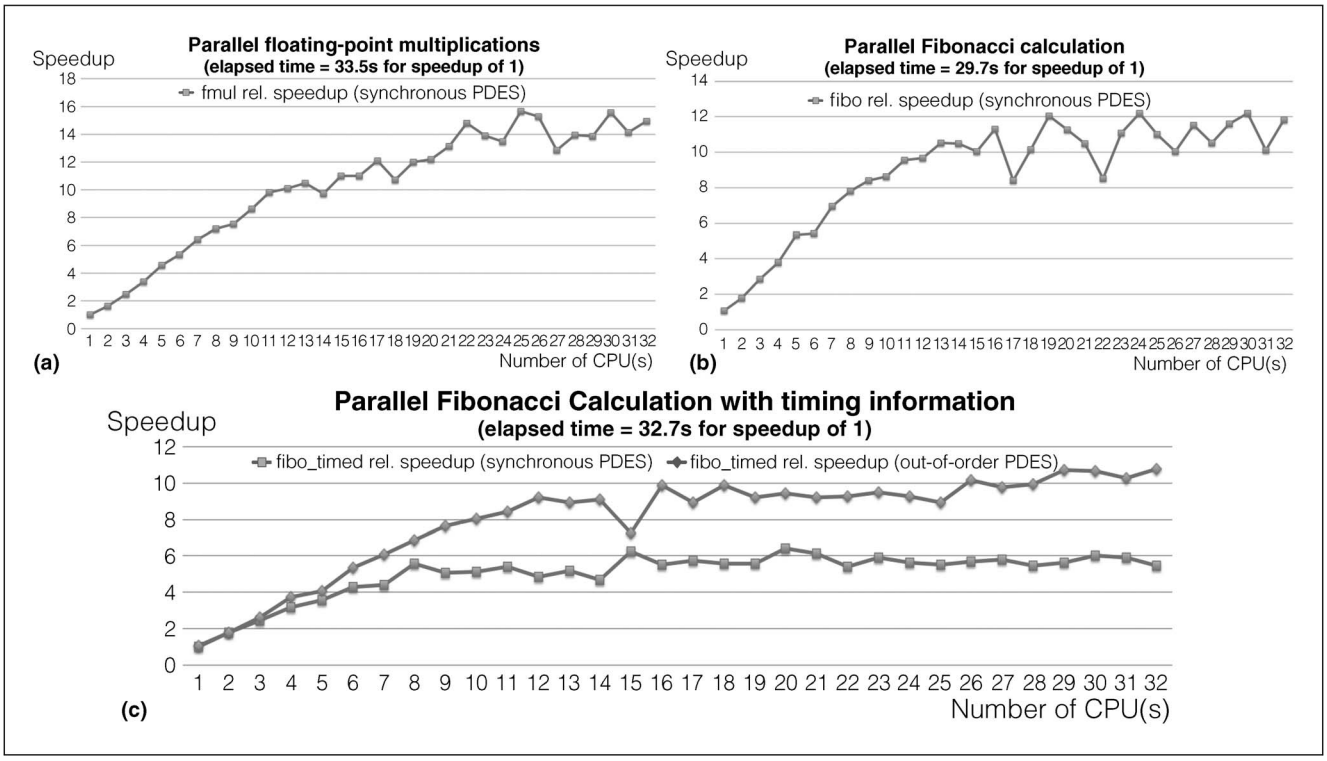

Figure 3. Simulation results for highly parallel benchmark models. (a) fmul (synchronous PDES); (b) fibo (synchronous PDES); (c) fibo_timed (synchronous PDES vs. out-of-order PDES).

Parallel system-level benchmarks

To demonstrate the potential of parallel simulation, we have designed three highly parallel benchmark models: a parallel floating-point multiplication example, a parallel recursive Fibonacci calculator, and a parallel recursive Fibonacci calculator with timing information. All these benchmarks are system-level models specified in SpecC SLDL.

For our experiments, we use a symmetric multiprocessing server running 64-bit Fedora 12 Linux. The multi-core hardware specifically consists of 2 Intel Xeon X5650 processors running at 2.67 GHz.[2] Each CPU contains 6 parallel cores, each of which supports 2 hyper-threads per core. Thus, in total the server hardware supports up to 24 threads running in parallel.

Parallel floating-point multiplications

Our first parallel benchmark **fmul** is a simple stress-test example for parallel floating-point calculations. Specifically, **fmul** creates 256 parallel instances which perform 10 million double-precision floating-point multiplications each. As an extreme example, the parallel threads are completely independent, i.e., do not communicate or share any variables.

The chart in Figure 3a shows the experimental results for our synchronous PDES simulator when executing this benchmark. To demonstrate the scalability of parallel execution on our server, we vary the number of parallel threads admitted by the parallel scheduler (the value `#CPUs` in Figure 2a) between 1 and 32.

We use the elapsed simulator run time for one core as the base (33.5 seconds). When plotting the relative speedup, one can see that, as expected, the simulation speed increases in nearly linear manner the more parallel cores are used and tops out when no more CPU cores are available. The maximal speedup is about 16× for this example on our 24-core server.

Parallel Fibonacci calculation

Our second parallel benchmark **fibo** calculates the Fibonacci series in parallel and recursive

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

fashion. Recall that a Fibonacci number is defined as the sum of the previous two Fibonacci numbers, $fib(n) = fib(n-1) + fib(n-2)$, and the first two numbers are $fib(0) = 0$ and $fib(1) = 1$. Our **fibo** design parallelizes the Fibonacci calculation by letting two parallel units compute the two previous numbers in the series. 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.

In contrast to the **fmul** example above, the **fibo** benchmark 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.

The **fibo** simulation results are plotted in Figure 3b. Again we use the elapsed simulator run time for one core as base (29.7 seconds). The curve for the relative simulation speedup shows the same increasing shape as in Figure 3a. Speed increases in nearly linear fashion until it reaches saturation at about a factor of 12×.

When comparing the **fmul** and **fibo** benchmark results, we notice a more regular behavior of the **fmul** example due to its even load and zero inter-thread communication.

Parallel Fibonacci calculation with timing information

Our third parallel benchmark **fibo_timed** is an extension of **fibo** with timing information. System models usually have timing information either back-annotated by estimation tools or added by the designers to evaluate the real-time behavior of the design. Compared to the untimed **fibo**, this timed benchmark is a more realistic embedded application example.

fibo_timed has the same structure as **fibo** with the same parallel decomposition depth (in our case 5). Timing information is annotated using *wait-for-time* statements at each leaf block where the classic recursive calculation method is used. The time delay is determined by the computational load of the unit, i.e. $T_{fib(n)} = 1.618 * T_{fib(n-1)}$.

Figure 3c plots the simulation results for both synchronous and out-of-order PDES. Using the 1-core elapsed simulator time as base (32.7 seconds for both simulators), the relative speedup shows that out-of-order PDES can exploit more parallelism during the simulation and is more efficient than synchronous PDES. This benchmark confirms the increased CPU utilization on a multi-core host by out-of-order PDES.

Embedded application examples

To demonstrate the effectiveness of the PDES approaches for realistic design examples, we use six embedded applications which we have modeled in-house based on reference source code for standard algorithms. We measure the results on the same host PC as in Parallel system-level benchmarks.[3]

JPEG image encoder with parallel color space encoding

The JPEG encoder performs its *DCT*, *Quantization* and *Zigzag* modules for the 3 color components in parallel, followed by a sequential *Huffman* encoder at the end. Table 2 shows the simulation speedup. The size of our input BMP image is 3216 × 2136 pixels. Note that, the model has maximal 3 parallel threads, followed by a significant sequential part.

We simulate this application model at four abstraction levels (specification, architecture mapped, OS scheduled, network linked). As shown in Table 2, simulation speed increases for both parallel simulators but the out-of-order PDES gains more speedup than synchronous PDES.

H.264 video decoder with parallel slice decoding

Our second application is a parallelized video decoder model based on the H.264/AVC standard. An H.264 video frame can be split into multiple independent slices during encoding. Our model uses four parallel slice decoders to decode the separate slices in a frame simultaneously. The H.264 stimulus module reads the slices from the input stream and dispatches them to the four following slice decoders for parallel processing. A synchronizer block at the end completes the decoding of

[3]Compared to the experiments in [11] and [12], the results for the JPEG image encoder and the H.264 video decoder here are based on improved models and have been simulated on a different host with different test streams.

Table 2 Experimental results for embedded application examples using standard algorithms.

Simulator: / Par. Issued Threads:		Single-thread reference n/a		Multi-core Synchronous parallel 24		Multi-core Out-of-Order parallel 24	
		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	2.0	1.7	2.1 / 0.91	1.4 / **1.23**	2.2 / 0.87	0.6 / **2.70**
	arch	2.5	1.7	2.7 / 0.91	1.4 / **1.21**	3.0 / 0.81	0.7 / **2.60**
	sched	2.6	1.7	2.6 / 0.98	1.4 / **1.24**	2.7 / 0.95	0.6 / **2.70**
	net	2.8	2.8	3.2 / 0.86	2.1 / **1.32**	3.2 / 0.87	1.1 / **2.63**
H.264 Decoder	spec	11.3	167.2	11.6 / 0.98	169.0 / **0.99**	11.6 / 0.97	96.9 / **1.73**
	arch	11.8	165.9	11.9 / 0.99	169.5 / **0.98**	12.2 / 0.96	97.3 / **1.71**
	sched	12.0	168.0	12.1 / 0.99	169.4 / **0.99**	12.3 / 0.98	95.7 / **1.76**
	net	12.3	168.0	12.6 / 0.98	169.5 / **0.99**	13.1 / 0.94	94.7 / **1.77**
Video Edge Detection		1.3	60.9	1.7 / 0.75	44.1 / **1.38**	1.8 / 0.71	40.1 / **1.52**
Image Edge Detection		2.5	3.0	2.7 / 0.95	2.3 / **1.28**	2.6 / 0.97	2.4 / **1.26**
H.264 Encoder		18.7	2875.8	20.9 / 0.89	1534.9 / **1.87**	22.9 / 0.81	1452.7 / **1.98**
MP3 Decoder		2.1	4.1	2.1 / 1.00	4.4 / **0.92**	2.3 / 0.92	4.7 / **0.87**

each frame and triggers the stimulus to send the next one. This design model is of industrial-size and consists of about 40*k* lines of code.

We use a test stream of 1079 video frames with 1280 × 720 pixels per frame (approximately 58.6% of the total computation is spent on the slide decoding which has been parallelized). Table 2 shows that synchronous PDES can hardly gain any speedup due to the simulation cycle barriers. Furthermore, protecting the shared resources and added synchronizations introduce simulation overhead for PDES. However, out-of-order PDES still gains significant speed up to a factor of 1.77×. Note that even for a large realistic design, such as this H.264 decoder model, the increased compilation time due to the static model analysis for out-of-order PDES is negligible.

Edge detection with parallel Gaussian smoothing

Our third application example, a Canny edge detector application, calculates edges in images of a video stream. In our model, we have parallelized the most computationally complex function *Gaussian Smooth* (approximately 45% of the total computation) on 4 cores. With a test stream of 100 frames of 1280 × 720 pixels, the simulation results in Table 2 show 1.38 speedup for synchronous PDES and 1.52 speedup for out-of-order PDES.

The fourth example uses the same edge detection algorithm but only detects the edges in a single image. Again we split the *Gaussian Smooth* function equally on 4 parallel modules, but use a larger image. For the test image with 3245 × 2500 pixels, PDES accelerates the simulation with an average speedup of 1.27. The workload is evenly distributed so it fully fills the simulation cycles of the mapped parallel threads. Thus, out-of-order PDES loses its advantage and performs slightly slower than synchronous PDES due to the out-of-order scheduling overhead.

H.264 video encoder with parallel motion search

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

In our model, multiple motion search units are processing in parallel so that the comparison between the current image and multiple reference frames can be performed simultaneously. Our test stream is a video of 95 frames with 176 × 144 pixels per frame, and the number of B-slices between every I-slice or P-slice is 4. That is, among every 5 consecutive frames 4 frames need bidirectional inter-frame prediction. Table 2 shows a similar simulation acceleration with a speedup of 1.87 for synchronous PDES, and 1.98 for out-of-order PDES.

MP3 stereo audio decoder

The last application, a MP3 player, is another example for which the performance of PDES is marginal due to the limited parallelism in the model. Our MP3 audio decoder is modeled with parallel decoding for stereo channels. Our test stream is a 99.6 Kbps, 44.1 Hz joint stereo MP3 file with 2372 frames. It takes less than 5 seconds to simulate, but there are 7114 context switches in scheduling the two parallel threads. Here, both PDES approaches take longer time than the traditional DE simulation due to the low computation workload and the then significant overhead for synchronization.

Overall, we can see that the 24 available parallel cores on the server are under-utilized for all six applications, and by both parallel simulators. The reason is clearly the limited available parallelism in the models.

Parallel Discrete Event Simulation carries the promise to exploit the explicit parallelism in an ESL design model by utilizing the parallel computing resources on a multi-core simulation host. Synchronous PDES parallelizes the threads in the same simulation cycles. In contrast, advanced out-of-order PDES aggressively breaks the simulation cycle barrier and allows threads in different cycles to run in parallel for the small cost of increased compile time for static dependency analysis. Both PDES approaches fully retain the SLDL simulation semantics and result in standard-compliant simulation with accurate timing. Moreover, both significantly reduce the simulator run time. In most cases, out-of-order PDES proves to be the winner which gains the highest speedup with only a small increase of compilation time.

Overall, PDES is highly desirable for ESL design due to the constantly rising complexity of embedded systems which requires accurate and fast simulation. Given the need for higher simulation speeds and the demonstrated potential of parallel simulation, it becomes clear that PDES is and will be an area of active research.

Future work includes further improvements in dependency analysis for both conservative and optimistic PDES techniques in order to exploit more parallelism, and research on model design suitable for faster execution on parallel architectures. However, all efforts in simulator design are limited by the amount of exposed parallelism in the application. How to expose thread-level parallelism in software applications remains as a Grand Challenge. ■

Acknowledgments

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.The authors also thank the reviewers and editors for valuable suggestions to improve this article.

■ References

[1] R. Fujimoto, ''Parallel discrete event simulation,'' *Communications of the ACM*, vol. 33, pp. 30–53, Oct. 1990.

[2] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele, ''Scalably distributed SystemC simulation for embedded applications,'' in *International Symposium on Industrial Embedded Systems, 2008,* Jun. 2008, pp. 271–274.

[3] K. Chandy and J. Misra, ''Distributed simulation: A case study in design and verification of distributed programs,'' *IEEE Trans. Software Engineering*, vol. SE-5, pp. 440–452, Sep. 1979.

[4] S. Sirowy, C. Huang, and F. Vahid, ''Online SystemC emulation acceleration,'' in *Proceedings of the Design Automation Conference (DAC)*, 2010.

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

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

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

[8] D. Yun, S. Kim, and S. Ha, ''A parallel simulation technique for multicore embedded systems and its performance analysis,'' *IEEE Trans. Computer-Aided*

Design of Integrated Circuits and Systems (TCAD), vol. 31, pp. 121–131, Jan. 2012.

[9] 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*, 2010, pp. 241–246.

[10] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, ''Parallelizing SystemC kernel for fast hardware simulation on SMP machines,'' in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009, pp. 80–87.

[11] R. Dömer, W. Chen, and X. Han, ''Parallel discrete event simulation of transaction level models,'' in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[12] W. Chen, X. Han, and R. Dömer, ''Out-of-order parallel simulation for ESL design,'' in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2012.

Weiwei Chen is a PhD candidate in the Electrical Engineering and Computer Science Department at the University of California, Irvine, where she is also affiliated with the Center for Embedded Computer Systems (CECS). Her research interests include system-level design and validation, and execution semantics of system-level description languages. She has an MS in computer science and engineering from Shanghai Jiao Tong University, Shanghai, China.

Xu Han is a PhD candidate in the Electrical Engineering and Computer Science Department at the University of California, Irvine, where he is also affiliated with the CECS. His research interests include system-level modeling and recoding of embedded systems. He has an MS in electrical engineering from the Royal Institute of Technology, Sweden.

Che-Wei Chang is a PhD candidate in the Electrical Engineering and Computer Science Department at the University of California, Irvine, where he is also affiliated with the CECS. His research interests include system-level modeling and formal verification. He has an MS in electrical engineering from Cheng Kung University, Tainan, Taiwan, and an MS in computer science and engineering from the University of California, Irvine.

Rainer Dömer is an associate professor in electrical engineering and computer science at the University of California, Irvine, where he is also a member of the CECS. His research interests include system-level design and methodologies, embedded computer systems, specification and modeling languages, system-on-chip design, and embedded hard- and software systems. He has a PhD in information and computer science from the University of Dortmund, Germany.

■ Direct questions and comments about this article to Weiwei Chen, Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA; weiwei.chen@uci.edu.