Parallel Discrete Event Simulation of Transaction Level Models

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

Abstract— Describing Multi-Processor Systems-on-Chip (MPSoC) at the abstract Electronic System Level (ESL) is one task, validating them efficiently is another. Here, fast and accurate system-level simulation is critical. Recently, Parallel Discrete Event Simulation (PDES) has gained significant attraction again as it promises to utilize the existing parallelism in today's multi-core CPU hosts. This paper discusses the parallel simulation of Transaction-Level Models (TLMs) described in System-Level Description Languages (SLDLs), such as SystemC and SpecC. We review how PDES exploits the explicit parallelism in the ESL design models and uses the parallel processing units available on multi-core host PCs to significantly reduce the simulation time. We show experimental results for two highly parallel benchmarks as well as for two actual embedded applications.

I. Introduction

Describing complex Multi-Processor Systems-on-Chip (MPSoC) at the abstract Electronic System Level (ESL) is one difficult task. However, getting the models to work correctly is another. The large size and great complexity of modern embedded systems with their heterogeneous components, complex interconnects, and sophisticated functionality pose enormous challenges to system validation and debugging. Accurate yet fast ESL simulation is a key to enabling effective and efficient design validation and implementation.

In this paper, we review recent approaches for simulating MPSoC designs described at the system or transaction level. We focus in particular on parallel simulation techniques. The well-known approach of Parallel Discrete Event Simulation (PDES) has recently gained a lot of attraction again due to the inexpensive availability of parallel processing capabilities in today's multi-core CPU hosts. PDES holds the promise to map the explicit parallelism described in Transaction-Level Models (TLMs) efficiently onto the parallel cores available on the simulation host. As such, it can exploit the available parallelism and significantly reduce the simulation time.

A. Parallel execution in SystemC and SpecC

In this paper, we discuss the parallel simulation of TLMs described in the System-Level Description Languages (SLDLs) SystemC [7] and SpecC [6]. We note that, in contrast to flat and sequential C/C++ programming code, both languages *explicitly* specify the key ESL concepts in the design model, including behavioral and structural hierarchy, potential for parallelism and pipelining, communication channels, and constraints on timing. Having these intrinsic features of the application explicit in the model enables efficient design space exploration and automatic refinement by computer-aided design (CAD) tools.

While both SystemC and SpecC model parallel execution explicitly, there is a significant difference in the parallel execution semantics between the languages. Both SLDLs define their execution semantics by use of DE-based scheduling of multiple concurrent threads which are managed and coordinated by a central simulation kernel. However, SystemC requires *cooperative* multi-threading, whereas SpecC allows *preemptive* multi-threading [4].

SystemC semantics guarantee the uninterrupted execution of threads (including their accesses to any shared variables) which makes it hard to implement a truly parallel simulator. In short, complex inter-dependency analysis over all variables in the system model is a prerequisite to parallel multi-core simulation in SystemC.

In contrast to the cooperative scheduling mandated for SystemC models, multi-threading in SpecC is explicitly defined as preemptive. This makes a parallel simulation significantly easier because only shared variables in channels (which act as monitors) need to be protected for mutually exclusive access. Such protection can easily be inserted automatically into the model [2].

B. Related Work

Most ESL design frameworks today still rely on regular Discrete Event (DE) simulators which issue only a single thread at any time to avoid complex synchronization of the concurrent threads. As such, the simulator kernel becomes an obstacle in improving simulation performance on multi-core host machines [8].

A well-studied solution to this problem is Parallel Discrete Event Simulation (PDES) [1, 5, 11]. [9, 10] discuss the PDES on Hardware Description Languages, VHDL and Verilog. To apply PDES solutions to today's SLDLs and actually allow parallel execution on multi-core processors, the simulator kernel needs to be modified to issue and properly synchronize multiple OS kernel threads in each scheduling step. [3] and [12] have extended the SystemC simulator kernel accordingly. Clusters with single-core nodes are targeted in [3] which uses multiple schedulers on different processing nodes and defines a master node for time synchronization. A parallelized SystemC kernel for fast simulation on SMP machines is presented in [12] which issues multiple runable OS kernel threads in each simulation cycle. The SpecC-based scheduling approach described in [2] is very similar. However, it features a detailed synchronization protection mechanism which automatically instruments any user-defined and hierarchical channels. As such, this approach does not need to work around the cooperative SystemC execution semantics, neither does it require a specially prepared channel library.

II. System-level Discrete Event Simulation

System design models with explicitly specified parallelism carry the promise of increased simulation performance through parallel execution of the threads on the available hardware resources of a multi-core host.

In this section, we will first review the DE scheduling scheme used in traditional SLDL simulators which issues only a single thread at any time. We will then discuss the extended scheduling algorithm with true multi-threading capability on symmetric multiprocessing (multi-core) machines. With the exception of the different requirements for protection of communication and synchronization between the concurrent threads, as outlined in Section I.A above, the following sections apply equally to both SystemC and SpecC SLDLs.

A. Traditional Discrete Event Simulation

In traditional system-level simulation, a regular DE simulator is used. Threads are created for the explicit parallelism described in the models (e.g. *par*{} and *pipe*{} statements in SpecC, and *SC_THREADS* in SystemC). The threads are generally managed in the scheduler by use of queues, such as **READY**, which contains all threads that ready to execute, and **WAIT**, which contains all waiting threads. When running, the threads communicate via events and shared variables or channels, and advance simulation time using *wait-for-time* constructs.

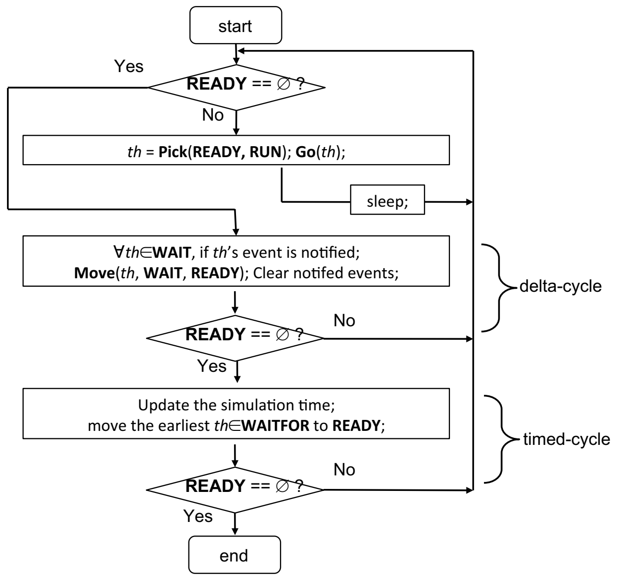
start
Yes
READY == ∅ ?
No
th = Pick(READY, RUN); Go(th);
sleep;
∀th∈WAIT, if th's event is notified;
Move(th, WAIT, READY); Clear notifed events;
delta-cycle
No
READY == ∅ ?
Yes
Update the simulation time;
move the earliest th∈WAITFOR to READY;
timed-cycle
No
READY == ∅ ?
Yes
end

Fig. 1.: Traditional DE scheduler.

Traditional DE simulation is driven by events and simulation time advances. Whenever events are delivered or time increases, the scheduler is called to move the simulation forward. As shown in Fig. 1, at any time the traditional scheduler runs a single thread which is picked from the **READY** queue. Within a *delta-cycle*, the choice of the next thread to run is non-deterministic (by definition). If the **READY** queue is empty, the scheduler will fill the queue again by waking threads who have received events they were waiting for. These are taken out of the **WAIT** queue and a new delta-cycle begins.

If the **READY** queue is still empty after event delivery, the scheduler advances the simulation time, moves all threads with the earliest timestamp from the **WAITFOR** queue into the **READY** queue, and resumes execution (*timed-cycle*). Note that at any time, there is only one thread actively executing in the traditional simulation.

B. Parallel Discrete Event Simulation

A parallel DE scheduler basically works the same way as the traditional scheduler, with one exception: in each cycle, it picks multiple threads from the **READY** queue and runs them in parallel on the available processor cores. For SLDL simulators in particular, the parallel scheduler picks and runs as many threads as CPU cores are available[1]. In other words, it keeps all parallel processing resources as busy as possible.

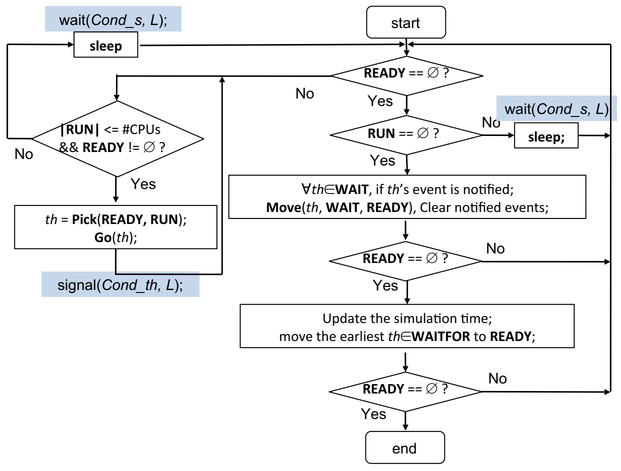

Fig. 2.: Parallel DE scheduler.

Fig. 2 shows the extended control flow of the parallel scheduler. Note the added loop at the left which issues running threads as long as CPU cores are available and the **READY** queue still has candidates.

Note that for the traditional sequential scheduler any threading model (including user-level or kernel-level threads) is acceptable since only one thread is running at any time. For the parallel scheduler to be effective on a multi-core platform, in contrast, OS kernel threads are necessary. Here, the underlying operating system needs to be aware of the multiple simulator threads so that it can utilize the available parallel CPU cores.

[1]Scheduling more threads to run than cores are available would be possible, but this would hurt the simulation performance due to unnecessary preemptive scheduling by the underlying operating system.

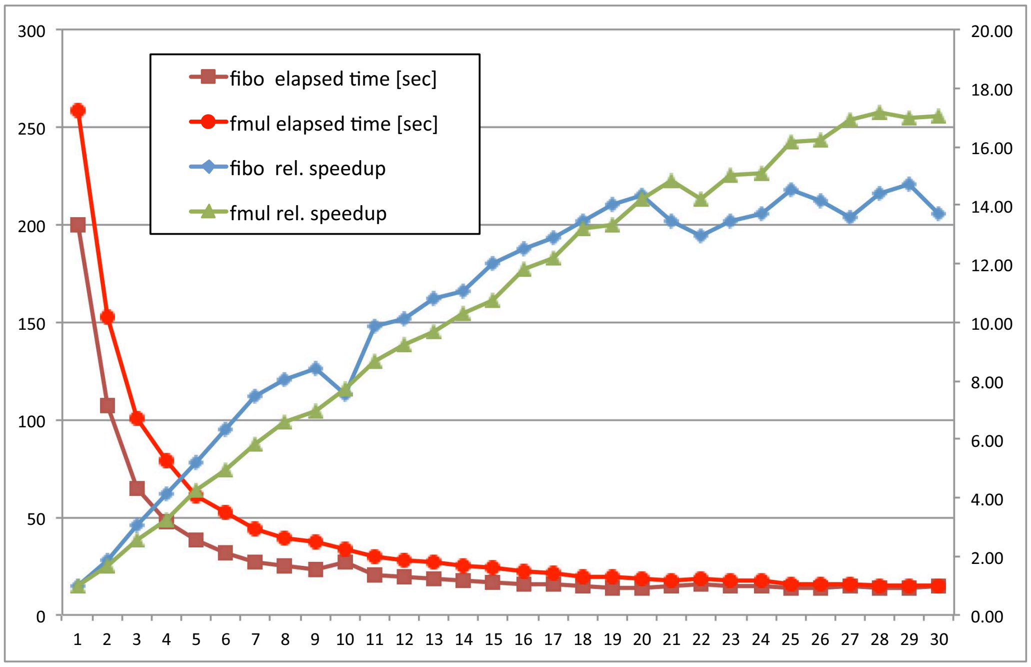

Fig. 3.: Simulation results for highly parallel benchmark designs.

III. Parallel System-Level Benchmarks

To demonstrate the potential of parallel simulation, we have designed two highly parallel benchmark design models, a parallel floating-point multiplication example and a parallel recursive Fibonacci calculator. Both benchmark designs are system-level models specified in SpecC SLDL.

For all our experiments, we use a symmetric multiprocessing (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[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.

A. 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 Fig. 3 shows the experimental results for our parallel 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 Fig. 2) between 1 and 30.

The elapsed simulator run times are shown in the red line with circle points. Clearly, with more and more CPU cores used, the elapsed simulation time decreases in near logarithmic fashion and flattens when no more CPU cores become available (> 24).

When plotting the relative speedup, as shown in the green line with triangle points, one can see that, as expected, the simulation speed increases in nearly linear manner the more parallel cores are used. The maximal speedup is about 17x for this example on our 24-core server.

B. Parallel Fibonacci calculation

Our second parallel benchmark **fibo** calculates the Fibonacci series in parallel and recursive 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 8), 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 also plotted in Fig. 3. The elapsed simulator run times, shown in the dark red line with square points, show the same decreasing curve as the **fmul** example. Accordingly, the relative simulation speedup, shown in the blue line with diamond points, also shows the same ex-

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

pected behavior. Speed increases in nearly linear fashion until it reaches saturation at about a factor of 14x.

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.

IV. Experimental Results for Embedded Applications

To demonstrate the improved simulation speed of the parallel multi-core simulator for actual embedded applications, we use a H.264 video decoder and a JPEG encoder application [2].

A. H.264 AVC Decoder with Parallel Slice Decoding

The H.264 Advanced Video Coding (AVC) standard [13] is widely used in video applications, such as internet streaming, disc storage, and television services. It provides high-quality video at less than half the bit rate compared to its predecessors H.263 and H.262. At the same time, it requires more computing resources for both video encoding and decoding.

The H.264 decoder takes as input a stream of encoded video frames. A frame can be further split into one or more slices, as illustrated in the upper right part of Fig. 4. Notably, slices are *independent* of each other in the sense that decoding one slice will not require any data from the other slices. For this reason, parallelism exists at the slice-level and parallel slice decoders can be used to decode multiple slices in a frame simultaneously.

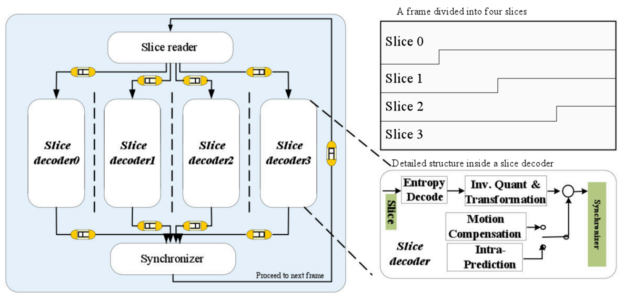

Fig. 4.: Parallelized H.264 decoder model.

Fig. 4 shows the block diagram of our H.264 decoder model. The decoding of a frame begins with reading new slices from the input stream. These are then dispatched into four parallel slice decoders. Finally, a synchronizer block completes the decoding by applying a deblocking filter to the decoded frame. All the blocks communicate via FIFO channels. Internally, each slice decoder consists of the regular H.264 decoder functions, such as entropy decoding, inverse quantization and transformation, motion compensation, and intra-prediction.

For our experiment, we use the stream ”Harbour” with 299 video frames, each with 4 slices of equal size. As discussed in [2], 68.4% of the total computation time is spent in the parallel slice decoding. Thus, while the maximum parallelism is 4 in this model, the effective maximum speedup we can gain is only 2.05 (due to the sequential parts).

Table 5 lists the simulator run times for TLMs at different abstraction levels. We compare the elapsed simulation time against the sequential reference simulator (the table also includes the CPU load reported by the Linux OS). We can clearly see that the 24 available cores on the server are under-utilized. The reason is obviouly the maximum available parallelism in the model (2.05). The measured speedups are also somewhat lower than the maximum due to the overhead introduced in parallelizing and synchronizing the slice decoders.

Simulator		Reference	Multi-Core	
Par. issued threads:		n/a	24	
		sim. time	sim. time	speedup
models	spec	37.71s (90%)	21.01s (173%)	1.79
	arch	38.26s (90%)	21.26s (171%)	1.80
	sched	38.10s (91%)	22.31s (166%)	1.71
	net	38.18s (91%)	22.45s (166%)	1.70
	tlm	39.17s (90%)	22.32s (167%)	1.75
	comm	49.92s (90%)	33.55s (156%)	1.49

Fig. 5.: Simulation results for H.264 Decoder example.

Simulator		Reference	Multi-Core	
Par. issued threads:		n/a	24	
		sim. time	sim. time	speedup
models	spec	3.46s (93%)	2.82s (146%)	1.23
	arch	3.92s (94%)	2.83s (147%)	1.39
	sched	4.64s (93%)	3.61s (137%)	1.29
	net	13.87s (97%)	45.96s (130%)	0.30

Fig. 6.: Simulation results for JPEG Encoder example.

B. JPEG Image Encoder

As a second embedded application, Table 6 shows the simulation speedup for a JPEG Encoder example which performs its DCT, $Quantization$ and $Zigzag$ modules for the 3 color components in parallel, followed by a sequential $Huffman$ encoder at the end. Again, the available parallelism in the model is low (maximal 3 parallel threads, followed by a significant sequential part). Some speedup is gained by the multi-core parallel simulator for the higher level models ($spec$, $arch$, $sched$). However, the simulator performance decreases for the model at the lowest abstraction level (net) due to the high number of bus transactions and arbitrations which are not parallelized.

V. Concluding Remarks

Parallel Discrete Event Simulation (PDES) has recently come into fashion again. PDES is highly desirable for ESL design due to the constantly rising complexity of embedded systems which makes accurate and fast simulation challenging.

After a brief review of the essential differences between traditional discrete event simulation and PDES, we have examined two highly parallel benchmark applications, **fmul** and **fibo**, and two actual embedded design examples, a H.264 decoder and a JPEG encoder. While the measured simulation times of the parallel **fmul** and **fibo** benchmarks promise a linear increase in simulation speed due to their excellent scalability, the speedup achieved for the two embedded applications is quite limited. Nevertheless, the measured speedup is significant (nearly 2x!) and can make a real difference in shortening the validation time of such systems.

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. Notably, the basic PDES algorithm discussed in Section B is very straightforward and advanced techniques, including conservative and optimistic approaches, can further improve the simulation speed by optimizing the scheduling around synchronization barriers. However, all efforts are limited by the amount of exposed parallelism in the application.

We can conclude that the parallelism available in the application, which in TLMs is explicitly specified by SLDL constructs, can be exploited by parallel simulators and then results in effective reduction of simulator run time. However, the Grand Challenge remains, the problem of how to efficiently parallelize applications.

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.

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. Multiprocessor SoC Platforms: A Component-Based Design Approach. *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 V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (4)*, volume 3994 of *Lecture Notes in Computer Science*, pages 653–660. Springer, 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)*, Yokohama, Japan, January 2011.

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

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

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

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

[9] T. Li, Y. Guo, and S.-K. Li. Design and Implementation of a Parallel Verilog Simulator: PVSim. *VLSI Design, International Conference on*, pages 329–334, 2004.

[10] E. Naroska. Parallel VHDL simulation. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 159–165, 1998.

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

[12] 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, Washington, DC, USA, 2009. IEEE Computer Society.

[13] 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.