Multi-Core Parallel Simulation of System-Level Description Languages

Rainer Dömer, Weiwei Chen, Xu Han

Center for Embedded Computer Systems
University of California, Irvine, USA
doemer@uci.edu, weiweic@uci.edu, hanx@uci.edu

Andreas Gerstlauer

Dept. of Electrical and Computer Engineering
University of Texas at Austin, USA
gerstl@ece.utexas.edu

Abstract— The validation of transaction level models described in System-level Description Languages (SLDLs) often relies on extensive simulation. However, traditional Discrete Event (DE) simulation of SLDLs is cooperative and cannot utilize the available parallelism in modern multi-core CPU hosts. In this work, we study the SLDL execution semantics of concurrent threads and present a multi-core parallel simulation approach which automatically protects communication between concurrent threads so that parallel simulation on multi-core hosts becomes possible. We demonstrate significant speed-up in simulation time of several system models, including a H.264 video decoder and a JPEG encoder.

I. Introduction

Modern embedded system platforms often integrate multiple processing elements into the system, including general-purpose CPUs, application-specific and digital signal processors, as well as dedicated hardware accelerators. The large size and complexity of these systems pose great challenges to design and validation using traditional design flows. System designers are forced to move to higher levels of abstraction to address these challenges, including large number of heterogeneous components, complex interconnect, sophisticated functionality, and slow simulation.

At the so-called Electronic System Level (ESL), system design and validation aim at a systematic top-down design methodology which successively transforms a given high-level specification model into a detailed implementation. As one example, the System-on-Chip Environment (SCE) is a refinement-based framework for heterogeneous MPSoC design [7]. SCE starts with a system specification model described in the SpecC [10] System-Level Description Language (SLDL) and implements a top-down ESL design flow based on the specify-explore-refine methodology.

Note that in contrast to flat and sequential C/C++ programming code, a well-defined SLDL design model *explicitly* specifies any potential for thread-level parallelism, among other key concepts. In this work, we exploit the existing explicit parallelism in SLDL design models to speed-up their simulation.

Regular SLDL model validation is based on traditional discrete event (DE) simulation. The SLDL simulator implements the existing parallelism in the design model in the form of concurrent user-level threads within a single process. The multi-threading model used is cooperative (i.e. non-preemptive), which greatly simplifies communication through events and variables in shared memory. Unfortunately, however, this threading model cannot utilize any available parallelism in multi-core host CPUs which nowadays are common and readily available in regular PCs. In Section III, we extend the SpecC simulator for parallel multi-core execution which leads to significant reduction of simulation time.

A. Related Work

Regular DE-based SLDL simulators issue only a single thread of execution at any time to avoid complex synchronization of the concurrent threads. This way, however, the single-threaded simulator kernel is an impediment in improving simulation performance on multi-core host machines [11].

A well-studied solution to this problem is Parallel Discrete Event Simulation (PDES) [2, 8, 14]. 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. [5, 15, 16] discuss extensions to the SystemC SLDL. Clusters with single-core nodes are targeted in [5] 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 [15] and [16] which issues multiple runable OS kernel threads in each simulation cycle.

In comparison, our work [3, 4] targets the SpecC SLDL. Moreover, our approach features a detailed synchronization protection mechanism which we automatically generate for any user-defined and hierarchical channels. Also, instead of synthetic benchmarks, we provide results for an actual H.264 video decoder and a JPEG encoder.

II. SLDL Multi-threading Semantics

Both SystemC and SpecC 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. More specifically, both the SystemC and SpecC reference simulators that are freely available from the corresponding consortia use *cooperative* multi-threading in their schedulers. That is, both reference schedulers select only a single thread to run at all times.

However, the reference simulator implementations do not define the actual language semantics. In fact, the execution semantics of concurrent threads defined by the SystemC Language Reference Manual (LRM) differ significantly from the semantics defined in the SpecC LRM.

A. Cooperative multi-threading in SystemC

The SystemC LRM [12] clearly states (in Section 4.2.1.2) that *"process instances execute without interruption"*, which is known as cooperative (or co-routine) multitasking. In other words, preemptive scheduling is explicitly forbidden.

As a consequence, when writing a SystemC model, the system designer *"can assume that a method process will execute in its entirety without interruption"* [12]. This is convenient when sharing variables among different threads (because mutual exclusive access to such variables in a critical region is implicitly guaranteed by the non-preemptive scheduling semantics). While this can aid significantly in avoiding bugs and race conditions if one is only concerned with modeling and simulation, such semantics are hard to verify and synthesize if the goal is to eventually reach an efficient implementation. Furthermore, sharing of plain variables also violates the overall system-level principle of separation of concerns where computation and communication are supposed to be clearly separated in modules and channels, respectively.

The uninterrupted execution guaranteed by the SystemC LRM makes it hard to synthesize concurrent threads into a truly parallel implementation. This same problem also prevents an efficient implementation of a fully standards-compliant parallel multi-core simulator, which we are aiming for in this paper. This particular problem of parallel simulation is actually addressed specifically in the SystemC LRM [12], as follows:

> *"An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the co-routine semantics defined [...]. In other words, the implementation would be obliged to analyze any dependencies between processes and constrain their execution to match the co-routine semantics."*

In short, complex inter-dependency analysis over all variables in the system model is a prerequisite to parallel multi-core simulation in SystemC.

B. Preemptive multi-threading in SpecC

In contrast to the cooperative scheduling mandated for SystemC models, multi-threading in the SpecC LRM [6] is (in Section 3.3) explicitly stated as *"preemptive execution"*. Thus, *"No atomicity is guaranteed for the execution of any portion of concurrent code"*. Consequently, shared variables need to be carefully protected for mutually exclusive access in critical regions.

Allowing preemptive execution of concurrent threads requires the system designer to plan and design the system model carefully with respect to all communication (in particular through shared variables!) and synchronization. However, exactly that is the main premise of clear separation of computation and communication in system-level models (which enables the reuse of existing components, such as IP).

In other words, if communication and computation are already separated in a well-written system model, this also provides the framework for solving the critical section problem in the a preemptive multi-threading environment. The SpecC Language Working Group realized this opportunity when defining the version 2.0 of the language in 2001 [9]. In fact, a new "time interval formalism" was developed that precisely defines the parallel execution semantics of concurrent threads in SpecC. Specifically, truly parallel execution (with preemption) is generally assumed in SpecC.

To then allow safe communication and synchronization, a single exception was defined for channels. The SpecC LRM [6] states (in Section 2.3.2(j)) that

> *"For each instance of a channel, the channel methods are mutually exclusive in their execution. Implicitly, each channel instance has a mutex associated with it that the calling thread acquires before and releases after the execution of any method of the channel instance."*

In other words, each SpecC channel instance implicitly acts as a *monitor* that automatically protects the shared variables for mutually exclusive access in the critical region of communication.

Note that the SpecC semantics based on separation of computation and communication elegantly solve the problem of allowing truly parallel execution of computational parts in the design, as well as provide built-in protection of critical regions in channels for safe communication. This builds the basis for both synthesis of efficient concurrent hardware/software realizations as well as implementation of efficient parallel simulators. While the freely available SpecC reference simulator does not utilize this possibility, we exploit these semantics in our parallel multi-core simulator.

III. Multi-Core Parallel Simulation

Design models with explicitly specified parallelism make it promising to increase simulation performance by parallel execution on the available hardware resources of a multi-core host. However, care must be taken to properly synchronize the concurrent threads.

In this section[1], we will first review the scheduling scheme in a traditional simulation kernel that issues only a single thread at a time. We will then present our improved scheduling algorithm with true multi-threading capability on symmetric multiprocessing (multi-core) machines and discuss the necessary synchronization mechanisms for safe parallel execution.

A. Traditional Discrete Event Simulation

In both SystemC and SpecC SLDLs, a traditional 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_METHODS* and *SC_THREADS* in SystemC). These threads communicate via events and advance simulation time using *wait-for-time* constructs.

[1] Without loss of generality, we assume use the SpecC SLDL execution semantics here. Please refer to Section II.A for a discussion on SystemC SLDL.

To describe the simulation algorithm, we define the following data structures and operations:

1. Definition of queues of threads th in the simulator:
   - **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**, **COMPLETE**}
   - **READY** = $\{th \mid th$ is ready to run$\}$
   - **RUN** = $\{th \mid th$ is currently running$\}$
   - **WAIT** = $\{th \mid th$ is waiting for some events$\}$
   - **WAITFOR** = $\{th \mid th$ is waiting for time advance$\}$
   - **COMPLETE** = $\{th \mid th$ has completed its execution$\}$
2. Simulation invariants:
   Let **THREADS** = set of all threads which currently exist. Then, at any time, the following conditions hold:
   - **THREADS** = **READY** ∪ **RUN** ∪ **WAIT** ∪ **WAITFOR** ∪ **COMPLETE**.
   - ∀ A, B ∈ **QUEUES**, A ≠ B : A ∩ B = ∅.
3. Operations on threads th:
   - **Go**(th): let thread th acquire a CPU and begin execution.
   - **Stop**(th): stop execution of thread th and release the CPU.
   - **Switch**(th_1, th_2): switch the CPU from the execution of thread th_1 to thread th_2.
4. Operations on threads with set manipulations:
   Suppose th is a thread in one of the queues, A and B are queues ∈ **QUEUES**.
   - th = **Create**(): create a new thread th and put it in set **READY**.
   - **Delete**(th): kill thread th and remove it from set **COMPLETE**.
   - th = **Pick**(A, B): pick one thread th from set A and put it into set B.
   - **Move**(th, A, B): move thread th from set A to B.
5. Initial state at beginning of simulation:
   - **THREADS** = $\{th_{root}\}$.
   - **RUN** = $\{th_{root}\}$.
   - **READY** = **WAIT** = **WAITFOR** = **COMPLETE** = ∅.
   - $time$ = 0.

SLDL 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.

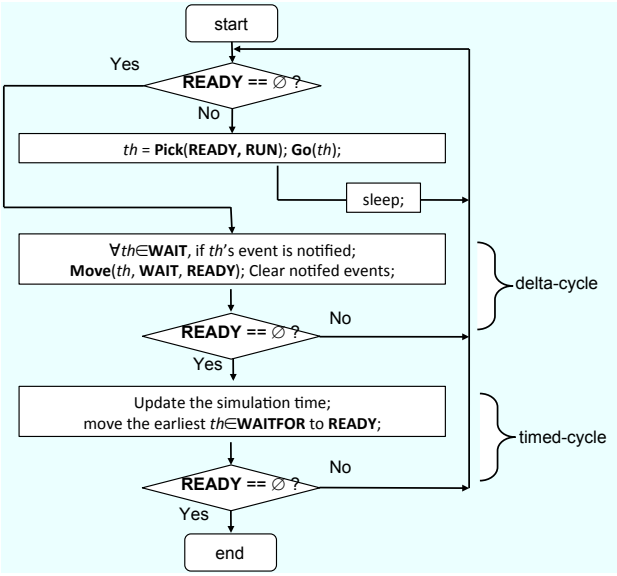

Fig. 1. Traditional SLDL scheduler.

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. At any time, there is only one thread actively executing in the traditional simulation.

B. Multi-Core Discrete Event Simulation

The scheduler for multi-core parallel simulation works the same way as the traditional scheduler, with one exception: in each cycle, it picks multiple OS kernel threads from the **READY** queue and runs them in parallel on the available cores. In particular, it fills the **RUN** set with multiple threads up to the number of CPU cores available. In other words, it keeps as many cores as busy as possible.

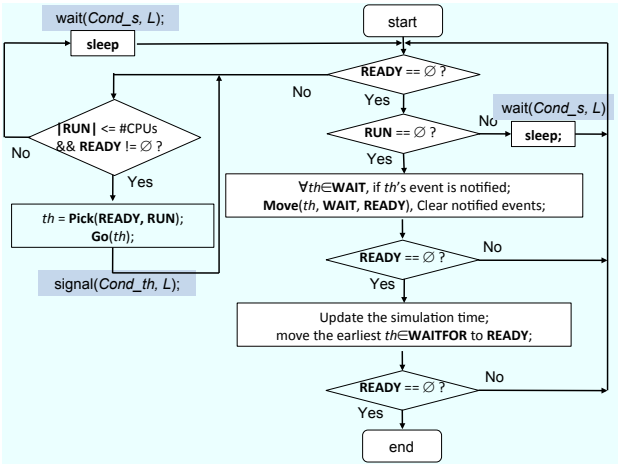

Fig. 2. Multi-core SLDL scheduler.

Fig. 2 shows the extended control flow of the multi-core scheduler. Note the extra loop at the left which issues OS ker-

nel threads as long as CPU cores are available and the **READY** queue is not empty.

C. Synchronization for Multi-Core Simulation

The benefit of running more than a single thread at the same time comes at a price. Explicit synchronization becomes necessary. In particular, shared data structures in the simulation engine, including the thread queues and event lists, and shared variables in communication channels of the application model need to be properly protected by locks for mutual exclusive access by the concurrent threads.

C.1 Protecting Scheduling Resources

To protect all central scheduling resources[2], we run the scheduler in its own thread and introduce locks and condition variables for proper synchronization. More specifically, we use

- one central lock L to protect the scheduling resources,
- a condition variable *Cond_s* for the scheduler, and
- a condition variable *Cond_th* for each working thread.

When a working thread executes a *wait* or *waitfor* instruction, we switch execution to the scheduling thread by waking the scheduler (signal(*Cond_s*) and putting the working thread to sleep (wait(*Cond_th, L*). The scheduler then uses the same mechanism to resume the next working thread.

C.2 Protecting Communication

Communication between threads also needs to be explicitly protected[3]. As discussed in Section II.B, SpecC channels are defined to act as monitors. That is, only one thread at a time may execute code wrapped in a specific channel instance.

```
1  send(d)                          receive(d)
   {                                {
3    Lock(this->Lock);                Lock(this->Lock);
     while(n >= size){                while(!n){
5      ws ++;                           wr ++;
       wait(eSend);                     wait(eRecv);
7      ws --;                           wr --;
     }                                }
9    buffer.store(d);                 buffer.load(d);
     if(wr){                          if(ws){
11     notify(eRecv);                   notify(eSend);
     }                                }
13   unLock(this->Lock);              unLock(this->Lock);
   }                                }
```

Fig. 3. Queue channel implementation for multi-core simulation.

To ensure this, we introduce a lock *ch→Lock* for each channel instance which is acquired at entry and released upon leaving any method of the channel. Fig. 3 shows this for the example of a simple circular buffer with fixed size.

The combination of a central scheduling lock and individual locks for channel and signal instances with proper locking scheme and well-defined ordering ensures safe synchronization among many parallel working threads. Fig. 4 summarizes the detailed use of all locks and the thread switching mechanism for the life-cycle of a working thread. Together with the scheduler presented in Fig. 2, this flow chart defines our multi-core DE simulator and also shows its design considerations.

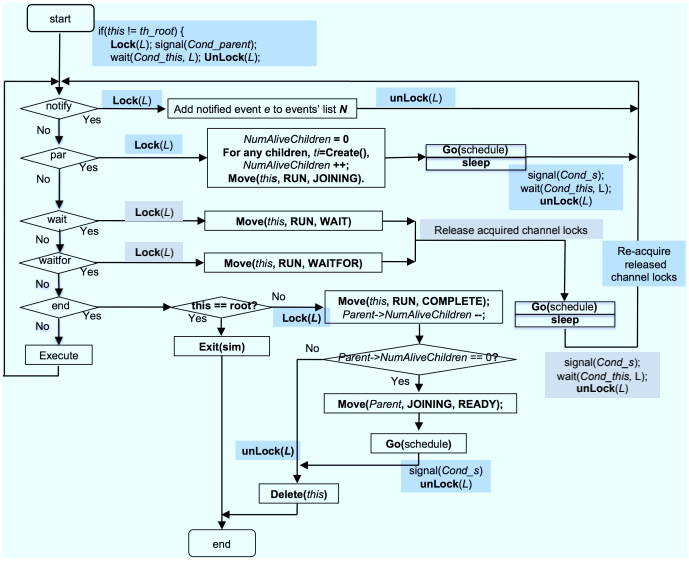

Fig. 4. Life-cycle of a thread in the multi-core simulator.

IV. Experiments and Results

To demonstrate the improved simulation time of our multi-core simulator, we use a H.264 video decoder and a JPEG encoder application.

A. Case Study on a H.264 Video Decoder

The H.264 AVC standard [17] is widely used in video applications, such as internet streaming, disc storage, and television services. H.264 AVC 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. In order to implement the standard on resource-limited embedded systems, it is highly desirable to exploit available parallelism in its algorithm.

The H.264 decoder takes as input a video stream consisting of a sequence of encoded video frames. A frame can be further split into one or more slices during H.264 encoding, as illustrated in the upper right part of Fig. 5. Notably, slices are *independent* of each other in the sense that decoding one slice will not require any data from the other slices (though it may need data from previously decoded reference frames). For this reason, parallelism exists at the slice-level and parallel slice decoders can be used to decode multiple slices in a frame simultaneously.

We have specified a H.264 decoder model based on the H.264/AVC JM reference software [13]. In the reference code, a global data structure (*img*) is used to store the input stream

[2]Protection of central scheduling resources is equally applicable to SpecC and SystemC SLDLs.

[3]Communication protection is efficiently implementable only in SpecC SLDL (see Section II.B).

and all intermediate data during decoding. In order to parallelize the slice decoding, we have duplicated this data structure and other global variables so that each slice decoder has its own copy of input stream data and can decode its own slice locally. As an exception, the output of each slice decoder is still written to a global data structure (*dec_picture*). This is valid because the macro-blocks produced by different slice decoders do not overlap.

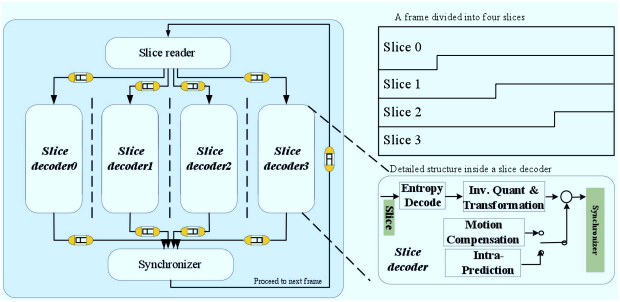

Fig. 5. Parallelized H.264 decoder model.

Fig. 5 shows the block diagram of our 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.

Using SCE, we partition the above H.264 decoder model as follows: the four slice decoders are mapped onto four custom hardware units; the synchronizer is mapped onto an ARM7TDMI processor at 100MHz which also implements the overall control tasks and cooperation with the surrounding testbench. We choose Round-Robin scheduling for tasks in the processor and allocate an AMBA AHB for communication between the processor and the hardware units.

For our H.264 experiment, we use the same stream "Harbour" of 299 video frames, each with 4 slices of equal size. As shown in [3], 68.4% of the total computation time is spent in the slice decoding, which we have parallelized in our decoder model.

As a reference point, we calculate the maximum possible performance gain as follows:

$$MaxSpeedup = \frac{1}{\frac{ParallelPart}{NumOfCores} + SerialPart}$$

For 4 parallel cores, the maximum speedup is

$$MaxSpeedup_4 = \frac{1}{\frac{0.684}{4} + (1 - 0.684)} = 2.05$$

The maximum speedup for 2 cores is accordingly $MaxSpeedup_2 = 1.52$.

Table I lists the simulation results for several design models generated with SCE at different levels of abstraction when using our multi-core simulator on a Fedora core 12 host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at 3.0 GHz, compiled with optimization (-O2) enabled. We compare the elapsed simulation time against the single-core reference simulator (the table also includes the CPU load reported by the Linux OS). Although simulation performances decrease when issuing only one parallel thread due to additional mutexes for safe synchronization in each channel and the scheduler, our multi-core parallel simulation is very effective in reducing the simulation time for all the models when multiple cores in the simulation host are used.

Table I also lists the measured speedup and the maximum theoretical speedup for the models. The more threads are issued in each scheduling step, the more speedup we gain. However, the measured speedups are somewhat lower than the theoretical maximum, which is reasonable given the overhead introduced due to parallelizing and synchronizing the slice decoders.

B. Case Study on a JPEG Encoder

As a second experiment, Table II shows the simulation speedup for a JPEG Encoder example [1] which performs the DCT, $Quantization$ and $Zigzag$ modules for the 3 color components in parallel, followed by a sequential $Huffman$ encoder at the end. Significant speedup is gained by our multi-core parallel simulator for the higher level models ($spec$, $arch$). Simulation performance decreases for the models at the lower abstraction levels ($sched$, net) due to the high number of bus transactions and arbitrations which are not parallelized and introduce large overhead due to the necessary synchronization protection.

V. Summary and Conclusion

SLDL design models at different levels of abstraction are typically validated by use of extensive simulation. Despite explicitly described thread-level parallelism in the design model, traditional SLDL simulators issue only a single simulation thread at any time in order to avoid the otherwise necessary complex synchronization of concurrent threads. Consequently, such simulators cannot utilize the available parallelism in modern multi-core CPU hosts.

In this work, we have compared the execution semantics of concurrent threads in the SystemC and SpecC SLDLs. We have then presented an extension of the SpecC SLDL simulation kernel that supports truly parallel simulation on multi-core hosts. Our careful channel protection scheme for safe synchronization and communication allows our parallel simulator to issue as many simulation threads simultaneously as CPU cores are available. The resulting increase in simulation speed enables significantly faster validation of large SLDL design models. Using two case studies on H.264 video decoding and JPEG encoding we have demonstrated the effectiveness of our approach.

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

TABLE I
SIMULATION RESULTS OF H.264 DECODER ("HARBOUR", 299 FRAMES 4 SLICES EACH, 30 FPS).

Simulator Par. issued threads:		Reference	Multi-Core					
		n/a	1		2		4	
		sim. time	sim. time	speedup	sim. time	speedup	sim. time	speedup
models	spec	20.80s (99%)	21.12s (99%)	0.98	14.57s (146%)	1.43	11.96s (193%)	1.74
	arch	21.27s (97%)	21.50s (97%)	0.99	14.90s (142%)	1.43	12.05s (188%)	1.76
	sched	21.43s (97%)	21.72s (97%)	0.99	15.26s (141%)	1.40	12.98s (182%)	1.65
	net	21.37s (97%)	21.49s (99%)	0.99	15.58s (138%)	1.37	13.04s (181%)	1.64
	tlm	21.64s (98%)	22.12s (98%)	0.98	16.06s (137%)	1.35	13.99s (175%)	1.55
	comm	26.32s (96%)	26.25s (97%)	1.00	19.50s (133%)	1.35	25.57s (138%)	1.03
maximum speedup		1.00	1.00		1.52		2.05	

TABLE II
SIMULATION RESULTS OF JPEG ENCODER.

Simulator Par. issued threads:		Reference	Multi-Core					
		n/a	1		2		4	
		sim. time	sim. time	speedup	sim. time	speedup	sim. time	speedup
models	spec	5.54s (99%)	5.97s (99%)	0.93	4.22s (135%)	1.31	3.12s (187%)	1.78
	arch	5.52s (99%)	6.07s (99%)	0.91	4.28s (135%)	1.29	3.15s (188%)	1.75
	sched	5.89s (99%)	6.38s (99%)	0.92	5.48s (108%)	1.07	5.47s (113%)	1.08
	net	11.56s (99%)	49.3s (99%)	0.23	40.63s (131%)	0.28	37.97s (128%)	0.30

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] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao, and D. D. Gajski. Design of a JPEG encoding system. Technical Report ICS-TR-99-54, Information and Computer Science, University of California, Irvine, November 1999.

[2] 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.

[3] W. Chen, X. Han, and R. Dömer. ESL Design and Multi-Core Validation using the System-on-Chip Environment. In *HLDVT'10: Proceedings of the 15th IEEE International High Level Design Validation and Test Workshop*, 2010.

[4] 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):to appear, May/June 2011.

[5] 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.

[6] R. Dömer, A. Gerstlauer, and D. Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, `http://www.specc.org`, December 2002.

[7] 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.

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

[9] M. Fujita and H. Nakamura. The standard SpecC language. In *Proceedings of the 14th international symposium on Systems synthesis*, ISSS '01, pages 81–86, New York, NY, USA, 2001. ACM.

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

[11] 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.

[12] IEEE Computer Society. *IEEE Standard SystemC Language Reference Manual*. IEEE Std 1666^{TM}-2005, `http://standards.ieee.org/getieee/1666/download/1666-2005.pdf`, March 2006.

[13] H.264/AVC JM Reference Software. `http://iphome.hhi.de/suehring/tml/`.

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

[15] 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.

[16] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann. parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures. In *CODES+ISSS'10: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Scottsdale, AZ, USA, Oct 2010.

[17] 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.