Multicore Simulation of Transaction-Level Models Using the SoC Environment

Weiwei Chen, Xu Han, and Rainer Dömer
University of California, Irvine

Editor's note:
To address the limitations of discrete-event simulation engines, this article presents an extension of the SoC simulation kernel to support parallel simulation on multicore hosts. The proposed optimized simulator enables fast validation of large multicore SoC designs by issuing multiple simulation threads simultaneously while ensuring safe synchronization.
—Prabhat Mishra, University of Florida

■ **MODERN EMBEDDED-SYSTEM** platforms often integrate various types of processing elements into the system, including general-purpose CPUs, application-specific instruction-set processors (ASIPs), digital-signal processors (DSPs), and dedicated hardware accelerators. The large size and great complexity of these systems pose enormous challenges to traditional design and validation. System designers are forced to move to higher abstraction levels to cope with the many problems, which include many heterogeneous components, complex interconnects, sophisticated functionality, and slow simulation.

At the electronic system level (ESL), system design and verification aim at a systematic top-down design methodology that 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 multiprocessor SoC (MPSoC) design.[1] The SCE, beginning with a system specification model described in the SpecC language,[2] implements a top-down ESL design flow based on the specify-explore-refine methodology. The SCE automatically generates a set of transaction-level models (TLMs) with an increasing amount of implementation details through stepwise refinement, resulting in a pin- and cycle-accurate system implementation.

Each TLM in the design flow explicitly specifies key ESL concepts, including behavioral and structural hierarchies, potential for parallelism and pipelining, communication channels, and timing constraints. Having these intrinsic features of the application made explicit in the model enables efficient design space exploration and automatic refinement by CAD tools.

We have exploited the existing explicit parallelism in ESL models to speed up the simulation needed to validate TLM and virtual prototype system models. Thus far, model validation has been based on traditional discrete-event (DE) simulation (see the "DE Simulator Progress" sidebar). The traditional SCE simulator implements the existing parallelism in the design model in the form of concurrent user-level threads within a single process. Its multithreading model is cooperative (i.e., nonpreemptive), which greatly simplifies communication through events and variables in shared memory. Unfortunately, however, this threading model cannot use any available parallelism in multicore host CPUs, which today are common and readily available in regular PCs. (Our focus here has been on accelerating a single simulation instance, not on parallel simulation of multiple models or multiple test vectors).

In comparison to earlier work,[3] we here significantly extend the discussion of proper channel protection; we also explain a new optimization technique that reduces the number of context switches in the simulator and results in higher simulation performance as well as significantly reduced simulation time. Finally, we show new results for a complete

DE Simulator Progress

Most ESL frameworks today rely on regular DE-based simulators that issue only a single thread at any time to avoid complex synchronization of concurrent threads. As such, the simulator kernel becomes an obstacle to improving simulation performance on multicore host machines.[1]

A well-studied solution to this problem is parallel discrete-event simulation (PDES).[2–4] Li et al.[5] and Naroska[6] have described the PDES solution, as implemented with hardware description languages VHDL and Verilog. To apply PDES solutions to today's system-level description languages (SLDLs) and allow parallel execution on multicore processors, the simulator kernel must be modified to issue and properly synchronize multiple operating-system kernel threads in each scheduling step. Chopard et al.[7] and Chandran et al.[8] have extended the SystemC simulator kernel accordingly. Chopard et al. has targeted clusters with single-core nodes,[7] using multiple schedulers on different processing nodes and defining a master node for time synchronization.

Chandran et al. demonstrated a parallelized SystemC kernel for fast simulation on SMP machines, in which multiple runnable operating-system kernel threads were issued in each simulation cycle.[8] Our scheduling approach is similar; however, we propose a detailed synchronization protection mechanism that we automatically generate for any user-defined and hierarchical channels, and we discuss an implementation optimization as well. Moreover, instead of synthetic benchmarks, we provide results for an actual H.264 video decoder and a JPEG encoder.

References

1. K. Huang et al., "Scalably Distributed SystemC Simulation for Embedded Applications," *Proc. Int'l Symp. Industrial Embedded Systems,* IEEE CS Society, 2008, pp. 271-274.
2. K. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Trans. Software Eng.,* vol. SE-5, no. 5, 1979, pp. 440-452.
3. R. Fujimoto, "Parallel Discrete Event Simulation," *Comm. ACM,* vol. 33, no. 10, 1990, pp. 30-53.
4. D. Nicol and P. Heidelberger, "Parallel Execution for Serial Simulators," *ACM Trans. Modeling and Computer Simulation,* vol. 6, no. 3, 1996, pp. 210-242.
5. T. Li, Y. Guo, and S.-K. Li, "Design and Implementation of a Parallel Verilog Simulator: PVSim," *Proc. 17th Int'l Conf. VLSI Design,* IEEE CS Press, 2004, pp. 329-334.
6. E. Naroska, "Parallel VHDL Simulation," *Proc. Design, Automation and Test in Europe Conf.* (DATE 98), IEEE CS Press, 1998, pp. 159-165.
7. B. Chopard, P. Combes, and J. Zory, "A Conservative Approach to SystemC Parallelization," *Int'l Conf. Computational Science,* part 4, LNCS 3994, Springer, 2006, pp. 653-660.
8. E.P.P. Chandran et al., "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," *Proc. ACM/IEEE/SCS 23rd Workshop Principles of Advanced and Distributed Simulation* (PADS 09), IEEE CS Press, 2009, pp. 80-87.

set of SCE-generated TLMs at different levels of abstraction.

Multicore 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 multicore host. However, care must be taken to properly synchronize parallel threads.

Here, we review the scheduling scheme in a traditional simulation kernel that issues only one thread at a time. We present our improved scheduling algorithm with true multithreading capability on symmetric multiprocessing (multicore) machines and discuss the necessary synchronization mechanisms for safe parallel execution. Without loss of generality, we assume the use of SpecC here.

Traditional DE simulation

Both SystemC and SpecC simulators use a traditional DE approach. 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.

To describe the simulation algorithm, we define five data structures and operations (for a formal definition of semantics, see Mueller et al.[4]).

First, we define queues of threads *th* in the simulator:

- `Queues` = {`Ready`, `Run`, `Wait`, `Waitfor`, `Complete`},
- `Ready` = {*th* | *th* is ready to run},
- `Run` = {*th* | *th* is currently running},

- `Wait` = {*th* | *th* is waiting for some events},
- `Waitfor` = {*th* | *th* is waiting for time advance}, and
- `Complete` = {*th* | *th* has completed its execution}.

Second, we define the simulation invariants: Let `Threads` be the set of all threads that currently exist. Then, at any time, the following conditions hold:

- `Threads` = `Ready` ∪ `Run` ∪ `Wait` ∪ `Waitfor` ∪ `Complete`.
- $\forall$ A ∈ `Queues`, $\forall$ B ∈ `Queues`, A ≠ B : A ∩ B = ∅.

Third, we define 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.

Fourth, we define operations on threads with set manipulations: Suppose *th* is a thread in one of the queues, and A and B are queues ∈ `Queues`. Then

- *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. (For formal selection and matching rules, see Mueller et al.[4])
- `Move`(*th*, A, B): move thread *th* from set A to B.

Finally, we define the initial state at the simulation's beginning:

- `Threads` = {th_{root}},
- `Run` = {th_{root}},
- `Ready` = `Wait` = `Waitfor` = `Complete` = ∅, and
- *time* = 0.

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. At any time, as Figure 1a shows, the traditional scheduler runs a single thread picked from the `Ready` queue. Within a delta cycle, the choice of the next thread to run is nondeterministic (by definition). If the `Ready` queue is empty, the scheduler will fill the queue again by waking threads that 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 time stamp 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.

Multicore discrete event simulation

The scheduler for multicore parallel simulation works the same way as the traditional scheduler, with one exception: in each cycle, it picks multiple operating-system 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.

Figure 1b shows the extended control flow of the multicore scheduler. Note the extra loop on the left, which issues operating-system kernel threads as long as CPU cores are available and the `Ready` queue is not empty.

Synchronization for multicore simulation

The benefit of running more than one 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—must be properly protected by locks for mutually exclusive access by the concurrent threads.

Protecting scheduling resources. To protect all central scheduling resources, 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.

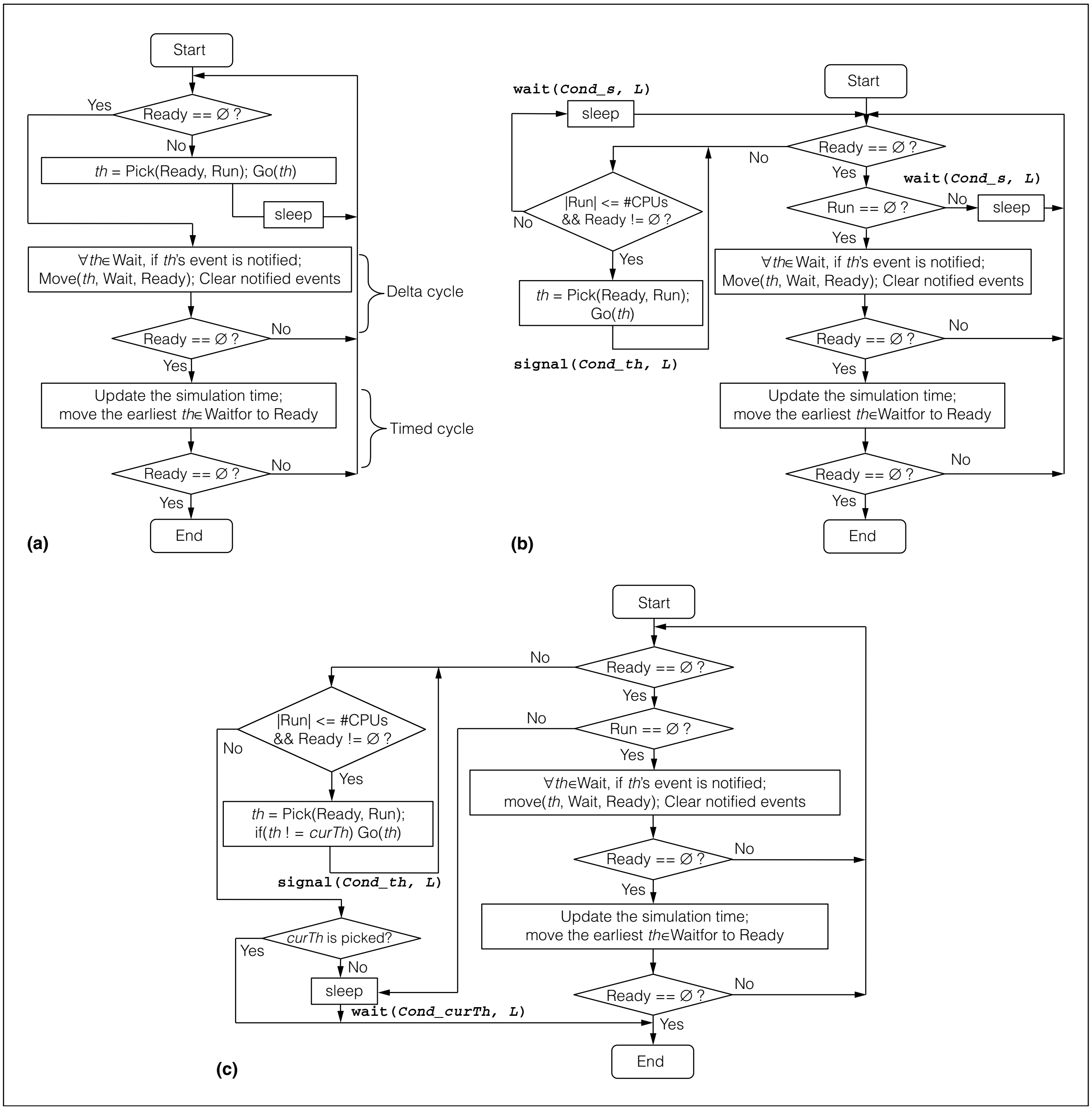

Start
Ready == ∅ ?
Yes
No
th = Pick(Ready, Run); Go(th)
sleep
∀th∈Wait, if th's event is notified; Move(th, Wait, Ready); Clear notified events
Delta cycle
Update the simulation time; move the earliest th∈Waitfor to Ready
Timed cycle
End
(a)
wait(Cond_s, L)
|Run| <= #CPUs && Ready != ∅ ?
th = Pick(Ready, Run); Go(th)
signal(Cond_th, L)
Run == ∅ ?
(b)
th = Pick(Ready, Run); if(th ! = curTh) Go(th)
curTh is picked?
wait(Cond_curTh, L)
(c)

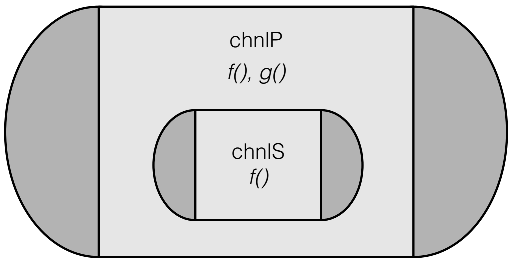
Figure 1. Discrete-event (DE) simulation algorithms: traditional DE scheduler (a); multicore DE scheduler (b); and optimized multicore DE scheduler (c).

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.

Protecting communication. Communication between threads must also be explicitly protected, because SpecC channels are defined to act as monitors (the *SpecC Language Reference Manual* v2.0 explicitly states in section 2.3.2.j that channel methods are protected for mutually exclusive execution). That is, only one thread at a time can execute

```
send(d)                       receive(d)
{                             {
  Lock(this→Lock);              Lock(this→Lock);
  while(n >= size){             while(!n){
     ws ++;                        wr ++;
     wait(eSend);                  wait(eRecv);
     ws - -;                       wr - -;
  }                             }
  buffer.store(d);              buffer.load(d);
  if(wr){                       if(ws){
     notify(eRecv);                notify(eSend);
  }                             }
  unLock(this→Lock);            unLock(this→Lock);
}                             }
```

(a)



(b)

```
channel chnlS;     channel chnlP;
interface itfs
{
   void f(chnlP * chp);
   void g( );
};
channel chnlP( )  implements itfs
{
   chnlS chs;
   void f(chnlP * chp){
      chs.f(this);  //(a)inner channel function call
      g( );         //(b)call another member function
   }
   void g( ) { }
};
channel chnlS( ) implements itfs
{
   void f(chnlP * chp){ chp→g( );}
   //(c) outer channel function call back
};
```

(c)

Figure 2. Protecting synchronization in channels for multicore parallel simulation. Queue channel implementation for multicore simulation (a), example of user-defined hierarchical channels (b), and SpecC description of the channels used in this example (c).

code wrapped in a specific channel instance. To ensure this, we introduce, for each channel instance, a lock *ch* → *Lock*, which is acquired at entry and released upon leaving any method of the channel. Figure 2a shows this for the example of a simple circular, fixed-size buffer.

Channel locking scheme. In general, however, channels can be more complex with multiple nested channels and methods calling one another. For such user-defined channels, we automatically generate a proper locking mechanism as follows. Figure 2b shows an example of a channel, *ChnlS*, wrapped in another channel, *ChnlP. ChnlP* has a method, *ChnlP::f()*, that calls another channel method, *ChnlP::g()*. The inner channel, *ChnlS*, has a method, *ChnlS::f()*, that calls back the outer method, *ChnlP::g()*.

To avoid duplicate lock acquiring and early releasing in channel methods, we introduce a counter *th*→ *ch* → *lockCount* for each channel instance in each thread *th*, and a list *th* → *list* of acquired channel locks for each working thread. Instead of manipulating *ch*→ *Lock* in every channel method, *th* → *ch* → *lockCount* is incremented at the entry, and is decremented before leaving the method, and *ch* → *Lock* is only acquired or released when *th* → *ch* → *lockCount* is zero. Also, *ch* → *Lock* is added into *curTh* → *list* of the current thread when it is first acquired, and removed from the list when the last method of *ch* in the call stack exits. Note that, to avoid the possibility of dead locking, channel locks are stored in acquisition order. The working thread always releases them in reverse order and keeps the original order when reacquiring them. Figure 3 illustrates the refined control flow in a channel method with a proper locking scheme.

The combination of a central scheduling lock and individual locks for channel and signal instances, together with proper locking scheme and well-defined ordering, ensures safe synchronization among many parallel-working threads. The careful ordering, along with the natural order of nested channels, ensures that locks are always acquired in acyclic fashion and, thus, deadlock cannot occur. Figure 4 summarizes the detailed use of all locks and the thread-switching mechanism for the life cycle of a working thread.

Implementation optimization for multicore simulation

As Figure 1b and Figure 4 show, the threads in the simulator undergo context switches as a result of event handling and time advancement. A dedicated scheduling thread introduces context switch overhead when a working thread needs scheduling. This overhead can be eliminated by letting the current working thread perform the scheduling task itself. In other words, rather than using a dedicated scheduler thread, we define a function `schedule()` and call it in each working thread when scheduling is needed.

At this point, the condition variable *Cond_s* is no longer needed. The scheduler thread (which is now the current working thread *curTh* itself) "sleeps" by waiting on *Cond_curTh*. In the left loop of Figure 1b, if the thread picked from the `Ready` queue is the same as *curTh*, the sleep step is skipped and *curTh* continues. After the sleep step, the current working thread will continue its own work rather than entering another scheduling iteration. In Figure 4, `Go (schedule)` is replaced by the function call `schedule()`, and the sleep step is no longer needed. Figure 1c shows the refined multicore scheduler with this optimization applied.

Our experiments show that this optimization reduces simulation time by about 7.8%.

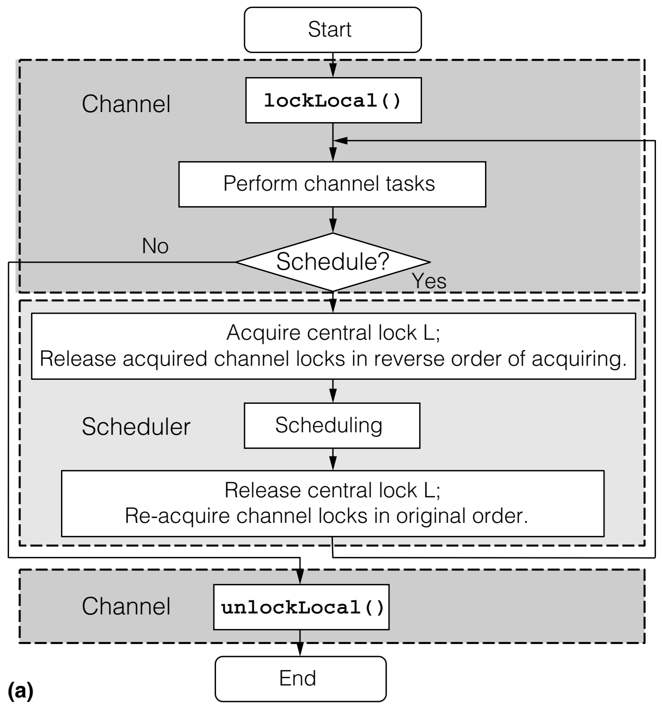
Case study: H.264 video decoder

To demonstrate the improved runtime of our multicore simulator, our first case study involved a video decoder application based on the H.264 Advanced Video Coding (AVC) standard.

H.264 Advanced Video Coding (AVC) standard

The H.264 standard, widely used in video applications such as Internet streaming, disk storage, and television services,[5] provides high-quality video at less than half the bit rate of its predecessors H.263 and H.262. At the same time, however, it requires more computing resources for both video encoding and decoding. To implement H.264 on resource-limited embedded systems, it is highly desirable to exploit available parallelism in its algorithm.

The H.264 decoder's input is a video stream sequence of encoded video frames. A frame can be further split into one or more slices during H.264 encoding, as the upper right part of Figure 5 shows. Notably, slices are independent of one another in the sense that decoding one slice will not require any data from the other slices (although it might need data from previously decoded reference frames). For this reason, parallelism exists at the slice level, and parallel slice decoders can decode multiple slices in a frame simultaneously.



(a)

```
ch→lockLocal( )
{
  if(curTh→ch→lockCount ==0){
    lock(ch→Lock);
    curTh→list.Append(ch→Lock);
  }
  curTh→ch→lockCount ++;
}

ch→unlockLocal()
{
  curTh→ch→lockCount - -;
  if(curTh→ch→lockCount == 0){
    curTh→list.Remove(ch→Lock);
    unlock(ch→Lock);
  }
}
```

(b)

Figure 3. Synchronization protection for the member functions of communication channels.

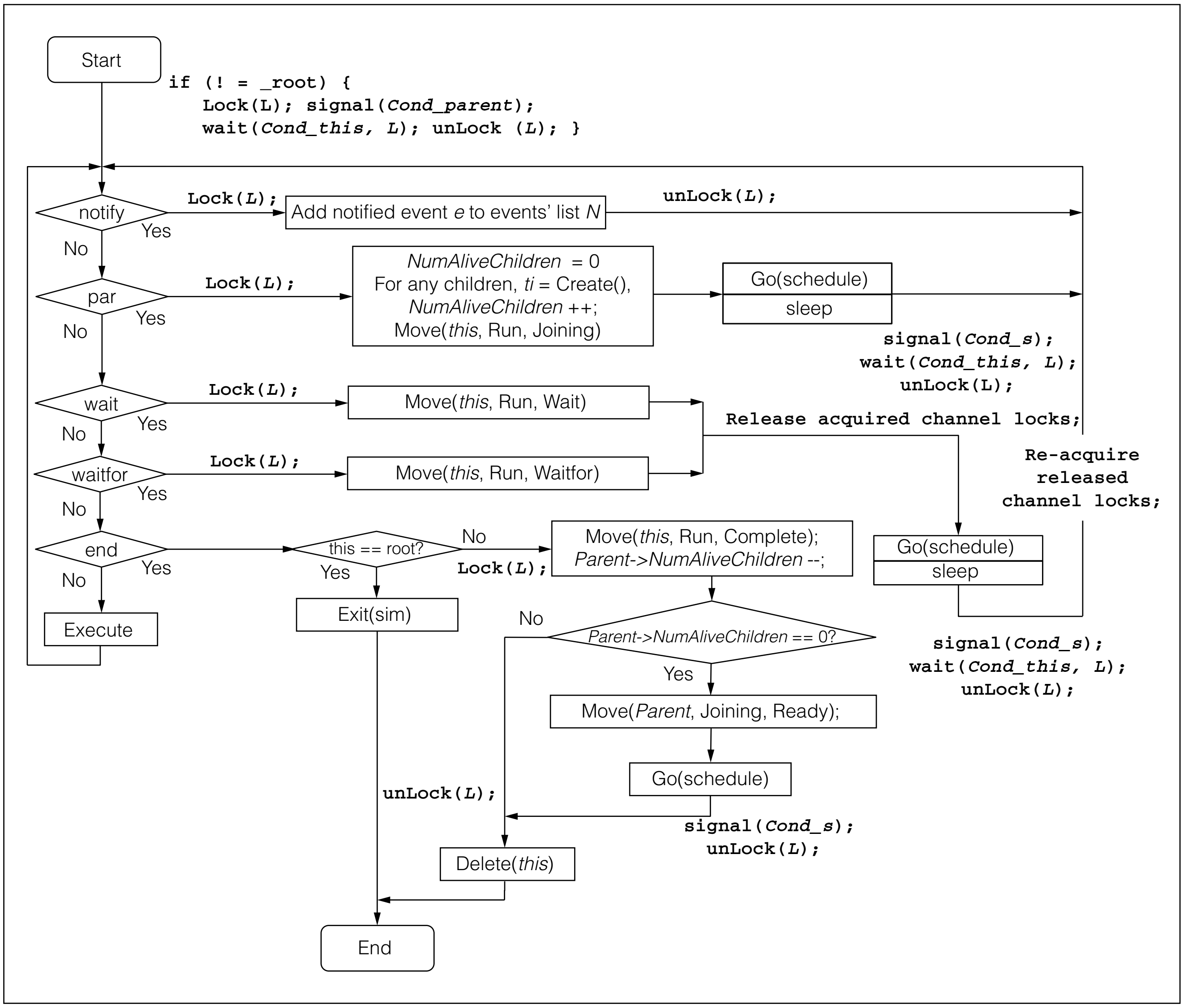


Figure 4. Life-cycle of a thread in the multicore simulator.

H.264 decoder model with parallel-slice decoding

We specified an H.264 decoder model on the basis of the H.264 AVC JM reference software (http://iphome.hhi.de/suehring/tml). In the reference code, a global data structure (`img`) stores the input stream and all intermediate data during decoding. 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 independently. As an exception, the output of each slice decoder is still written to a global data structure (`dec_picture`). This is valid because the macroblocks produced by different slice decoders do not overlap.

Figure 5 shows the block diagram of our model. Frame decoding 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 intraprediction.

Using the SCE, we partitioned the H.264 decoder model by, first, mapping the four slice decoders onto four custom hardware units, and then mapping the synchronizer onto an ARM7TDMI processor at

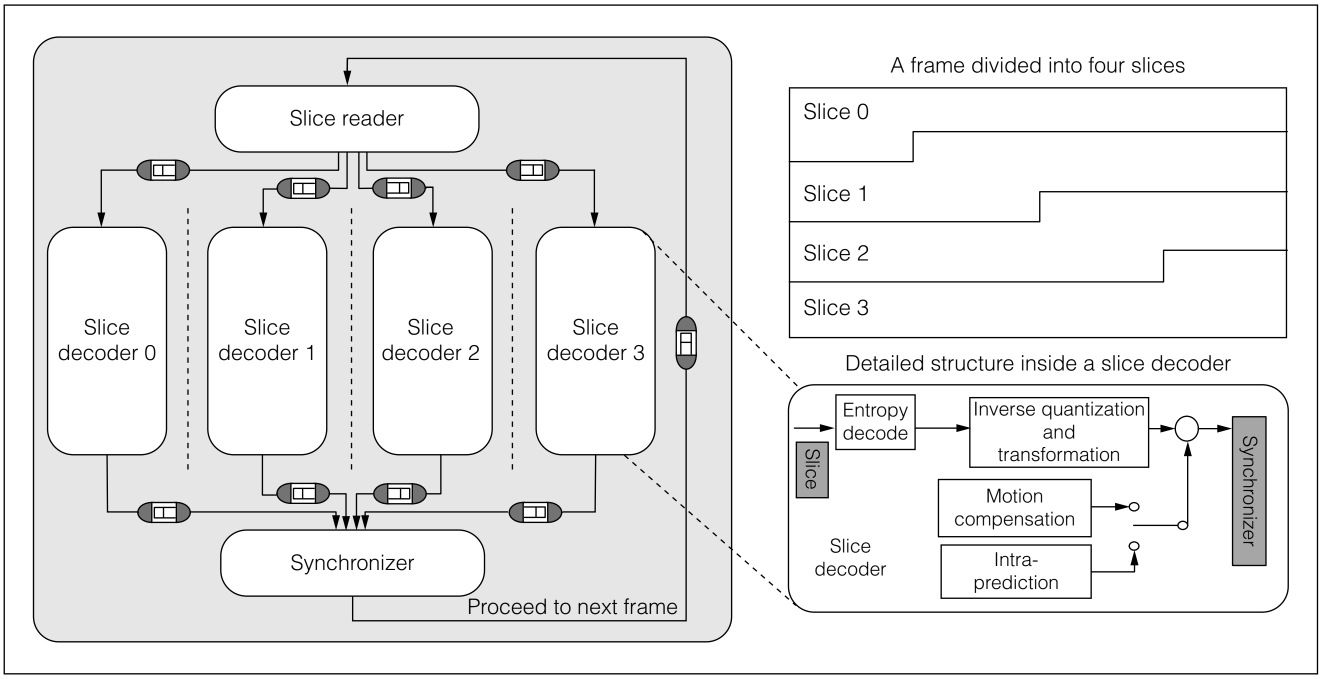


Figure 5. Parallelized H.264 decoder model.

100 MHz, which also implemented the overall control tasks and cooperation with the surrounding testbench. We chose round-robin scheduling for processor tasks and allocated an AMBA (Advanced Microcontroller Bus Architecture) AHB (AMBA high-performance bus) for communication between the processor and the hardware units.

Experimental results

In the first experiment, we used the standard "Harbour" video stream of 299 video frames, each with four equally sized slices. As we estimated in other work,[3] 68.4% of the total computation time was in the slice decoding, which we have parallelized in our decoder model.

We calculated the maximum possible performance gain as follows:

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

For four parallel cores, the maximum speedup is

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

The maximum speedup for two cores was, accordingly, $MaxSpeedup_2 = 1.52$.

Table 1 compares simulation results for several transaction-level models (TLMs) that were generated with SCE. We obtained these results with our multicore simulator on a host PC with a 3-GHz Intel quad-core CPU using the Fedora 12 Linux operating system, compiled with optimization enabled. The table compares the elapsed simulation time against the single-core reference simulator (and also includes the CPU load reported by the operating system). Although simulation performance decreases when issuing only one parallel thread because of additional mutexes for safe synchronization in each channel and the scheduler, our multicore parallel simulation effectively reduced the simulation time for all models when multiple cores in the simulation host were used.

Table 1 also lists the measured speedup and maximum theoretical speedup for the models we created following the SCE design flow. The greater the number of parallel threads issued in each scheduling step, the more speedup is gained. The "No. of delta cycles" column shows the total number of delta cycles executed when simulating each model. This number increases when the design is refined and is the reason we gain less speedup at lower abstraction levels. Also, at lower abstraction levels more communication overhead is introduced, and the increasing need for scheduling reduces the

Table 1. Simulation results of the H.264 decoder (with the "Harbour" video stream: 299 frames, 4 slices each, 30 fps).

	Reference (single-core) simulator	Multicore simulator							
		1 parallel thread		2 parallel threads		4 parallel threads			
SCE design model	Simulation time	Simulation time	Speedup	Simulation time	Speedup	Simulation time	Speedup	No. of delta cycles	No. of threads
Spec	20.80s (99%)	21.12s (99%)	0.98	14.57s (146%)	1.43	11.96s (193%)	1.74	76,280	15
Arch	21.27s (97%)	21.50s (97%)	0.99	14.90s (142%)	1.43	12.05s (188%)	1.76	76,280	15
Sched	21.43s (97%)	21.72s (97%)	0.99	15.26s (141%)	1.40	12.98s (182%)	1.65	82,431	16
Net	21.37s (97%)	21.49s (99%)	0.99	15.58s (138%)	1.37	13.04s (181%)	1.64	82,713	16
Tlm	21.64s (98%)	22.12s (98%)	0.98	16.06s (137%)	1.35	13.99s (175%)	1.55	115,564	63
Comm	26.32s (96%)	26.25s (97%)	1.00	19.50s (133%)	1.35	25.57s (138%)	1.03	205,010	75
Maximum speedup	1.00	1.00		132		2.05		NA	NA

Table 2. Comparison of our earlier implementation with our current optimized implementation for four parallel-issued threads.

	Unoptimized simulator[3]		Optimized simulator		
SCE design model	Simulation time (s)	No. of context switches	Simulation time (s)	No. of context switches	Gain (%)
Spec	13.18	161,956	11.96	81,999	10
Arch	13.62	161,943	12.05	82,000	13
Sched	13.44	175,065	12.98	85,747	4
Net	13.52	178,742	13.04	88,263	4
Tlm	15.26	292,316	13.99	140,544	9
Comm	27.41	1,222,183	25.57	777,114	7

parallelism. However, the measured speedups are somewhat lower than the maximum, which is reasonable, given the overhead introduced, because of parallelizing and synchronizing the slice decoders.

The comparatively lower performance gain achieved with the *comm* model in simulation with four threads is probably due to the unbalanced cache utilization in our Intel Core2 Quad machine.

Table 2 compares the simulation times of our earlier implementation, described elsewhere,[3] with the one we optimized for the work we describe here. The number of the context switches in the optimized simulation dropped to about half and resulted in an average performance gain of about 7.8%.

Using a video stream with four slices in each frame is ideal for our model with four hardware decoders. We can achieve simulation speedup, however, even for less-ideal cases. Table 3 shows the results when the test stream contains different numbers of slices. We also created a test stream file with four slices per frame, where the size of the slices was imbalanced (the ratio of slice sizes was 31%, 31%, 31%, and 7%). Here, the speedup of our multicore simulator versus the reference one is 0.98 for issuing one thread, 1.28 for two threads, and 1.58 for four threads. As expected, the speedup decreased when the available parallel workload was imbalanced.

Case study: JPEG encoder

We conducted a second experiment on a JPEG encoder.[6] Table 4 shows the simulation speedup for a JPEG encoder example,[1] which performed the

Table 3. Simulation speedup for H.264 streams with different numbers of slices per frame for the specification model.

No. of slices per frame	Reference (single-core) simulator	Multicore simulator: 1 parallel thread	Multicore simulator: 2 parallel threads	Multicore simulator: 4 parallel threads
1	1.00	0.98	0.98	0.95
2	1.00	0.98	1.40	1.35
3	1.00	0.99	1.26	1.72
4	1.00	0.98	1.43	1.74
5	1.00	0.99	1.27	1.53
6	1.00	0.99	1.41	1.68
7	1.00	0.98	1.30	1.55
8	1.00	0.98	1.39	1.59

Table 4. Simulation results of a JPEG encoder.

SCE design model	Reference (single-core) simulator	Multicore simulator					
		1 parallel thread		2 parallel threads		4 parallel threads	
	Simulation time	Simulation time	Speedup	Simulation time	Speedup	Simulation time	Speedup
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

DCT, quantization, and zigzag modules for the three color components in parallel, followed by a sequential Huffman encoding module at the end. Significant speedup is gained by our multicore parallel simulator for the higher-level models (Spec, Arch). Simulation performance deteriorated for the models at the lower abstraction levels (*Sched*, *Net*) because of the high number of bus transactions and arbitrations that were not parallelized, which introduce significant overhead because of the necessary synchronization protection.

In conclusion, while our optimized multicore simulation kernel allows the system designer to validate large SoC design models faster by about an order of magnitude, much research work lies ahead for CAD tools like SCE to cope with the major challenges of ESL design. While researchers collaboratively work on system design, synthesis, and verification aspects, we will focus on further improving the modeling and validation of SoC design models. In future work, we plan to optimize our model analysis and multicore simulator in particular by better grouping and partitioning the parallel threads and exploiting CPU affiliation to reduce communication cost and cache misses. We also plan to avoid synchronization insertion in special situations when it is not necessary. ■

Acknowledgments

This work has been supported in part by funding from the National Science Foundation under research grant NSF Award #0747523. We thank the NSF for this 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 NSF. We also thank the reviewers and editors for valuable suggestions to improve this article.

References

1. R. Dömer et al., "System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design," *EURASIP J. Embedded Systems,* 2008, doi:10.1155/2008/647953.
2. D.D. Gajski et al., *SpecC: Specification Language and Design Methodology,* Kluwer Academic Publishers, 2000.
3. W. Chen, X. Han, and R. Dömer, "ESL Design and Multicore Validation Using the System-on-Chip Environment," *Proc. 15th IEEE Int'l High Level Design Validation and Test Workshop* (HLDVT 10), IEEE CS Press, 2010, pp. 142-147.
4. W. Mueller, R. Dömer, and A. Gerstlauer, "The Formal Execution Semantics of SpecC," *Proc. 15th Int'l Symp. System Synthesis,* IEEE Press, 2002.
5. T. Wiegand et al., "Overview of the H.264/AVC Video Coding Standard," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 13, no. 7, 2003, pp. 560-576.
6. L. Cai et al., *Design of a JPEG Encoding System,* tech. report ICS-TR-99-54, Information and Computer Science Dept., Univ. California, Irvine, 1999.

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 exploration of embedded systems. He has an MS in electrical engineering from the Royal Institute of Technology, Sweden.

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, SoC 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, Dept. of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697-2625; weiwei.chen@uci.edu.