ESL Design and Multi-Core Validation using the System-on-Chip Environment

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

***Abstract*—Design at the Electronic System-Level (ESL) tackles the increasing complexity of embedded systems by raising the level of abstraction in system specification and modeling. Aiming at an automated top-down synthesis flow, effective ESL design frameworks are needed in transforming and refining the high-level design models until a satisfactory multi-processor system-on-chip (MPSoC) implementation is reached.**

In this paper, we provide an overview of the System-on-Chip Environment (SCE), a SpecC-based ESL framework for heterogeneous MPSoC design. Our SCE framework has been shown effective for its designer-controlled top-down refinement-based design methodology. After reviewing the SCE design flow, this paper highlights our recent extension of the SCE simulation engine to support multi-core parallel simulation for fast validation of large MPSoC designs. We demonstrate the benefits of the parallel simulation using a case study on a H.264 video decoder application.

I. Introduction

Modern embedded computer systems often consist of a large set of heterogeneous processors with a complex interconnect network. Corresponding to the variety of intended applications, embedded system platforms integrate various types of processing elements into the system, including general-purpose CPUs, application-specific instruction-set processors (ASIPs), digital signal processors (DSPs), as well as dedicated hardware accellerators implemented as application-specific integrated ciruits (ASICs) and intellectual property (IP) components. However, the large size and complexity of these systems poses a great challenge to design and validation using traditional design flows. System designers are forced to move to higher levels of abstraction to cope with the many problems, including large number of heterogeneous components, complex interconnect, sophisticated functionality, and slow simulation.

At the so-called Electronic System Level (ESL), system design and verification aim at a systematic top-down design methodology which successively transforms a given high-level specification into a detailed implementation. In this work, we review the System-on-Chip Environment (SCE), a refinement-based framework for heterogeneous MPSoC design [7]. SCE starts with a system specification model described in the SpecC language [9] and implements a top-down ESL design flow based on the specify-explore-refine methodology. SCE supports a heterogeneous target platform consisting of custom hardware components, embedded software processors, dedicated IP blocks, and complex communication bus architectures. Starting from an abstract specification model of the desired system, models at various levels of abstraction are automatically generated through successive step-wise refinement, resulting in a pin- and cycle-accurate system implementation.

After every refinement step, the generated model is validated through simulation. So far, validation in SCE is based on traditional discrete event (DE) simulation. The original SpecC simulator implements the explicit 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 existing parallelism in multiple host CPUs which nowadays are common and readily available in regular PCs. In Section III, we show an extension of the SCE simulation engine that overcomes this shortcoming.

A. Related Work

Early ESL design, which a decade ago was commonly referred to as hardware/software co-design, includes academic approaches such as COSYMA [16], COSMOS [19], and POLIS [1]. The supported target architecture typically consisted of a single microcontroller combined with a custom hardware co-processor. Later approaches, such as OCAPI [18] and OSSS [10], aim at more complex multi-processor systems. Here, the input model is not specified in separate languages any more (e.g. VHDL for hardware and C for software), but has evolved to use extended C/C++ or variants of SystemC for the common description of hardware and software blocks in the design model. Other SystemC-based frameworks have been developed focusing on the TLM concept, including [5], [13], [14], and [20]. Cosimulation of heterogenous processors with support of different models of computation in the same design model is provided by Ptolemy [4]. Metropolis [2], on the other hand, emphasizes a platform-based design methodology.

While many of the above approaches focus more on modeling and validation than on synthesis, the SpecC-based SCE described in Section II provides a systematic design flow down to an implementation based on iterative refinement.

Today, several commercial approaches exist that support the ESL design and verification methodology, in particular in the form of hardware synthesis from C-like languages. Examples include Bluespec [3], Catapult C [15], and Forte Design [8]. While these tools start from different specification languages, all of them offer a path down to an actual implemenation in silicon.

For validation, most ESL frameworks rely on regular DE-based simulators which map the parallelism in the design model on multiple threads in the simulation process. However, usually only a single thread is run at any time to avoid complex synchronization of the concurrent threads. As such, the simulator kernel becomes an obstacle in improving simulation performance by using multi-core machines [11]. To actually allow multi-core parallel simulation, the simulator kernel needs to be modified to issue and properly synchronize multiple OS kernel threads in each scheduling step. [6] and [17] have extended the SystemC simulator kernel accordingly. Clusters with single-core nodes are targeted in [6] 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 [17] which issues multiple runable OS kernel threads in each simulation cycle. Our approach described in Section III is very similar. However, instead of synthetic benchmarks, we provide results for an actual H.264 video decoder.

II. SCE Design Flow

As mentioned above, the ESL design flow in the SCE framework starts with an abstract specification model written in the SpecC [9] system-level description language (SLDL). In contrast to flat and sequential C/C++ programming code, this system specification contains key concepts needed for ESL design *explicitly* expressed in the design model, including behavioral and structural hierarchy, potential for parallelism or pipelining, communication separated into channels, and any 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.

The initial specification model is subsequently refined by specific tools integrated in the SCE framework which automatically generate various Transaction Level Models (TLM), each with an increasing amount of implementation detail, and a final implementation model that is pin- and cycle-accurate. In this design process, SCE relies on various component models in its database and, most importantly, on design decisions made by the system designer through an intuitive graphical user interface (GUI) or powerful scripting capabilities.

Fig. 1 shows the entire refinement-based design flow in the SCE framework in an overview [7]. In the following sections, we will briefly review the four system refinement stages which lead into the separated software generation and hardware synthesis tools.

Architecture Exploration is the first step in the SCE design flow. Here, the system designer defines the target platform by allocating processing elements including software processors, hardware accelerators, and communication and IP units. He then maps the computation blocks in the specification onto the selected platform components. After this decision making by the designer, the architecture refinement tool automatically

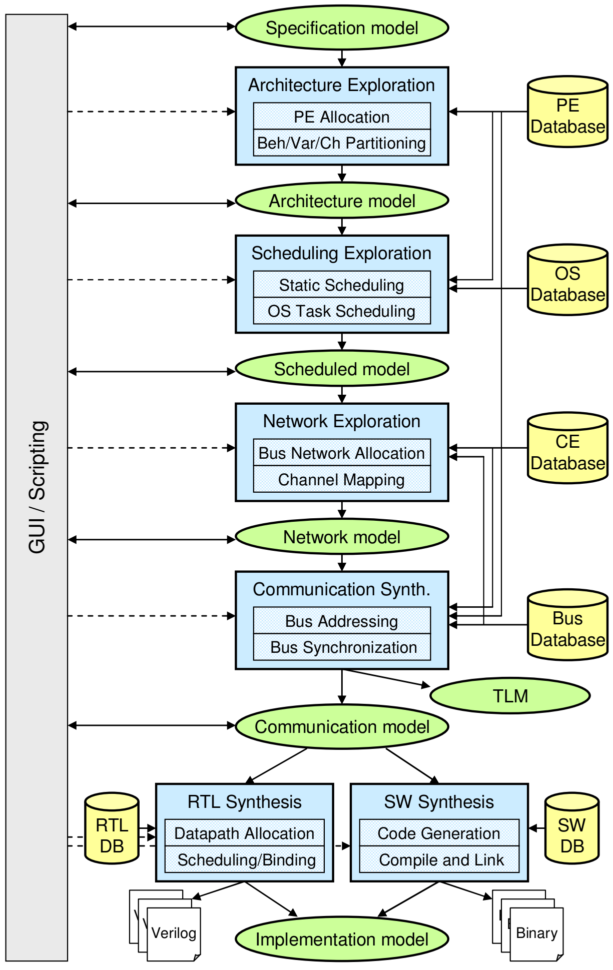

Fig. 1. Refinement-based design flow in the SCE framework [7].

partitions the specification and creates an architecture model which accurately reflects the system platform.

Scheduling Exploration then allows the designer to evaluate different static and dynamic scheduling strategies on the software processors in the platform. SCE scheduling refinement offers several abstract RTOS models with different scheduling algorithms, including round-robin, priority-based, or first-come-first-served scheduling. The automatically inserted RTOS models support task management, real-time scheduling, preemption, task synchronization, and interrupt handling.

Network Exploration is the first step towards communication synthesis in SCE. Here, the system designer specifies the overall communication topology in the platform and maps the channels used in the model onto a network of busses and bridges. Based on these decisions, the network refinement tool then automatically inserts the required communication components from the database into the model and establishes end-to-end communication over point-to-point links.

Communication Synthesis then refines the individual point-to-point links down to an implementation over the actual communication protocol and bus media (wires). As a result, a pin- and bit-accurate model of the communication architecture is obtained. In addition to this pin-accurate model (PAM), SCE can alternatively generate a corresponding TLM that abstracts away the pin-level details of individual bus transactions and simulates significantly faster.

After the four system refinement stages, the hardware and software components in the system model are implemented separately by dedicated hardware and software synthesis tools, respectively.

RTL Synthesis generates, for each hardware component in the system, a structural RTL model from the behavioral description (C code) in the design. This behavioral synthesis step is fully automatic in SCE. However, the system designer can overwrite all synthesis decisions at will, including scheduling, allocation, and binding. SCE hardware synthesis produces structural RTL output in both Verilog HDL (for further logic synthesis) and SpecC SLDL (for simulation within the entire system context).

Software Synthesis, on the other hand, provides code generation for all software components in the design. Here, SCE follows a layer-based structure of the programmable processors and the software stack executing on them. Code is automatically generated for the application, the selected RTOS is configured, required communication stacks are synthesized, and the final binary image file is cross-compiled and linked. The generated SW image can be used for both the implementation on the actual target platform as well as in an instruction-set simulator (ISS) context within the system model.

In summary, at the end of the SCE ESL design flow, a complete pin- and cycle-accurate implementation model of the intended MPSoC has been generated.

III. Multi-Core Parallel Simulation

Design models written in SLDLs contain explicitly specified parallelism which makes it straightforward and 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 parallel threads.

In this section, we will first review the scheduling scheme in the traditional simulation kernel that issues only one thread at any 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. Without loss of generality, we assume the use of SpecC SLDL here (i.e. our technique is equally applicable to SystemC).

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 increase simulation time using *wait-for-time* constructs.

To formally 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** $\cup$ **RUN** $\cup$ **WAIT** $\cup$ **WAITFOR** $\cup$ **COMPLETE**.
   - $\forall$ A, B $\in$ **QUEUES**, A $\neq$ B : A $\cap$ B = $\varnothing$.
3) Operations on threads:
   Suppose th is a thread currently running on a CPU.
   - **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 $\in$ **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 (according to certain rules) 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** = $\varnothing$.
   - $time$ = 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. Fig. 2 shows the control flow of the traditional scheduler. At any time, the 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. At any time, there is only one thread actively executing in the traditional simulation.

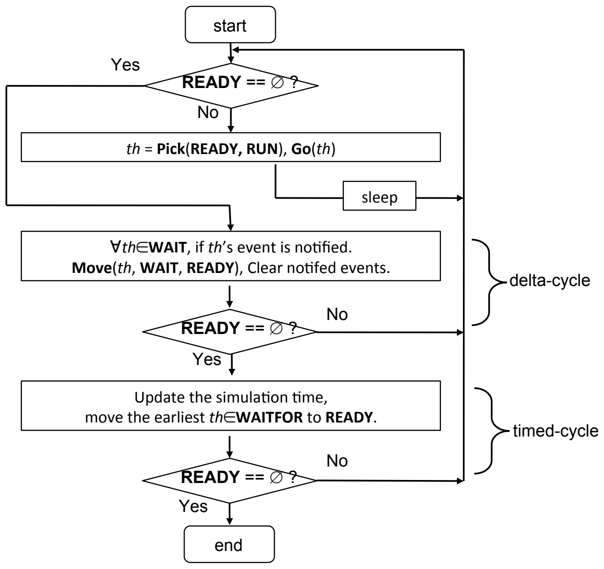
start
Yes
READY == ∅ ?
No
th = Pick(READY, RUN), Go(th)
sleep
∀th∈WAIT, if th's event is notified.
Move(th, WAIT, READY), Clear notifed events.
delta-cycle
No
READY == ∅ ?
Yes
Update the simulation time,
move the earliest th∈WAITFOR to READY.
timed-cycle
No
READY == ∅ ?
Yes
end

Fig. 2. Traditional SLDL scheduler.

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.

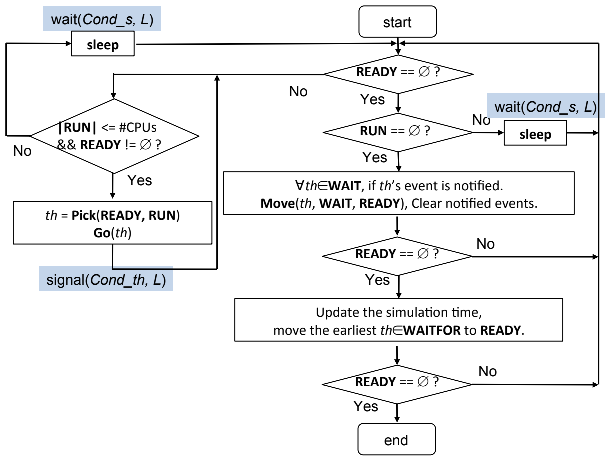
wait(Cond_s, L)
sleep
start
READY == ∅ ?
No
Yes
wait(Cond_s, L)
No
|RUN| <= #CPUs && READY != ∅ ?
RUN == ∅ ?
sleep
No
Yes
Yes
∀th∈WAIT, if th's event is notified.
Move(th, WAIT, READY), Clear notified events.
th = Pick(READY, RUN) Go(th)
signal(Cond_th, L)
READY == ∅ ?
No
Yes
Update the simulation time,
move the earliest th∈WAITFOR to READY.
READY == ∅ ?
No
Yes
end

Fig. 3. Multi-core SLDL scheduler.

Fig. 3 shows the extended control flow of the multi-core scheduler. Note the extra loop at the left which issues OS kernel 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 in the scheduler, and shared variables in communication channels of the application need to be properly protected by locks for mutual exclusive access by the concurrent threads.

1) 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.

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.

2) Protecting Communication: Communication between threads also needs to be explicitly protected as SLDL channels are defined to act as monitors. That is, only one thread at a time may execute code wrapped in a specific channel instance. To ensure this, we introduce a lock *ch→L* for each channel instance which is acquired at entry and released upon leaving any method of the channel. Fig. 4 shows this for the example of a simple circular buffer with fixed size.

```
send(d)                          receive(d)
{                                {
  Lock(this->L);                   Lock(this->L);
  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->L);                 unLock(this->L);
}                                }
```

Fig. 4. Queue channel implementation for multi-core simulation.

The combination of a central scheduling lock and individual locks for channel instances ensures safe synchronization among many parallel working threads. Fig. 5 summarizes the detailed use of these locks and the thread switching mechanism for the life-cycle of a working thread.

IV. Case Study on a H.264 Video Decoder

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

A. H.264 Advanced Video Coding (AVC) Standard

The H.264 AVC standard [21] 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

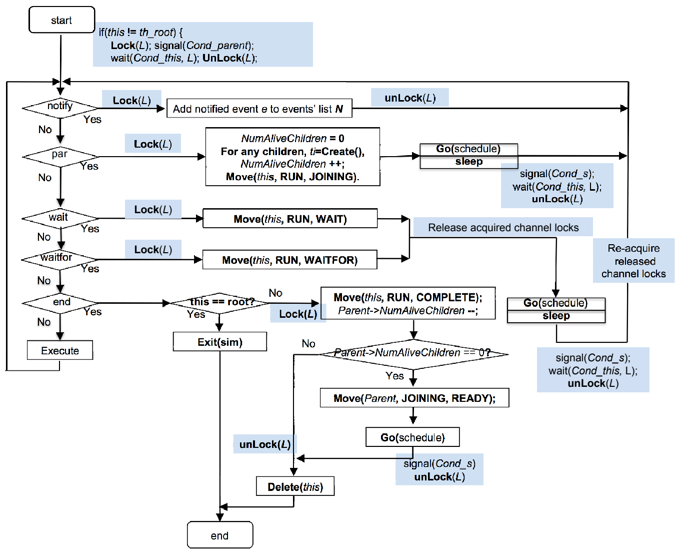
start
if(this != th_root) {
Lock(L); signal(Cond_parent);
wait(Cond_this, L); UnLock(L);
notify
Yes
No
Lock(L)
Add notified event e to events' list N
unLock(L)
par
Lock(L)
NumAliveChildren = 0
For any children, ti=Create(),
NumAliveChildren ++;
Move(this, RUN, JOINING).
Go(schedule)
sleep
signal(Cond_s);
wait(Cond_this, L);
unLock(L)
wait
Move(this, RUN, WAIT)
Release acquired channel locks
waitfor
Move(this, RUN, WAITFOR)
Re-acquire released channel locks
end
this == root?
Move(this, RUN, COMPLETE);
Parent->NumAliveChildren --;
Go(schedule)
sleep
Execute
Exit(sim)
Parent->NumAliveChildren == 0?
Move(Parent, JOINING, READY);
Go(schedule)
signal(Cond_s)
unLock(L)
Delete(this)
end

Fig. 5. Life-cycle of a thread in the multi-core simulator.

standard on resource-limited embedded systems, it is highly desirable to exploit parallelism in its algorithm.

The H.264 decoder takes a video stream as input 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 Fig. 6. 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 that reason, parallelism exists at the slice-level and parallel slice decoders can be used to decode multiple slices in a frame simultaneously.

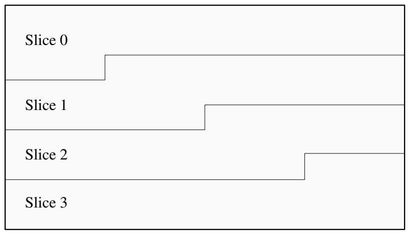

Fig. 6. Example of a H.264 AVC frame divided into four slices.

B. H.264 Decoder Model with Parallel Slice Decoding

We have specified a H.264 decoder model based on the H.264/AVC JM reference software [12]. In the reference code, a global data structure (*img*) is used to store the input stream 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 the input stream 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.

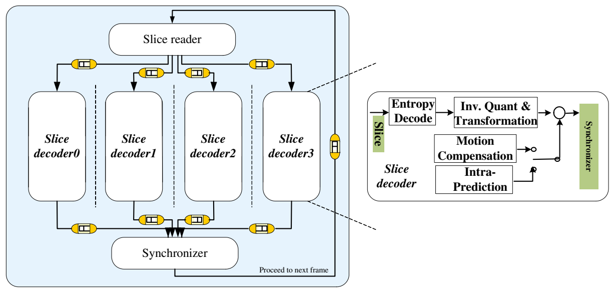

Fig. 7. Parallelized H.264 decoder model.

Fig. 7 shows a 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.

C. Experimental Results

For our experiment, we have prepared a test stream ("Harbour") of 299 video frames, each with 4 slices of equal size. Profiling the JM reference code with this stream showed that 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 can then calculate the maximum possible performance gain as follows:

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

For 4 parallel cores, the maximum speedup is therefore

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

The maximum speedup for 2 and 3 cores is accordingly $MaxSpeedup_2 = 1.52$ and $MaxSpeedup_3 = 1.84$.

Table I lists the results of our multi-core simulator on a host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at 3.0 GHz. First, we compare the elapsed simulation time against the single-core reference simulator. Clearly, our multi-core parallel simulation is very effective in reducing the simulation time.

TABLE I
Simulation Results ("Harbour", 299 frames, 30 fps).

Simulator:	Reference	Multi-Core			
Par. issued threads:	n/a	1	2	3	4
User time:	32.05s	31.83s	32.35s	32.70s	33.36s
System time:	0.51s	1.49s	1.89s	1.74s	1.81s
Elapsed time:	**32.59s**	**33.65s**	**25.22s**	**23.13s**	**17.76s**
CPU load:	99%	99%	136%	148%	197%
Measured speedup:	**1.00**	**0.97**	**1.29**	**1.41**	**1.84**
Maximum speedup:	1.00	1.00	1.52	1.84	2.05

Fig. 8 shows a bar chart comparing the measured speedup against the theoretical maximum. For our multi-core simulator, the measured speedups are somewhat lower than the maximum, which is reasonable given the overhead introduced due to parallelizing and synchronizing the slice decoders. The comparatively lower performance gain for the 3-core simulation can be explained due to under-utilization of 3 cores when running 4 parallel processes.

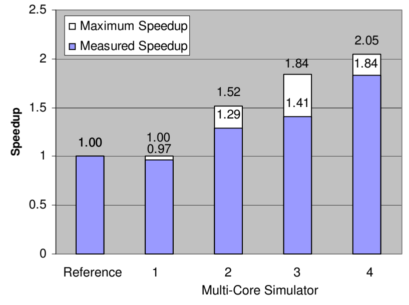

Fig. 8. Comparison of measured and maximum performance gains.

V. Summary and Conclusion

The System-on-Chip Environment (SCE) addresses the increasing complexity of embedded system design by raising the level of abstraction to the Electronic System-Level (ESL). SCE is an example of an effective ESL framework that provides an automated top-down synthesis flow. Starting from an abstract specification model described in the SpecC SLDL, SCE allows the system designer to explore different design alternatives at various levels of abstraction, and to refine the design model step-by-step down to a pin- and cycle-accurate implementation. At each step, the system designer makes the decisions and SCE tools generate new models automatically, which then can be validated through simulation. The resulting MPSoC platform can consist of a set of heterogeneous components, including general-purpose or application-specific processors and dedicated hardware accellerators, interconnected via a complex network of communication busses.

In this paper, we have presented our recent extension of the SCE simulator kernel in order to support parallel simulation on multi-core hosts. Issuing multiple simulation threads simultaneously while ensuring safe synchronization, our simulator allows the fast validation of large MPSoC designs. Using a case study on a H.264 video decoder application, our experimental results demonstrate a significant reduction in simulation time close to the theoretical maximum.

Acknowledgment

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

References

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer, 1997.

[2] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Environment for Electronic System Design. *IEEE Computer*, 36(4), April 2003.

[3] Bluespec, Inc. Bluespec. http://www.bluespec.com/.

[4] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Intl. Journal of Computer Simulation*, 4(2):155–182, April 1994.

[5] W. O. Cesário, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, L. Gauthier, and M. Diaz-Nava. Multiprocessor SoC Platforms: A Component-Based Design Approach. *IEEE Design and Test of Computers*, 19(6), November/December 2002.

[6] 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.

[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] Forte Design Systems. Forte. http://www.forteds.com/.

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

[10] K. Grüttner, F. Oppenheimer, W. Nebel, A.-M. Fouilliart, and F. Colas-Bigey. SystemC-based Modelling, Seamless Refinement, and Synthesis of a JPEG 2000 Decoder. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2008.

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

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

[13] T. Kempf, M. Dörper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2005.

[14] W. Klingauf, H. Gädke, and R. Günzel. TRAIN: A Virtual Transaction Layer Architecture for TLM-based HW/SW Codesign of Synthesizable MPSoC. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2006.

[15] Mentor Graphics. Catapult C Synthesis. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthe%sis/.

[16] A. Österling, T. Brenner, R. Ernst, D. Herrmann, T. Scholz, and W. Ye. The COSYMA system. In J. Staunstrup and W. Wolf, editors, *Hardware/Software Co-Design: Principles and Practice*. Kluwer, 1997.

[17] 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.

[18] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens. A Programming Environment for the Design of Complex High Speed ASICs. In *Proceedings of the Design Automation Conference (DAC)*, San Francisco, CA, June 1998.

[19] C. A. Valderrama, M. Romdhani, J.-M. Daveau, G. F. Marchioro, A. Changuel, and A. A. Jerraya. COSMOS: A transformational Co-Design Tool for Multiprocessor Architectures. In J. Staunstrup and W. Wolf, editors, *Hardware/Software Co-Design: Principles and Practice*. Kluwer, 1997.

[20] K. van Rompaey, D. V. I. Bolsens, and H. D. Man. CoWare: A Design Environment for Heterogeneous Hardware/Software Systems. In *Proceedings of the European Design Automation Conference (Euro-DAC)*, Geneva, Switzerland, September 1996.

[21] 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.