Abstract, Multifaceted Modeling of Embedded Processors for System Level Design

Gunar Schirner, Andreas Gerstlauer and Rainer Dömer.
Center for Embedded Computer Systems
University of California, Irvine, USA
{hschirne,gerstl,doemer}@cecs.uci.edu

Abstract— Embedded software is playing an increasing role in todays SoC designs. It allows a flexible adaptation to evolving standards and to customer specific demands. As software emerges more and more as a design bottleneck, early, fast, and accurate simulation of software becomes crucial. Therefore, an efficient modeling of programmable processors at high levels of abstraction is required.

In this article, we focus on abstraction of computation and describe our abstract modeling of embedded processors. We combine the computation modeling with task scheduling support and accurate interrupt handling into a versatile, multi-faceted processor model with varying levels of features.

Incorporating the abstract processor model into a communication model, we achieve fast co-simulation of a complete custom target architecture for a system level design exploration. We demonstrate the effectiveness of our approach using an industrial strength telecommunication example executing on a Motorola DSP architecture. Our results indicate the tremendous value of abstract processor modeling. Different feature levels achieve a simulation speedup of up to 6600 times with an error of less than 8% over a ISS based simulation. On the other hand, our full featured model exhibits a 3% error in simulated timing with a 1800 times speedup.

I. Introduction

With increasing complexity of modern SoC designs a larger design space has to be explored throughout the design process. At the same time, shorter product life cycles require a reduction in time-to-market, which demands more and more efficient design cycles. System-Level-Design is one solution to address this need based on an increased level of abstraction.

Transaction Level Modeling (TLM) [10] is a widely accepted approach for abstracting communication. It dramatically increases the simulation speed and is an efficient enabler for exploring a larger design space. Motivated by the more than encouraging results of communication TLM, we will focus in this paper on abstraction of computation. We address the need for abstract modeling and simulation of programmable processors, which play an increasingly significant role in todays SoCs, allowing adaptation to emerging standards and specific customer demands.

Traditionally, computation is simulated using an Instruction Set Simulator (ISS), which provides functional and timing accurate simulation on a host platform at a very fine granularity. However, an interpreting ISS simulates very slowly, making it unacceptable for a realistic system design space exploration. Host compiled ISS schemes have been developed to address the speed disadvantage. They simulate faster, but may not be accurate in all cases. In addition, their execution performance is not yet sufficient to match the needs for a rapid design space exploration at the system level. Thus, a higher level of abstraction is needed.

In this article, we describe our approach to the abstraction of a software execution environment as a complement to the TLM for communication abstraction. In particular, we will describe our abstract modeling of a processor as an integral part of a typical system design.

Using an abstract processor model in conjunction with the communication TLM dramatically increases the execution speed in a co-simulation environment. With high accuracies in timing, it enables an early functional and fast simulation of the desired target architecture, exposing the implications of architectural decisions, and thus allowing rapid design space exploration.

A. Problem Definition

We address the need for fast software simulation by abstracting the software execution environment providing timed execution, dynamic scheduling and external communication. We develop a corresponding high-level, abstract processor model. We aim to significantly increase simulation performance while maintaining an acceptable accuracy in simulated timing and exposing the structure of the software architecture (e.g. drivers and interrupts).

Our target architecture (see Fig. 1) is a CPU/DSP subsystem with a single embedded processor driving one system bus. We consider processors that contain an internal memory, which stores the execution binaries and local variables. The processor communicates with external memory (holding globally shared variables) and with custom hardware IP blocks (through memory mapped I/O and interrupts) over the shared single bus.

At the input, we assume that the user application is given in the form of C code for each task, including information about task relations. Furthermore, we assume that data about execution delays for each task is available.

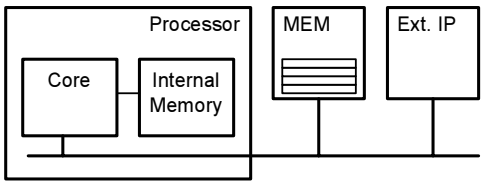

Fig. 1. Generic target architecture.

1-4244-0630-7/07/$20.00 ©2007 IEEE.

B. Outline

After introducing the relevant related work in Section II, we will outline our high-level processor modeling approach in Section III, discussing feature candidates for processor abstraction.

Section IV constitutes the main part of the article. We will incrementally lay out our processor modeling feature by feature. In total, we distinguish five separate feature levels.

Typically, the speedup through abstraction comes at a loss in accuracy. To validate our approach, we will quantitatively analyze each feature level in Section V, allowing us to determine essential features for an efficient processor abstraction. We conclude the paper with a summary in Section VI.

II. Related Work

System level modeling has become an important research area that aims to improve the SoC design process and its productivity. Languages for capturing SoC models have been developed, e.g. SystemC [10] and SpecC [9]. The languages provide means to describe systems, but by themselves do not offer any modeling solutions.

Using TLM [10] for capturing and designing communication architectures has received much attention. Abstracting computation, on the other hand, as an essential element of the system level exploration has been introduced only recently.

Bouchhima et al. [3] describe an abstract CPU subsystem that allows execution of target code on top of a hardware abstraction layer that simulates the processor capabilities. Their approach includes multiple processors on a higher level of abstraction. In contrast, our proposed solution provides a finer grained model with the resulting feature observability advantages at similar simulation performance levels.

Kempf et al. [11] introduce their Virtual Processing Unit for analysis of task mapping and scheduling effects using a quantitative model. They do not, however, include any processor specific features, such as interrupts.

At the very high abstraction level of application modeling, Ptolemy [4] uses a modeling environment that integrates different models of computation (such as petri nets and boolean dataflow) in a hierarchically connected graph.

The traditional approach of an ISS based co-simulation is provided by several commercial vendors, such as ARM's SoC Designer with MaxSim Technology [1], VaST Systems' [16] virtual system prototyping tools and CoWare's [7] Virtual Platform Designer. In addition, ISS based co-simulation is used in many academic projects, such as the MPARM [2] platform.

III. Approach

Traditionally, software is simulated using an Instruction Set Simulator (ISS), which emulates the target Instruction Set Architecture (ISA). For co-simulation, the ISS is wrapped to adapt the ISS API to the simulation environment, as shown in Fig. 2 The wrapper translates the ISS bus accesses to and relays the interrupt inputs from the simulation environment.

The ISS includes internal memory for storing the executed binaries and local variables. It provides a complete software execution environment with all features. Our abstract processor

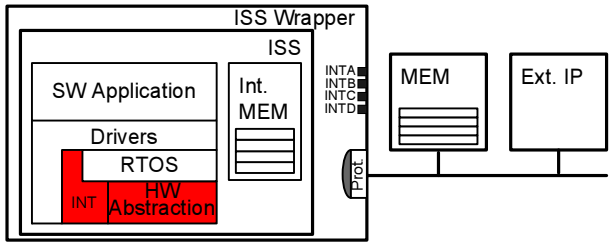

Fig. 2. Bus Functional Model with ISS.

model replaces the ISS plus the surrounding wrapper with an abstract, high-level representation. With the ISS as our reference model in mind, we will now discuss features as candidates for an expressive simulation with regards to exposed functional aspects and timing accuracy.

For simulation of the application given in the input, we employ an approach that executes the user C code natively on the simulation host. This dramatically increases the simulation speed. We achieve target processor specific execution timing, which is the most fundamental requirement for any performance evaluation, by back-annotating timing information into the C code at the function level.

In terms of processor features, we abstract away the processor micro structure and do not simulate the Instruction Set Architecture. Instead, we abstractly model the behavior of the processor and the software environment, including task scheduling, interrupt handing and external communication.

In a typical software architecture, a processor executes more than one function. In such a case, the application is mapped to different tasks and a task scheduler dynamically assigns tasks to the CPU for execution. To observe the effects of dynamic scheduling, an accurate model of the hierarchical task graph and the target specific dynamic scheduling is necessary.

For communication with external components, the choices within the low level drivers are important to the overall system performance. Therefore, a software simulation environment should expose the concepts of interrupts and low level drivers. Additionally, proper scheduling of interrupts is desired, including models for suspension of task execution, interrupt priorities and interrupt nesting.

IV. Processor Modeling

In previous work related to communication modeling [15], we have observed the effectiveness of a layered approach. Therefore, we structure our processor model in layers along features. Our model has five different feature levels with an increasing number of represented features. The subsequent paragraphs will describe each feature level. Later, we will use the feature levels for a fine grained analysis and evaluate each feature for its cost in simulation speed and contribution to timing accuracy.

A. Application

Our first feature level, the *application* level, is equivalent to a timed simulation of the user application natively on the simulation host. We use it as the innermost layer of our abstract processor, as depicted in Fig. 3. In addition, we wrap the user code in an SLDL as a hierarchical composition of behaviors separating computation and communication.

Communication is expressed using abstract channels for high-level, typed message passing. Global variables, which are shared between processing elements, are accessed directly at this abstraction level without special synchronization or bus communication.

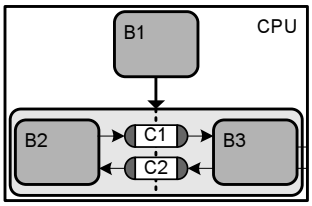

Fig. 3. Application model.

The user application is executed natively on the simulation host inside the discrete event simulation environment. The native execution allows for highest simulation speeds. Wait-for-time statements, inserted at the function level, emulate the target specific execution timing.

B. Task Scheduling

Concurrent execution of software on the same processor requires an operating system on the target. In order to explore the effects of dynamic scheduling decisions, we wrap behaviors to tasks and schedule them using an abstract task scheduler as shown in Fig. 4. Abstractly modeling a task scheduler, in contrast to using a real RTOS implementation, integrates better with the simulation environment, enables highest execution performance and thus allows early exploration of scheduling decisions.

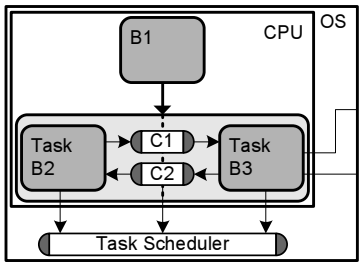

Fig. 4. Task model.

The behaviors of the user application are mapped to individual tasks in this model. Each primitive that could trigger scheduling (e.g. task start, channel communication, wait-for-time statements) is wrapped to interact with the abstract scheduler, which emulates the dynamic scheduling on top of the SLDL framework. Furthermore, in this step, the communication via external global variables is properly wrapped into channel communication. All external communication simulates concurrently and untimed at this feature level.

C. Firmware

The firmware model adds the features of interrupt handling and low level software drivers as shown in Fig. 5. This model is the first that contains the complete software. Its layer marks the boundary between software implementation and hardware features.

At this abstraction level, the interrupt sources trigger the system interrupt by a direct channel call. Thus, interrupts are not scheduled and may execute concurrently to the user application. The utilized system interrupts are connected to the user interrupt handler. In the example of Fig. 5, the system interrupt *INTC* is shared between two interrupt sources. Hence, the system interrupt handler demultiplexes it to two user interrupts (*UsrInt1* and *UsrInt2*).

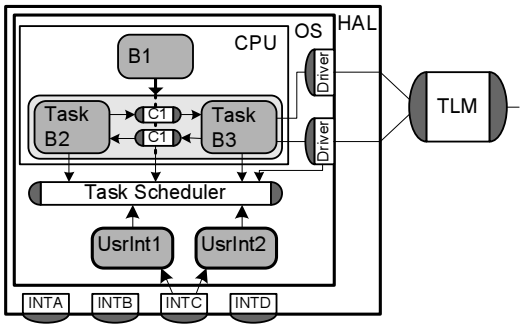

Fig. 5. Firmware model.

External communication is refined down to low level software drivers. The drivers implement synchronization with external components, e.g. through interrupts or polling. They also introduce system absolute addressing and communication with external components using a Transaction Level Model (TLM). The bus TLM simulates the bus at a granularity of user transactions (arbitrary sized blocks of data) and introduces timed communication with external components on the simulated bus.

D. Processor Transaction Level Model

Our complete processor model adds a description of the processor hardware. This includes hardware interrupt handling and bus accesses at bus transaction granularity, as shown in Fig. 6.

For a correct interrupt simulation, the behavior *HW Int* monitors the interrupt lines. Upon occurrence of an interrupt, *HW Int* suspends the main simulation thread, which executes the user application and the scheduler, and triggers the activated system interrupt handler. The *HW Int* observes interrupt priorities and interrupt nesting for an accurate scheduling. Additionally, it provides interrupt related control registers to the driver software e.g. for enabling and disabling interrupts.

The processor TLM also introduces a bus specific Media Access Layer (MAC). It splits the arbitrary sized user transactions into bus transactions (bus protocol primitives) for accessing the processor bus. The Arbitrated TLM (ATLM) simulates the bus protocol with a granularity of bus transactions. In the more general case, the ATLM can perform accurate arbitration. However, in our single master target architecture arbitration is not necessary.

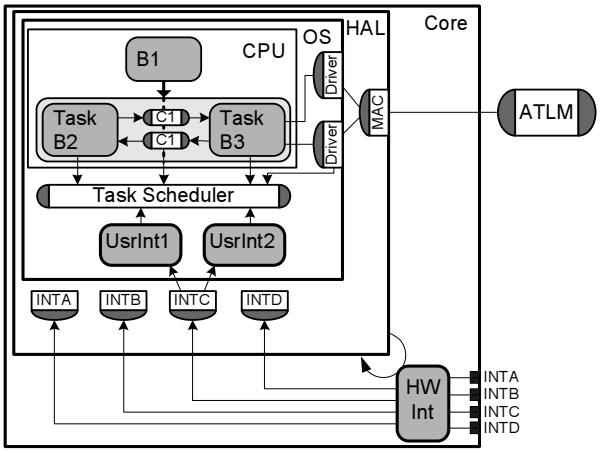

Fig. 6. Processor Transaction Level Model.

E. Processor Bus Functional Model

The bus functional variant of our processor model, see Fig. 7, uses a pin- and cycle-accurate model of the processor bus interface. Its modeling of computation is identical to the Transaction Level Model. However, the BFM includes an implementation of the bus interface state machines that realize the bus protocol by driving and sampling the individual wires.

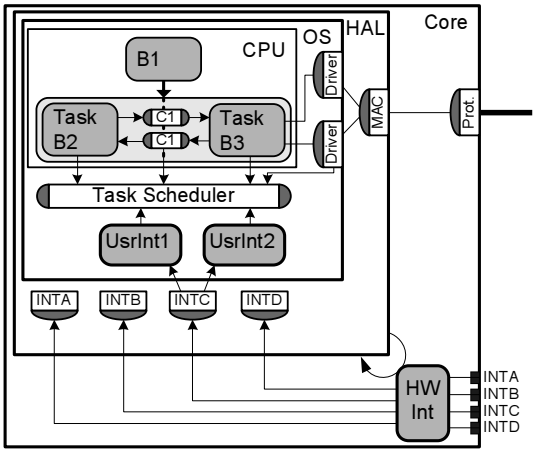

Fig. 7. Processor Bus Functional Model.

Tab. I summarizes the features we capture in our processor model. It also indicates at which level each feature is introduced. The most abstract model at the application level implements only a single feature. On the other hand, the ISS reference realizes all listed features.

TABLE I
SUMMARY OF MODEL FEATURES.

Features	Level
Target approx. computation timing	Appl.
Task mapping, dynamic scheduling Task communication, synchronization	Task
Interrupt handlers, low level SW drivers	Firmware
HW interrupt handling, int. scheduling	TLM
Cycle accurate communication	BFM
Cycle accurate computation	BFM - ISS

V. Experimental Results

To validate our abstract processor modeling approach, we will now separately analyze each function level to determine their effects with respect to simulation speed and timing accuracy. We have applied our approach to an industrial strength telecommunication example: the GSM 06.60 [8] voice encoding and decoding algorithm. We mapped the application to a Motorola DSP 56600 [13] architecture, as shown in Fig. 8.

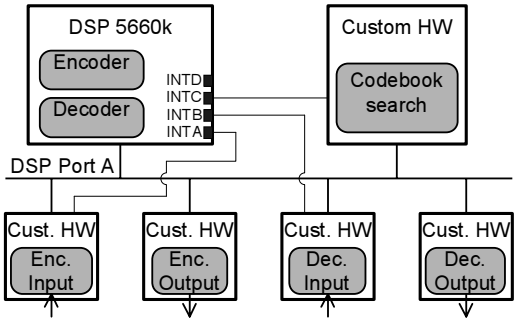

Fig. 8. Example Architecture.

The DSP executes separate tasks for the encoder and the decoder. We use a primitive custom task scheduler that executes priority based scheduling of the two tasks (the encoder with lower priority than the decoder) and uses interrupts for context switching. In the target architecture, the time critical function of the codebook search is mapped to a custom hardware block for increased performance. Four additional custom hardware I/O blocks perform the input and output of the speech frames.

A. Setup

Following our abstraction approach, we have incrementally implemented the processor model with the outlined feature levels using an SLDL[1]. We have measured all feature levels executing on a Sun Fire V240 with a UltraSPARC IIIi processor running at 1.5GHz. For all tests, we use a common data set of 163 speech frames. The hardware blocks are simulated at the behavioral level with cycle approximate timing.

For our accuracy analysis, we focus on the simulated timing. Generally, the error in simulated timing may stem from two main sources: first, the feature abstraction in modeling of the processor, and second, the error due to inaccurate or too coarse grain profiling and back annotation of task execution delays.

In this paper, we focus on the processor model and related inaccuracies due to the feature abstraction. To isolate processor features from task delay errors, we back-annotate accurate execution timing obtained by executing the application on a cycle accurate ISS. While this approach for the timing back annotation is not efficient for exploration of new designs, it allows us to separate the quality of processor modeling. An automatic target specific profiling of user code [6] is outside the scope of this paper and considered as an input as described in the problem definition.

[1] We used SpecC[9] for our experiments. The concepts, however, should be equally applicable to other SLDLs, like SystemC.

We analyze each feature level of our processor model focusing on two aspects: since the main goal of abstraction is to increase the simulation speed, we will first examine the performance. Second, we analyze the loss in accuracy as a side effect of abstracting features. Combining both, we can evaluate the quality of our processor abstraction.

B. Performance Analysis

Fig. 9 shows the simulation time for each feature level when encoding and decoding the test data set of 163 speech frames.

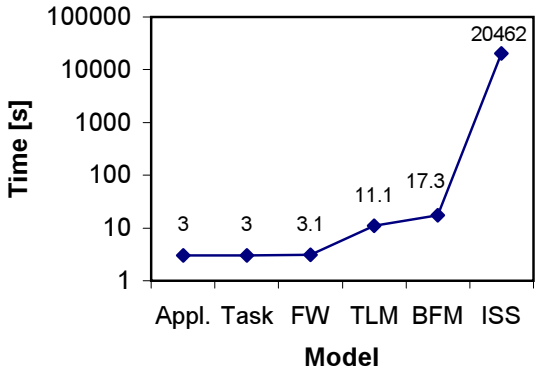

Fig. 9. Simulation time per model.

The measurements confirm the expectation of an increase in simulation performance with abstraction. In other words, the simulation time increases exponentially with the amount of modeled features. Modeling only the *application* is by far the fastest. Including the interrupt structure, as introduced by the *firmware* feature level, increases the simulation effort only minimally. Adding interrupt modeling and communication as performed by the *TLM* and the *BFM* increases the simulation time by a factor of four. The *ISS*-based cycle accurate simulation of our reference model is by far the slowest, about 1800 times slower than the TLM. Tab. II shows the detailed numerical results.

In summary, our proposed model, the TLM, is four orders of magnitude faster than the ISS based co-simulation. Eliminating the abstract feature of hardware interrupt handling and scheduling results in another fourfold speed increase.

C. Accuracy Analysis

In the previous section, we have quantified the significant speedup of our abstract models. Now, we will evaluate the accuracy limitations as a trade-off for achieving these high simulation speeds.

We focus on the timing accuracy and use the metric of the simulated delay per individual speech frame. While executing the model under test, we record the simulated delay for each frame and compare it against the reference model with the cycle accurate ISS to calculate the timing error. For this paper, we define the error in simulated frame delay as a percentage error over the reference model:

$$\begin{aligned} d_{ISS} &: \quad \text{frame delay in ISS simulation} \\ d_{test} &: \quad \text{frame deleay in model under test} \\ error_i &= 100 * \frac{|d_{test} - d_{ISS}|}{d_{ISS}} \end{aligned} \tag{1}$$

Given this definition, an accurate model exhibits 0% error.

TABLE II
EXPERIMENTAL RESULTS FOR TIMING AND ACCURACY.

	Appl.	Task	FW	TLM	BFM	ISS
Sim. Time [s]	3.0	3.0	3.1	11.1	17.3	20462
Speedup over ISS	6821	6821	6601	1843	1183	1
Avg. Enc. Error	20.3%	8.8%	8.5%	3.8%	3.7%	0.0%
Avg. Dec. Error	2.4%	2.4%	2.6%	2.0%	2.3%	0.0%

Tab. II shows the measurement results in detail for both the encoder and the decoder. Fig. 10 shows the average error in the transcoding frame delay for each model. It also includes error bars for the maximal and minimal observed error. For the analysis, we have chosen the transcoding frame delay, the summation of the encoding delay and the decoding delay individually per frame. It expresses the overall accuracy in the system simulation, including effects of dynamic scheduling and external communication.

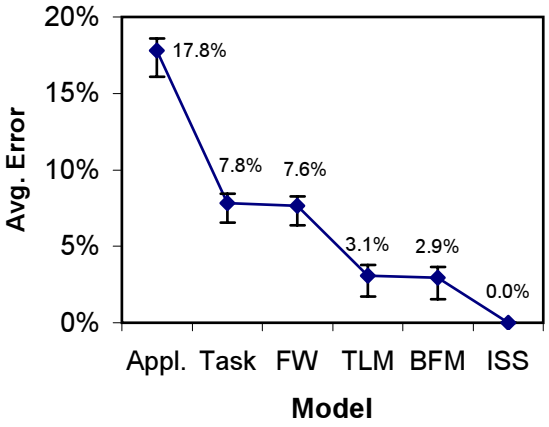

Fig. 10. Average error in simulated encoding delay per frame.

The results indicate that a loss in accuracy has to be accepted due to feature abstraction. The observed error reduces when including more features.

Modeling the *application* only yields the most inaccurate results with 18% error. Including task scheduling, as done in the *task model*, achieves the largest accuracy improvement and reduces the timing error by 10% down to 8%. This improvement is especially pronounced in the encoder, the low priority task (see Tab. II), since it includes preemption influences from all higher priority tasks. For the high priority task of the decoder, on the other hand, we measured no improvement.

Adding low level drivers and the concept of interrupts for external communication only marginally improves accuracy, since the *firmware model* still executes interrupts concurrently. Including proper hardware interrupt handling and interrupt scheduling, as we propose in our *TLM*, reduces the error down to 3%. This model includes suspension of the main software thread on execution of the interrupt handler, and it also models the actual interrupt delays.

Increasing the accuracy of communication, as done in the

BFM variant, does not significantly increase the accuracy since the target architecture uses a single bus master only. As analyzed in [15], a coarse grain abstract communication simulation can already be accurate under the absence of bus contention.

The execution on the cycle accurate ISS yields a timing accurate simulation. Data dependencies can influence the execution timing. As indicated by the error bars for each of the model stages, the timing accuracy varies by 2.1%, caused by data dependencies in the code execution. Due to the timing annotation at the function level such effects are not completely captured in our model. Furthermore, although we have used accurate cycle information from the reference model, our back annotation is only accurate for leaf functions that do not call other functions. The overhead due to the function call hierarchy is not yet adequately reflected. Finally, the execution of the task scheduler contributes to the timing inaccuracy. In our model, we have not modeled the overhead of context switches and the delays due to the custom scheduler.

VI. Summary and Conclusion

In this paper, we presented our abstract model of embedded processors. We used a layered approach, incrementally describing our processor modeling with essential features of task mapping, dynamic scheduling, interrupt handling, low level firmware and hardware interrupt handling.

We validated our abstraction approach using an industrial strength example of a GSM 06.60 speech transcoder mapped to a Motorola DSP plus external IPs. We applied our processor modeling approach with its five feature levels, and we analyzed each level with respect to the gain in simulation performance and the loss in accuracy.

Our results show the tremendous benefits of our proposed processor model. Based on the analysis, we identified two main feature levels. Our firmware feature level, which covers the complete software including low level drivers and interrupts, executes 6600 times faster than the traditional ISS based co-simulation approach with an error of less than 8%. It is suitable for early rapid design space exploration. For an even higher accuracy, e.g. for system validation, our TLM feature level can be used. It simulates 1800 times faster than the ISS with an error of less than 3%.

In future work, we plan to extend our concept to general purpose processors that include peripherals, like a programmable interrupt controllers. Furthermore, we plan to add a more detailed model of a full Real-Time Operating System and further improve the timing accuracy.

References

[1] Advanced RISC Machines Ltd. (ARM). SoC Developer with MaxSim Technology. http://www.arm.com/products/DevTools/MaxSim.html.

[2] L. Benini, D. Bertozzi, A. Bogoliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *Journal of VLSI Signal Processing*, 41(2):169–184, 2005.

[3] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, and A. Jerraya. Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Shanghai, China, January 2005.

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

[5] L. Cai and D. Gajski. Transaction Level Modeling: An Overview. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, CA, October 2003.

[6] L. Cai, A. Gerstlauer, and D. D. Gajski. Retargetable Profiling for Rapid, Early System-Level Design Space Exploration. In *Proceedings of the Design Automation Conference (DAC)*, San Diego, CA, June 2004.

[7] CoWare. Virtual Platform Designer. http://www.coware.com.

[8] European Telecommunication Standards Institute (ETSI). *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding*, final draft edition, 1996. GSM 06.60.

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

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

[11] 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.

[12] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Jan. 1999.

[13] Motorola Inc., Semiconductor Products Sector, DSP Division. *DSP56600 16-bit Digital Signal Processor Family Manual*, 1996. DSP56600FM/AD.

[14] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Proceedings of the Design Automation Conference (DAC)*, Anaheim, USA, June 2003.

[15] G. Schirner and R. Dömer. Quantitative Analysis of Transaction Level Models for the AMBA Bus. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2006.

[16] VaST Systems. VaST tools and models for embedded system design. http://www.vastsystems.com.