Fast and Accurate Processor Models for Efficient MPSoC Design

Gunar Schirner
University of California, Irvine
Andreas Gerstlauer
University of Texas at Austin
and
Rainer Dömer
University of California, Irvine

With growing system complexity and ever increasing software content, the development of embedded software for upcoming MPSoC architectures is a tremendous challenge. Traditional ISS-based validation becomes infeasible due to the large complexity.

Addressing the need for flexible and fast simulating models, we introduce in this paper our approach of abstract processor modeling in the context of multi-processor architectures. We combine modeling of computation on processors with an abstract RTOS and accurate interrupt handling into a versatile, multi-faceted processor model with several levels of features.

Our processor models are utilized in a framework allowing designers to develop a system in a top-down manner using automatic model generation and compilation down to a given MPSoC architecture. During generation, instances of our processor models are integrated into a system model combining software, hardware and bus communication. The generated system model serves for rapid design space exploration and a fast and accurate system validation.

Our experimental results show the benefits of our processor modeling using an actual multi-processor mobile phone baseband platform. Our abstract models of this complex system reach a simulation speed of 300MCycles/sec within a high accuracy of less than 3% error. In addition, our results quantify the speed/accuracy trade-off at varying abstraction levels of our models to guide future processor model designers.

Categories and Subject Descriptors: I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms: Design

Additional Key Words and Phrases: processor modeling, system level design, TLM, transaction level model, performance prediction/estimation, Multi-Processor System-on-Chip, MPSoC

Author's addresses: Gunar Schirner, Center for Embedded Computer Systems, UC Irvine, Irvine, CA 92697-2625, USA; email: hschirne@uci.edu; Andreas Gerstlauer, Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712; email: gerstl@ece.utexas.edu; Rainer Dömer, Center for Embedded Computer Systems, UC Irvine, Irvine, CA 92697-2625, USA; email: doemer@uci.edu

1. INTRODUCTION

As system complexity increases in general, the software content in current and future system designs grows even more so. Recent state-of-the art MPSoC system architectures increasingly employ a large number of processors. Validation and debugging of embedded software for such systems is becoming a tremendous challenge, and more and more infeasible with current methods.

Transaction Level Modeling (TLM) 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 MPSoCs, allowing adaptation to emerging standards and specific customer demands.

Traditionally, embedded software is validated and debugged using Instruction Set Simulators (ISSs) which provide functional and timing accurate simulation on a host platform at a very fine granularity. However, interpreting ISSs simulate very slowly, especially when multiple instances are integrated into a MPSoC system simulation. Therefore, ISS-based validation and debugging is not sufficient to match the needs for rapid design space exploration at the system level. Thus, a higher level of abstraction is needed.

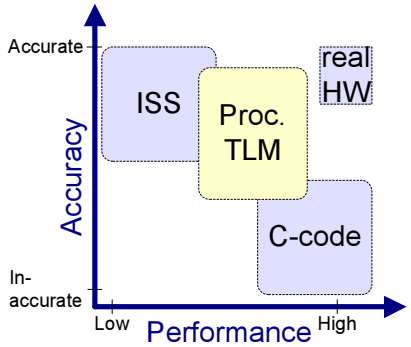


Fig. 1. Trade-off in system simulation.

Abstracting a software execution environment results in a trade-off between simulation performance and simulation accuracy as shown in Figure 1. Using an ISS yields an accurate but slow simulation. A purely functional C-code execution that does not express any target performance metric is on the other extreme. It shows a high simulation performance, however, is very inaccurate in structure and timing. A model, such as a processor TLM, is desirable which abstractly reflects the processor characteristics, simulates fast and yet shows sufficiently accurate timing results.

In this article, we present our approach of abstracting the software execution environment while retaining a high accuracy of the models. In particular, we describe our abstract processor modeling as an integral part of current MPSoCs. Our processor models provide a high-level software execution environment with

accurately timed execution, real-time operating system (RTOS) services and external communication. Our models exhibit highest simulation performance while maintaining an acceptable accuracy in simulated timing, and exhibiting the actual structure of the software architecture (e.g. drivers and interrupts). Using our abstract processor models in conjunction with transaction-level models (TLMs) of communication [Schirner and Dömer 2008] dramatically increases the execution speed in a co-simulation environment. With high accuracy in timing, our models enable an early functional and fast simulation of the desired target architecture, clearly exposing the implications of programming and architectural decisions, and thus allowing rapid validation, debugging and design space exploration.

1.1 Problem Definition

With the importance of embedded software within an MPSoC design, fast and accurate simulation of embedded software is needed. The traditional ISS based co-simulation satisfies all functional requirements, however at a very low speed. Faster simulation capabilities are needed to aid efficient exploration, validation and debugging.

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 properly reflecting the structure of the software architecture (e.g. drivers and interrupts).

We focus on a Multi-Processor SoC (MPSoC) target architecture as outlined in Figure 2. It consists of a set of processors, where each processor is connected to a processor specific main bus. We assume that each processor contains an internal memory, which stores the execution binaries and local variables. Additionally, we associate a customizable Programmable Interrupt Controller (PIC) and a timer with each processor.

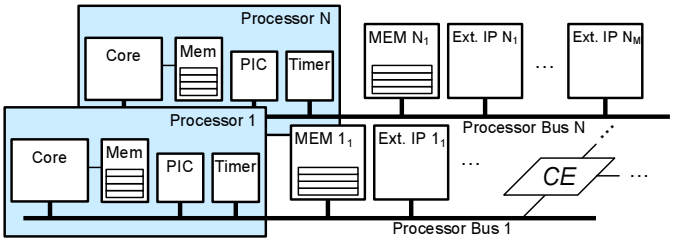


Fig. 2. Generic MPSoC target architecture.

Each processor communicates with external memory (holding globally shared variables) and with custom hardware IP blocks (through memory mapped I/O and interrupts) over the processor main bus. Any number of IP blocks and memories may be connected to each bus. A processor can also communicate with other

processors and external IPs or memories connected to other busses through one or more Communication Elements (CEs)[1], such as a bridges or a routers.

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 and explicit communication between tasks. Furthermore, we assume that data about execution delays for each task is available. We make this assumption to segregate the two main error sources for modeling software execution: (a) target specific user code profiling and back annotation, (b) errors due to processor feature abstraction (see Section 4.1.2). In this article, we focus on the processor modeling and therefore are interested only in the error due to feature abstraction. The equally important target specific user code profiling is outside of the scope of this publication.

1.2 Outline

The article is organized as follows: after a brief introduction of related work in Section 1.3, we will outline our overall MPSoC development approach in Section 2. Section 3 then describes our abstract processor modeling approach in detail by incrementally describing its features. In Section 4, we validate our modeling approach using a set of industrial-strength MPSoC examples. We analyze the simulation performance and the achievable accuracy in detail. Finally, we conclude the paper with a summary in Section 5.

1.3 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 [Grötker et al. 2002] and SpecC [Gajski et al. 2000]. The languages provide means to describe systems, but by themselves do not offer any modeling solutions.

Using TLM [Grötker et al. 2002; Ghenassia 2005; Montoreano 2007] 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 only later gained attention. Bouchhima et al. [2005] 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. [2005] 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. Furukawa et al. [2007] introduce a HW/SW cosimulator which uses a host compiled RTOS approach. To simulate a target processor, the target RTOS is compiled to run natively on the simulation host and the user application executes on top of that RTOS. Each simulated processor (target RTOS) runs in an own process and communication is realized through a backplane tool which also facilitates communication with SystemC modules and RTL hardware simulators.

[1]In extension to what is shown in Figure 2, we assume that the busses may be arranged as a hierarchy of busses.

The approach includes simulation of interrupts, however does not reflect target processor execution timing.

At the very high abstraction level of application modeling, Ptolemy [Buck et al. 1994] 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 ISS based co-simulation is provided by several commercial vendors, such as ARM's SoC Designer with MaxSim Technology [ARM], VaST Systems' virtual system prototyping tools [VaST], CoWare's Virtual Platform Designer [CoWare], ARC's xISS [ARC], and Virtutech's Simics [Virtutech]. Typically, commercial ISS vendors advertise simulation speeds of up to 200+MIPS on off-the-shelf PCs. In addition, ISS based co-simulation is also used in many academic projects, such as the MPARM [Benini et al. 2005] platform. To improve the speed of co-simulation with multiple ISS instances, Yi et al. [2007] propose a trace-driven virtual synchronization which greatly reduces the synchronization overhead between ISS instances. Significant research effort has been invested to improve ISS performance [Nohl et al. 2002; Mong and Zhu 2004; Reshadi et al. 2009]. Nohl et al. [2002] present just-in-time cache compiled simulation a hybrid between compiled ISS and interpretive ISS, which maintains the performance of a compiled solution while reducing the large memory requirements associated with a purely compiled approach. The presented approach reached 8 MCycles/sec on an 1.2 GHz Athlon. Reshadi et al. [2009] demonstrate with a similar hybrid between compiled ISS and interpretive ISS a performance of 12 MCycles/sec on a 1 GHz P3. Conversely, our solution does not aim for instruction accuracy and reaches up to 600 MCycles/sec (see section Section 4.2), still one magnitude faster after correcting for the simulation platform.

A newer research direction is a hybrid simulation approach that alternates between ISS-based simulation and natively compiled execution. One example is HySim [Gao et al. 2008], which can switch between binary interpretation in an ISS and native execution to gin simulation performance. For the hand-over, it keeps a consistent memory content between the two versions. Since the approach includes a binary interpretation, it can also simulate 3rd party binary code, as well as assembly. An ISS-based simulation within HySim of a MIPS32 core averages with 2.7 MIPS, while the native execution averages with 185MIPS. On the other hand, our abstract processor simulates at higher speeds, as it does not require the synchronization overhead with the ISS.

In previous work [Schirner et al. 2007], we have introduced our processor modeling for a single processor system and have shown a simple DSP example application. In this article, we extend the concepts to multi-processor systems. We also add abstract modeling of an RTOS, and support Programmable Interrupt Controllers (PIC). Furthermore, we present more detailed descriptions. Finally, our experimental results stem from a wider range of applications, showing the more general applicability and benefits of our approach.

2. CONTEXT: OUR MPSOC DEVELOPMENT APPROACH

Before we describe our abstract processor modeling, we will show in this section how the models are used in our electronic system-level (ESL) flow. Figure 3 shows our

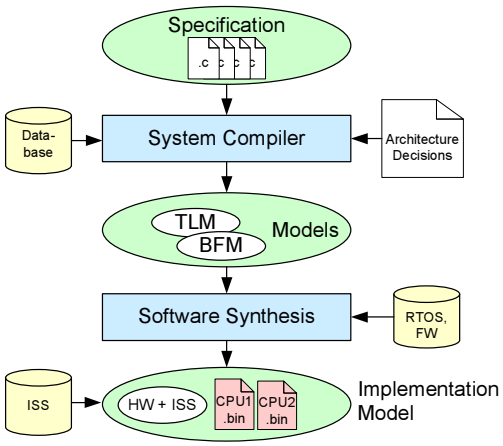


Fig. 3. SCE software development framework.

framework for programming, validation and debugging of embedded software for MPSoCs. At the input, we assume the user application as a set of communicating processes with C code for each task, including information about task relations and inter-process communication (IPC).

We have developed the System-on-Chip Environment (SCE) [Dömer et al. 2008], an ESL framework that takes the application specification and compiles it down to an implementation on a target architecture, creating a binary for each processor in an MPSoC architecture, describing the hardware components, and their connectivity. The designer specifies the architecture by selecting components from the database. The designer also selects a mapping of application tasks and IPC channels to those components.

Our SCE framework automatically generates the processor models that we describe in this article. In a stepwise process, SCE inserts necessary implementations of task scheduling, bus drivers, interrupt handlers, hardware abstraction layers and other firmware. Furthermore, SCE performs back-annotation of task execution delays at the function level for timing feedback during simulation. At its output, SCE generates transaction-level (TLM) and bus-functional (BFM) models of the target platform running the application. In addition, SCE can produce intermediate system models at varying levels of detail.

The generated models are then used as an input to the software synthesis back-end [Schirner et al. 2009]. This automatically synthesizes the embedded software for each processor down to the final binary image. The software synthesis includes generation of C code for the application tasks, bus drivers and firmware. It targets the code for a chosen RTOS implementation to perform processor initialization, multi-tasking and interrupt handling. The software synthesis furthermore includes cross-compilation and generates a binary for each processor.

The generated images can be directly loaded onto the target platform for execution. They are the final implementation for the software. The back-end synthesis also allows validation of the final target binaries independent of the availability of the hardware platform. For that, the back-end synthesis generates additionally

an ISS-based co-simulation model. This model is based on the system BFM. It describes all hardware and includes one ISS instance per processor.

Although all models are generated automatically by SCE, we do not want to deviate into the synthesis aspect in this article. Instead, we focus on the abstract processor modeling itself.

3. ABSTRACT PROCESSOR MODELING

Our models are captured in a System Level Design Language (SLDL). We chose SpecC [Gajski et al. 2000] for our experiments. The concepts however, are applicable to other SLDSs, i.e. SystemC, as well. The SLDL framework provides a timed, fast, discrete event simulation natively on a simulation host. Our processor models abstract away the processor micro architecture and do not simulate the instruction set architecture (ISA). Instead, they abstractly reflect the behavior of the processor and the software execution environment, including all desired features like RTOS task scheduling, external bus communication, and interrupt handling with processor suspension, interrupt nesting, and interrupt priorities.

Observing SW execution on a typical processor, we can identify several layers of functionality. We have organized our models in layers along features. We will uses these layers to incrementally describe our model. In total, we will show three intermediate models until reaching the actual proposed processor TLM. For our detailed analysis, we will consider the intermediate models, then the TLM, and finally also investigate the accuracy of a processor BFM. We will compare the models to an ISS-based reference model. This detailed analysis, including the intermediate models, will allow us to evaluate each feature individually for its cost in simulation speed and contribution to timing accuracy.

The subsequent sections describe each step of the processor modeling, starting with the innermost layer.

3.1 Application

In order to achieve maximum simulation speed, we execute the *application* natively on the simulation host. This is the innermost layer of our abstract processor, as depicted in in the box *CPU* in Figure 4.

The user code is captured in the SLDL as a hierarchical composition of behaviors separating computation and communication. Behaviors can be flexibly constructed to execute sequentially, in parallel, in a pipelined fashion, or state machine controlled.

Communication is expressed using a set of standardized abstract channels for high-level, typed message passing. These standardized channels offer synchronous and asynchronous communication with the option of one-way, two-way data traffic, or synchronization only. Additionally, behaviors may communicate through shared variables. The connectivity between behaviors (either through channels or global variables) is expressed through ports, where channel/variable instances connect to. As such, our application model captures the three aspects of computation, communication, and connectivity.

For processor modeling, we distinguish between internal communication (tasks on the same processor) and external communication (to other processing elements). The internal communication, as shown with channels *C1* and *C2* in Figure 4, largely

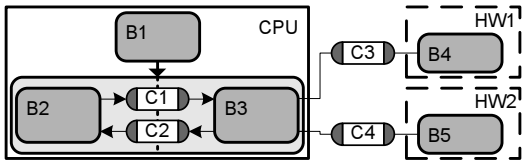


Fig. 4. Application model and external communication.

remains unchanged throughout adding the more detailed layers. The external communication (e.g. as initiated by behavior *B3* through *C3* and *C4*) on the other hand, will be refined throughout the modeling stages to follow a generic portable communication stack [Gerstlauer et al. 2007]. Note that for illustration purposes, Figure 4 additionally shows external hardware components *HW1* and *HW2*, which are not part of the processor model.

At the level of the application model, external communication is the same in functionality as the internal communication (i.e. untimed, without bus connectivity). Global variables, which are shared between processing elements, are accessed directly at this abstraction level without special synchronization or bus communication.

The user application is executed natively on the simulation host inside the discrete event simulation environment to achieve highest simulation speeds. In order to exhibit target-specific execution timing, the application C code is back-annotated with estimated execution timing. In general, the granularity of timing annotation influences the achievable accuracy and the simulation performance. A very fine granularity of back annotating each individual C-operation would yield a high accuracy, since this captures data dependent execution[2]. However, such fine granularity will also result in a large number of wait-for-time statements, therefore slowing down the simulation. On the other hand, a too coarse grain back annotation would not express data dependent execution sufficiently.

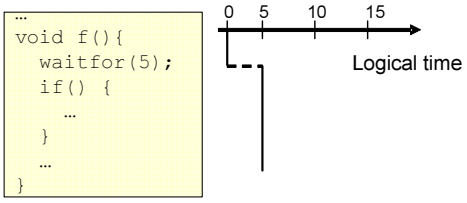


Fig. 5. Timing back annotation.

Timing information can be automatically back annotated (e.g. see [Cai et al. 2005]). In our case, the system compiler inserts wait-for-time statements at the function level (see Figure 5). We chose to back annotate at the function level, since the applications of our examples are sufficiently separated into functions, which already express data dependency and do not have much data dependency below the function level.

[2]Also the preemption simulation improves significantly, which we will discuss in Section 3.2.

3.2 Task Scheduling (OS Kernel)

Concurrent execution of software on an inherently sequential processor requires an operating system on the target. In order to explore the effects of dynamic scheduling decisions, SCE inserts an abstract RTOS model [Gerstlauer et al. 2003] as shown in Figure 6. The abstract RTOS model emulates the dynamic scheduling on top of the SLDL framework. Since it integrates with the simulation environment, it enables high execution performance, thus allowing early exploration of scheduling policy decisions.

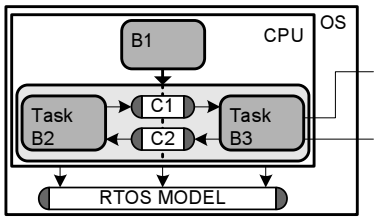


Fig. 6. Task model.

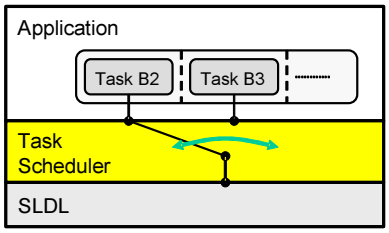


Fig. 7. Abstract scheduler switching between tasks.

In order to execute on top of the OS model, each parallel executing behavior is refined into a task (e.g. *Task B2, Task B2*), defining task execution control and scheduling policy parameters (i.e. task priority). A sequentially executing behavior becomes part of its parent task. For example, behavior *B1* becomes part of the processor main task, since it is only executed on startup. It spawns *Task B2* and *Task B3*.

To simulate the dynamic scheduling, each primitive that potentially triggers scheduling is wrapped to interact with the abstract RTOS model. Such primitives include task start, channel communication, and wait-for-time statements.

As a result, the abstract RTOS model controls each task's state and decides the task scheduling specific to the selected scheduling policy (e.g. priority based or first come first serve (FCFS)). It switches between the tasks (Figure 7) so that at any point only a single task is running on top of the underlying SLDL. Our OS model allows to observe the number of context switches, the performance effects of the selected scheduling policy, and the idle time for each processor in the MPSoC system.

3.3 Firmware (External Communication)

In addition to the OS kernel, the processor firmware has to include the necessary drivers for all external communication. Figure 8 shows the model extended with a firmware representation and adds an additional layer, the Hardware Abstraction Layer (HAL). External communication is first refined to untyped streams (captured in the half channels labeled *Net*). These data streams perform data typing (e.g. adjusting for differences in endianess or data size) between processing elements and therefore enable communication in a heterogeneous system. Each abstract channel in the model is refined to a network layer link, which marshals the user data into

the untyped byte stream. SCE automatically generates the marshalling code depending on the user data types. Similarly, external communication through shared variables (e.g. memory mapped IO) is wrapped into proper channel communication, performing the data marshalling.

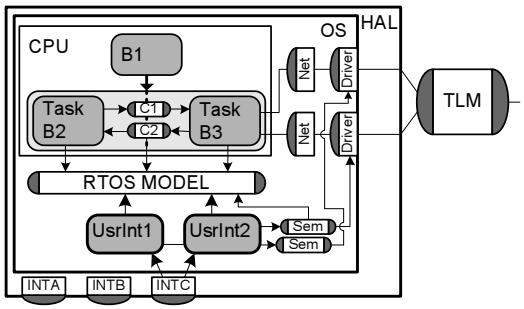


Fig. 8. Firmware model.

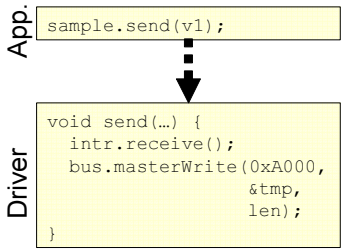


Fig. 9. Example of inserted driver code for synchronization.

In addition to the data typing, SCE generates low level software drivers that implement external bus communication and synchronize with external sources. The *Driver*, implemented as a half channel, adds system absolute addressing and maps the external communication to the processor bus (see an example driver code in Figure 9). The processor bus is modeled at the transaction level. The processor bus TLM simulates the bus at a granularity of user transactions, which are arbitrary sized blocks of data.

The selection of synchronization is an important issue for embedded system design. The type (e.g. interrupt or polling) and the granularity of synchronization significantly influence system performance. The *firmware* level adds modeling of the external synchronization to allow early exploration of the synchronization options.

Depending on the designer's decisions, the system compiler generates the low level drivers to synchronize using polling or interrupts. In case of polling, the generated driver code checks the external status with a designer selectable polling period. In the latter case of interrupt synchronization, the system compiler generates an interrupt handler that is registered to the external interrupt source.

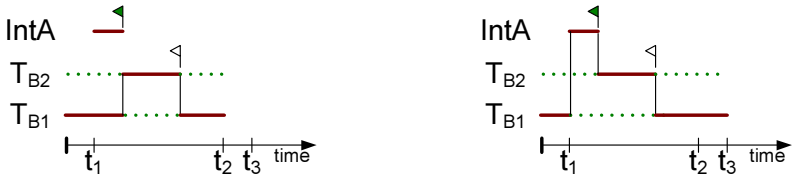
To simulate interrupt handling, the firmware model captures the interrupt inputs of a programmable interrupt controller (PIC) connected to the processor (*INTA* - *INTC* in Figure 8). This model abstracts away the actual processor interrupt for simulation speed. Interrupt inputs are implemented as channels and can be triggered directly by external sources (such as a slave). The interrupt input channel then triggers the execution of one or more user interrupt handlers. The example in Figure 8 shows the option of interrupt sharing, were two user interrupts share the same interrupt input. Here, each user interrupt handler contains additional code to determine its interrupt source. Finally, the user interrupt handler uses a standard channel (i.e. semaphore) to release its driver after successful synchronization.

The firmware model is the first to contain the complete software (including the driver and interrupt code). The firmware layer marks the boundary between software implementation and hardware features of the processor.

3.4 Processor Transaction Level Model

To complete our processor model a description of the processor hardware is needed. This includes hardware interrupt handling, interrupt scheduling, timer and PIC hardware, and simulation of bus accesses at bus transaction granularity.

So far (i.e. the firmware model), an interrupt triggering HW directly calls the interrupt handler. Thus, interrupts are not scheduled and may execute concurrently to the user application. An example of unscheduled interrupt handling is shown in Figure 10(a). The tasks *B1* and *B2* are scheduled by the abstract RTOS model. However an incoming interrupt at t_1 executes in parallel to *B1*, which is not realistic for a real processor. Hence, the model finishes the sequence incorrectly early at t_2 instead of t_3.



(a) Unscheduled. (b) Scheduled.

Fig. 10. Hardware interrupt scheduling.

Our processor TLM, shown in Figure 11, reflects the additional hardware needed for a scheduled interrupt handling. It models the actual interrupt chain of the processor. It represents external interrupt wires, which are monitored by a PIC and multiplexed into the processor interrupt signal(s). An interrupt behavior *HW Int* inside the processor monitors its interrupt input(s) and triggers execution of the system interrupt handler. The system interrupt handler, in turn, communicates with the PIC to determine the actual interrupt source and executes the appropriate user interrupt handler.

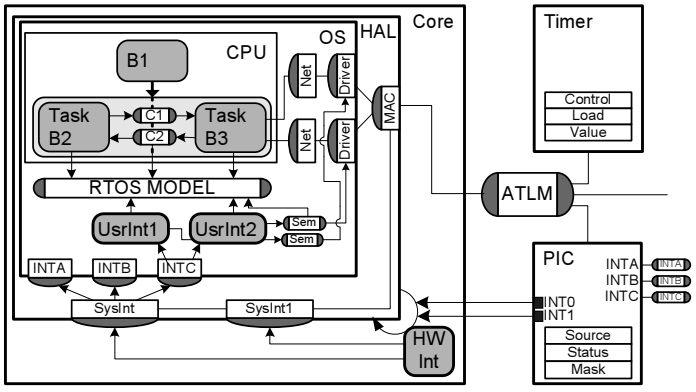


Fig. 11. Processor Transaction Level Model.

Our processor TLM offers interrupt scheduling to avoid that interrupts are executed in parallel with application tasks. The behavior *HW Int* monitors the processor interrupt line(s). Upon occurrence of an interrupt, *HW Int* suspends the processor simulation - all SW behaviors inside the HAL shell (as indicated by the half circle around the bottom right corner of the HAL shell). Therefore all tasks of that processor are now suspended.

The behavior *HW Int* implements the HW interrupt scheduling and calls the system interrupt handler *SysInt* via a channel call. To handle multiple occurring interrupts, the *HW Int* observes interrupt priorities and nesting for an accurate scheduling. Additionally, it provides interrupt related control registers to the driver software, e.g. for enabling and disabling interrupts.

The system interrupt handler communicates through the bus interface with the PIC determining the incoming external interrupt. The system interrupt handler releases through the interrupt channel (e.g. *INTC*) the actual user interrupt (e.g. *UserInt2*). The user interrupt handlers are modeled as high priority tasks inside the abstract RTOS model. At this point, the hardware and low level interrupt handling is completed. The call chain, initiated by the *HW Int*, terminates and the HAL shell is released from the suspended state. Next, the abstract RTOS model schedules the highest priority ready task, the user interrupt handler *UserInt2* in this case.

As a result, the complete interrupt chain is replicated for an interrupt execution of a processor and is modeled closely to existing hardware. This allows us to accurately simulate the interrupt execution overhead and provides proper interrupt scheduling. The effects of the scheduled interrupts are shown in Figure 10(b). In constrast to the unscheduled version, the interrupt handler *IntA* now preempts the execution of all tasks. The simulation sequence in this example now correctly ends at t_3.

External communication in the processor TLM is modeled by a transaction level model of the bus, providing a cycle approximate simulation with user transaction granularity.

3.5 Processor Bus Functional Model (BFM)

To obtain a bus-functional variant of our processor model, we employ a pin- and cycle-accurate model of the processor bus interface. The modeling of computation is identical to the TLM. The communication, on the other hand, is further refined. SCE introduces a bus-specific Media Access Layer (MAC) that splits the arbitrary sized user transactions into bus transactions (bus protocol primitives). The BFM includes protocol channels that provide bus access with a bus transaction granularity and implement each bus interface pin accurately. Channels implement state machines to realize the bus protocol to drive and sample the individual bus wires. Additional active components (such as arbiter, multiplexer) are inserted if needed to accurately implement the bus protocol. Using the BFM allows us to attach RTL components and to create signal traces of the bus traffic (Figure 12) for debugging purposes.

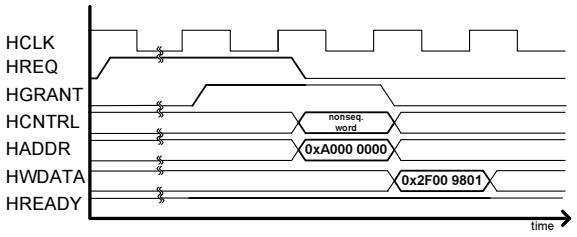


Fig. 12. Bus trace in BFM.

3.6 ISS-based Cosimulation Model

In order to validate SW binaries and for a detailed cycle-accurate SW execution, we use an Instruction Set Simulator (*ISS*) based reference model. This model includes a behavioral model of all hardware components and an ISS instance for each processor in the system. Each ISS is wrapped to adapt the ISS API to the simulation environment, as shown in Figure 13.

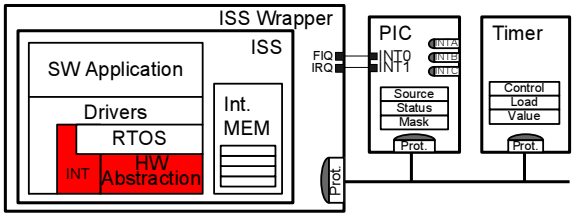


Fig. 13. Bus Functional Model with ISS.

We use ISSs that are available in library form with a cycle callable interface. Each ISS is called within an own wrapper module that executes the ISS cycle-by-cycle. The wrapper uses the ISS API to detect ISS external memory accesses and translates them to transactions on the simulated bus. It also listens to incoming interrupt signals to the processor and forwards them to the ISS. An ISS typically includes internal memory for storing the executed binaries and local variables.

From the software perspective, the ISS-based model represents the real implementation. It executes the final binaries, accesses external hardware through the bus, and allows interrupt-based synchronization.

3.7 Model Benefits

To summarize our layered processor model, Table I lists the features captured and 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.

Our most abstract model, the *application* model, allows with the target approximate computation timing an early functional validation of the specification. It exposes the hierarchical task graph and allows observing of data and control flow across multiple processors.

Table I. Summary of model features.

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

With the *task* model, additionally the effects of dynamic scheduling are observable in simulation. It enables selecting a proper scheduling scheme and priority distribution for all tasks in the system. It is the most abstract model that enables monitoring of the processor utilization. It therefore is important for design decisions regarding load balancing within a multi-processor system.

The *firmware* model completes the modeling of software. It shows the synchronization across the network, interaction between interrupt handlers and tasks. This model enables firmware and driver debugging at a high-level.

The *TLM* adds a detailed model of the interrupt chain. The designer therefore can accurately observe the interrupt latencies for each processor and the overall interrupt overhead in the processor load. Furthermore, the designer can validate the efficient selection of the synchronization scheme and the interrupt priority assignment across the complex MPSoC. This model provides a detailed view of the busses in the system (e.g. in terms of utilization, contention, and access latency). With this rich detail, the *TLM* serves as a validation platform with a high confidence level.

The *BFM*, in turn, allows observing all bus traffic in pin and cycle accurate detail. This exposes all communication details and provides observability beyond what is typically feasible in real hardware. Furthermore, the *BFM* enables the integration of RTL-only IP components. Finally, the cycle-accurate *ISS-based* model allows validation of the software binaries before availability of the final hardware. It provides very fine-grained performance measurements of the system.

4. EXPERIMENTAL RESULTS

To validate our approach for abstract processor modeling, we will now separately analyze the output of each modeling layer to compare the effects with respect to simulation speed and timing accuracy.

We have applied our approach to an industry-strength cellphone example, including three subsystems: a GSM 06.60 voice transcoder [ETSI 1996], a MP3 audio decoder, and a JPEG image compressor [Bhaskaran and Konstantinides 1997]. We have mapped these applications to a MPSoC platform (Figure 14) similar to the baseband platform used in the RAZR cellphone [Giridhar 2005]. This platform includes two processors, an ARM7TDMI [ARM 2001] and a Motorola DSP56600 [Motorola 1996] connected by a transducer. The ARM processor executes cellphone control tasks, MP3 decoding, and JPEG encoding, using the MicroC/OS-II [Labrosse 2002] RTOS for priority scheduling. The DSP is dedicated to GSM

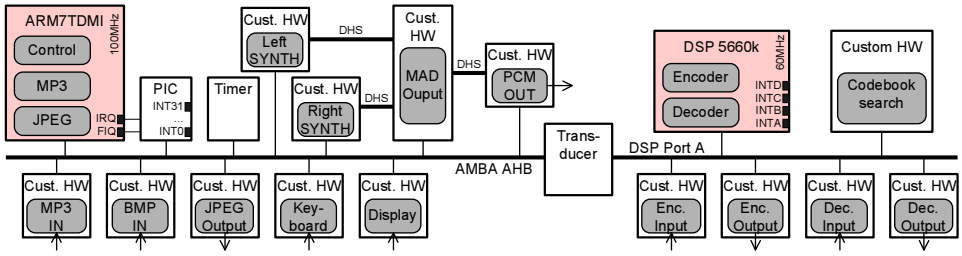


Fig. 14. Example cellphone architecture.

transcoding. Its encoding and decoding tasks are scheduled by a custom priority-based scheduler.

Computation intensive operations, such as the MP3 synthesis filter and the GSM codebook search, are mapped to custom hardware accelerators. Ten additional hardware units perform I/O operations, including the keypad and display interfaces to the user.

4.1 Setup

Following our modeling approach, we have specified the phone application and stand-alone versions of each subsystem in the SpecC SLDL for our experiments and used our SCE framework to automatically generate the individual models. Please note that our SCE framework [Dömer et al. 2008] is separated into individual compilation stages. We have simulated all[3] models on a 3.0GHz Pentium IV running Linux Redhat Enterprise 3. For all tests, we use the same input data consisting of GSM voice samples, a MP3 audio stream, and a BMP picture. To balance the impact of each application, each input data set is sized to represent 1.5 seconds of real-time; voice/audio samples (for MP3, GSM) and computation time (for picture encoding). The hardware blocks are simulated at behavioral level with cycle-approximate timing.

We use the ISS-based model for cycle-accurate reference execution of software. We have integrated the ISS SoftwareARM (SWARM) [Dales 2000] for simulating the ARM7TDMI and a Motorola proprietary ISS for simulating the DSP.

We analyze our models focusing on two aspects: the *performance*, since an increased speed is the main goal of abstraction, and the loss in *accuracy*, as a side effect of abstracting features. Combining both, we can quantitatively evaluate the quality of our processor abstraction. We now describe our performance and accuracy metrics.

4.1.1 *Performance Metrics.* We report the performance measurements using three metrics. First, we measure and report the simulation time. This is the time it takes to simulate our test data set on the simulation host. Second, we then use the simulation time to compute the speedup over the ISS-based simulation. This

[3]Note the Motorola proprietary ISS for the DSP56600 was only available for Sun SPARC Solaris. Hence, we have executed the ISS models that include the DSP56600 on a Sun Fire V240 with a 1.5GHz UltraSPARC IIIi processor. To estimate a comparable simulation time on the Pentium VI, we have scaled the measured simulation time (divided by 1.42) according to the performance ratio between the processors using the SPECint2000 benchmark.

indicates the relative improvement over the reference solution. Finally, we calculate the number of cycles simulated per second, providing an absolute reference. For that, we first measure the number of cycles when using the cycle-accurate ISS, and then divide by simulation time of the abstract model. Please note that this metric does not include the cycles computed by hardware.

4.1.2 *Accuracy Metrics.* Accuracy can be segregated into many aspects. For the purpose of this article, we will distinguish three aspects of accuracy: functionality, represented feature detail, and timing. For our analysis, we will fix the first, vary the second, and measure the third aspect.

The first aspect of functional accuracy requires that the model executes the algorithms described in the specification and that the correct data is received by external hardware components. Since this is a necessary requirement for a functional simulation, all our models are functionally accurate. The second aspect is the represented detail level, which states how many features of the processor are actually present in the model. We vary this aspect in our model under test, ranging from only a timed execution (as in the application model) to the pin accurate processor model BFM. The third aspect is the accuracy in timing which we measure in our tests.

Within the aspect of timing accuracy, we can identify two main sources of possible errors. One, the timing back annotation of the user code and two, the presence of modeled features in the actual processor model. Both contribute to the timing accuracy of a model. However, it is difficult to measure them separately.

In this paper, we focus on the processor models and their inaccuracies due to feature abstraction. Target-specific profiling of user code and the timing back annotation are outside the scope of this paper. We therefore minimize the effect of the timing back annotation by using the most accurate available numbers. Therefore, the accuracy measured in our tests reflects only the errors induced by feature abstraction. We obtain accurate timing information by simulating the application on a cycle-accurate ISS. We use the same input data for obtaining the execution timing via the ISS, and for measuring the accuracy of our abstract models. As stated before, we back-annotate at the function level. Note that the ISS-based approach for the timing back-annotation is not efficient for exploration of new designs. It, however, allows us here to better examine the quality of our processor modeling.

To calculate the timing error in our models, we use the simulated delay per individual frame (speech frame for the GSM, PCM frame for the MP3, and stripe for the JPEG). While executing the model under test, we record the simulated delay for each frame and compare it against the ISS-based reference model. For this paper, we define the error in simulated frame delay as a percentage error over the reference model:

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

Given this definition, accurate models exhibit 0% error. We determine the error for each individual frame, and report the average error over all frames for each subsystem.

For our analysis, we first focus on performance and then on accuracy. We will discuss these metrics separately for each subsystem within our example to illustrate the application and platform dependency. Later, we analyze the entire system combining performance and accuracy measurements.

4.2 Performance Analysis

The achievable performance depends significantly on the system setup, especially on the balance between computation in software and in hardware. Furthermore, the amount of bus communication and its complexity affects performance. To isolate the impact of the system composition, we analyze three sets of systems. First, we examine software-only solutions of each subsystem. Second, we investigate hardware-assisted versions of each subsystem. Finally, we will analyze the complete cellphone system as outlined before. These respectively correspond to traditional simulation using a single ISS, co-simulation of a single ISS with additional hardware, and co-simulation of multiple ISS's in a system simulation.

4.2.1 *Performance of SW-only Systems.* In the software-only version of each subsystem, all computation is performed on the processor. Input and output are mapped to external hardware components to maintain a realistic test. These I/O blocks, however, perform only a negligible amount of computation. Therefore, the

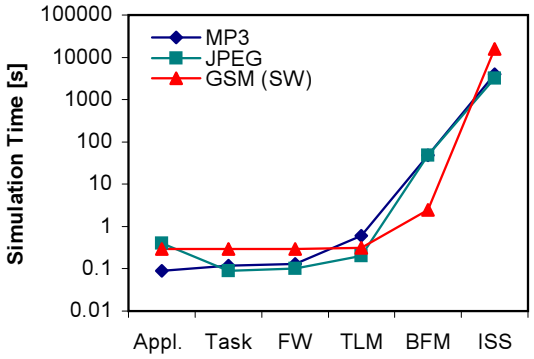


Fig. 15. Simulation time for SW-only systems.

Table II. Simulation performance of SW-only systems.

		Appl.	Task	FW	TLM	BFM	ISS
Simulation Time [s]	**MP3**	0.09	0.12	0.13	0.61	48.8	3921
	JPEG	0.40	0.09	0.10	0.20	48.6	3209
	GSM	0.29	0.29	0.29	0.31	2.5	15881[3]
Speedup	**MP3**	43563	32672	30159	6427	80	1
	JPEG	8022	35653	32088	16044	66	1
	GSM	54762	54761.8	54762	51229	6456	1
Sim. Perf. $[\frac{MCycles}{sec}]$	**MP3**	2080	1560	1440	307	3.8	0.05
	JPEG	318	1415	1273	637	2.6	0.04
	GSM	411	411	411	384	48.4	0.01

model's performance is dominated by our processor model (including its bus interface and interrupt synchronization), and allows us to focus on it for the analysis.

Figure 15 shows the simulation time for each software-only subsystem, when individually simulated. Table II shows the numerical results and compares them in terms of speedup and simulation performance.

The simulation time for the SW-only subsystems remains in the same range up to the firmware model. Here, the model abstracts away most of the processor features and the simulation speed is limited by the algorithm computation. Starting with the *TLM*, processor features start dominating the simulation effort and the results become specific for the modeled processor. The JPEG shows exceptionally long simulation time for the application model. It's specification frequently uses pipelined constructs, which leads to many parallel simulation threads in the application model, slowing down the simulation. These pipelined constructs are statically scheduled in the task model, reducing the simulation time.

The *TLM* shows a sustained simulation performance of 307 MCycles/sec to 637 MCycles/sec depending on the application. In addition to what is shown in the table, we measured a peak performance of 2,800 MCycles/sec in code sections without external communication, which demonstrates the performance potential when excluding the impact of the processor bus Adding bit- and cycle-accurate communication in the *BFM* reduces the performance to about 3 MCycles/sec. The GSM application shows shallower drop in performance with adding details. The abstracter models are limited by the application complexity and the emulation of multiply accumulate operations. The *GSM TLM* and *BFM*, however, profit from the a simpler processor model (e.g. no external PIC), a simpler single master bus model, and from comparatively little communication.

4.2.2 *Performance of HW/SW Systems.* The HW/SW systems are the subsystems of the cellphone example. We use the MP3 with significant HW support (4 accelerator blocks and 4 busses in total) as shown in Figure 14. To examine multitasking between different applications, we also examine a combined MP3 and JPEG (at lower priority). The third example is the GSM with HW acceleration for the codebook search. Figure 16 shows the simulation time for each HW/SW subsystem and Table III gives the numerical results.

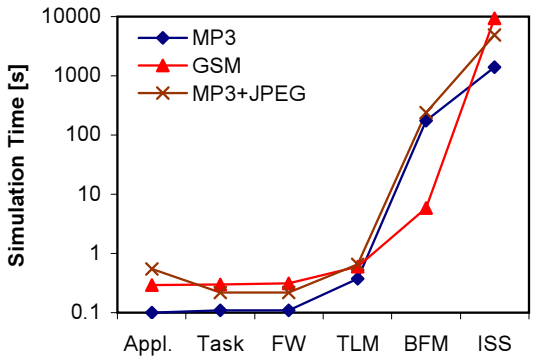


Fig. 16. Simulation time for HW/SW systems.

Table III. Simulation performance of HW/SW systems.

		Appl.	Task	FW	TLM	BFM	ISS
Simulation Time [s]	**MP3**	0.10	0.11	0.11	0.37	174.8	1383
	MP3+JPEG	0.55	0.22	0.22	0.66	239.4	4838
	GSM	0.29	0.30	0.31	0.60	5.8	9471[3]
Speedup	**MP3**	13835	12577	12577	3739	8	1
	MP3+JPEG	8797	21992	21992	7331	20	1
	GSM	32658	31570	30551	15785	1627	1
Sim. Perf. $[\frac{MCycles}{sec}]$	**MP3**	1440	1309	1309	389	0.8	0.10
	MP3+JPEG	367	916	916	305	0.8	0.04
	GSM	341	330	319	165	17.0	0.02

In general, the number of simulated cycles is lower with the additional hardware and communication, since we do not account computation on hardware components or bus communication toward the simulation cycles. With this definition, the TLM reaches half of the previous performance with a range from 165 MCycles/sec to 389 MCycles/sec. For the MP3, less than one third of the computation is performed on the processor (the SW-only frame delay of 32ms drops to 9ms with the HW support)[4] so that the *TLM* performance slightly increases. With the increased communication, the gap between *TLM* and *BFM* is larger. The *BFM* only reaches 0.8 MCycles/sec for the ARM based systems. The GSM with the simpler communication still reaches 17 MCycles/sec.

Summarizing the performance measurements, our processor abstraction provides very high simulation speeds. The achievable performance is higher with an increased software content (up to 637 MCycles/sec sustained, 2,800 MCycles/sec peak). Abstracting beyond the *FW* level is not efficient, since the performance does not significantly increase further.

4.3 Accuracy Analysis

After having asserted the benefits of abstract processor modeling in terms of simulation performance, we will now look at the achievable accuracy with abstract simulation. Again we separate our analysis into SW-only and HW/SW systems.

4.3.1 *Accuracy of SW-only Systems.* As described earlier, we express the error as the average timing error in the simulated frame delay. Table IV shows the results for the SW-only version of the subsystems.

Table IV. Simulation accuracy of SW-only systems.

	Appl.	Task	FW	TLM	BFM	ISS
MP3 (SW)	46.19%	1.51%	1.51%	1.63%	1.63%	0%
JPEG	58.13%	3.64%	3.61%	2.32%	2.32%	0%
GSM (SW)	6.78%	1.13%	1.10%	0.72%	0.72%	0%

[4]Note that we only count the processor cycles for computing the simulation performance. The cycles for computation in HW are ignored. However, CPU idle cycles are counted toward the performance.

For SW-only subsystems, only modeling of the *Task* level significantly reduces the error. Here, parallel executing tasks are scheduled on the processor. Each further simulation level, which models the external communication in more detail, does not significantly reduce the error. The SW-only systems insignificantly use external communication. The error remains constant at about a 2% level. The *JPEG* subsystem still profits from the *TLM* simulation since it uses the bus slightly more comparing to the other two examples.

4.3.2 *Accuracy of HW/SW Systems.* With the additional communication and synchronization present in the HW/SW systems, the abstract models exhibit a higher error amount (see Figure 17 and Table V). For the ARM-based systems, the error drops linearly from ≈50% error for the application down to few percent for the *TLM*. The *FW* level, that produced excellent results in the SW-only measurements, now exhibits more than 10% error. The *TLM* simulates with less than 2% error. The *JPEG+MP3* has a slightly higher error of 3%, due to the frequent preemption of the JPEG encoder by the MP3 decoder.

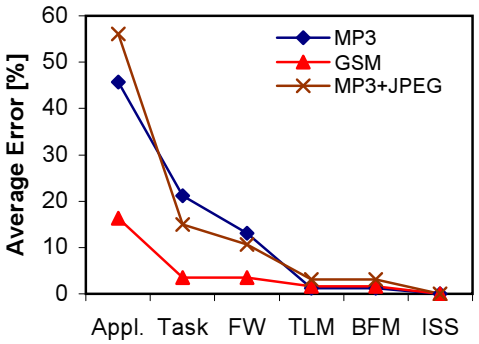


Fig. 17. Simulation accuracy of HW/SW systems.

Table V. Simulation accuracy of HW/SW systems.

	Appl.	**Task**	**FW**	**TLM**	**BFM**	**ISS**
MP3	45.72%	21.11%	13.02%	1.17%	1.17%	0%
MP3+JPEG	56.06%	14.90%	10.64%	3.09%	3.09%	0%
GSM	16.37%	3.56%	3.53%	1.68%	1.68%	0%

4.4 Performance and Accuracy in System Simulation

Combining the performance and the accuracy measurements yields the trade-off in system simulation. We show this trade-off for a simulation of all three subsystems of the cellphone, as shown in Figure 14.

Figure 18 shows the results of our performance and accuracy analysis for the cellphone system with all subsystems running concurrently. It shows the simulation

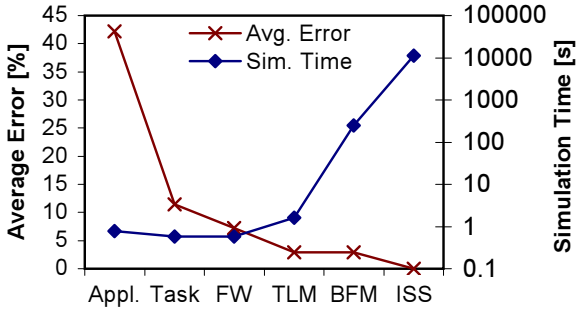


Fig. 18. Performance and accuracy for cellphone system simulation.

Table VI. Performance and accuracy for cellphone system simulation.

	Appl.	**Task**	**FW**	**TLM**	**BFM**	**ISS**
Avg. Error	42.2%	11.4%	7.25%	2.93%	2.93%	0%
Sim Time [s]	0.78	0.57	0.58	1.6	250	11363[3]
Sim. Perf. $[\frac{MCycles}{sec}]$	614	840	825	299	1.91	0.07
Speedup	14568	19935	19591	7102	45	1

time and the average system error, which we define as the mean of the subsystem errors. For reference, Table VI lists the detailed numerical results.

Our combined cellphone example simulates 3 seconds of real-time, with 180 million DSP cycles and 300 million ARM processor cycles. The simulation time increases over-exponentially with the added features. Our processor *TLM* executes 299MCycles/sec and is four orders of magnitude faster than the ISS-based co-simulation. With an increased abstraction, a higher error in simulated time has to be accepted. Our processor *TLM* exhibits a average error of only 2.9%. Representing fewer processor features further increases the simulation performance at the cost of a larger error. The *firmware* compilation step, for example, simulates 825 MCycles/sec, with a 7.25% error.

Our measurements clearly show the TLM trade-off between simulation performance and accuracy. Both the *TLM* and the *firmware* compilation step indicate efficient alternatives based on the preference for accuracy and speed, respectively. Adding further detail with the *BFM* does not yield a further advantage in accuracy. However, *BFM*s are suitable for a pin level integration of RTL IP components and provide a cycle detailed view of the bus traffic.

5. SUMMARY AND CONCLUSION

In this paper, we have presented our approach for system level software modeling using abstract processor models. We use a layered modeling approach that allows models at varying abstraction level to be generated automatically by our SCE framework. We have incrementally described our processor models with the essential features of task mapping, dynamic scheduling, interrupt handling, low level firmware, and hardware interrupt handling.

We have evaluated our approach using an industrial strength cellphone example with GSM 06.60 speech transcoding, MP3 decoding and JPEG encoding mapped to a heterogeneous MPSoC with custom hardware blocks. We have used our SCE framework to automatically generate the processor models and intermediate compilation steps to analyze five levels of abstraction. We analyzed each level in detail with respect to the gain in simulation performance and the loss in accuracy.

Our results show the tremendous benefits of our abstract processor modeling of fast and accurate simulation. For SW-centric examples, our proposed *TLM* executes up to 600 MCycles/sec sustained and up to 2,800 MCycles/sec peak, which is multiple orders of magnitude faster than a traditional cycle-accurate *ISS*-based simulation. For the complex cellphone example, our model reaches 300 MCycles/sec and exhibits an error of only 2.9%. An even higher speed of 825 MCycles/sec is achieved by the *firmware* model (with an error of 7.25%).

Our processor models enable rapid design space exploration at an early stage of the design process. Automatically generating the models improves productivity for software development. It exposes performance implications of the design choices already early in the process. Furthermore, our processor TLM, with its rich set of details, serves as a platform for a fast functional validation with a high confidence level.

Acknowledgments

The authors would like to acknowledge the contribution of Prof. Daniel D. Gajski's Embedded Systems Methodology Group in the Center for Embedded Computer Systems at UC Irvine. Especially, the authors thank Junyu Peng and Dongwan Shin for their support of the architecture and communication refinement tools. Furthermore, the authors thank Pramod Chandraiah for providing the initial specification model of the MP3 decoder.

REFERENCES

ARC. ARC xISS Simulators. http://www.arc.com/software/simulation/xiss.html.

ARM. SoC Developer with MaxSim Technology. http://www.arm.com/products/DevTools/MaxSim.html.

ARM. 2001. ARM7TDMI (Rev 3) Product Overview. http://www.arm.com/pdfs/DVI0027B_7_R3.pdf.

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

Bhaskaran, V. and Konstantinides, K. 1997. *Image and Video Compression Standards: Algorithms and Architectures, 2nd edition*. Kluwer Academic Publishers.

Bouchhima, A., Bacivarov, I., Yousseff, W., Bonaciu, M., and Jerraya, A. 2005. 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.

Buck, J., Ha, S., Lee, E. A., and Messerschmitt, D. G. 1994. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation 4,* 2 (April), 155–182.

Cai, L., Gerstlauer, A., and Gajski, D. D. 2005. Multi-Metric and Multi-Entity Characterization of Applications for Early System Design Exploration. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*. Shanghai, China.

CoWare. Virtual Platform Designer. http://www.coware.com.

Dales, M. 2000. *SWARM 0.44 Documentation*. Department of Computer Science, University of Glasgow.

Dömer, R., Gerstlauer, A., Peng, J., Shin, D., Cai, L., Yu, H., Abdi, S., and Gajski, D. D. 2008. System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *2008,* 647953, 13.

ETSI. 1996. *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding*, GSM 06.60 ed. European Telecommunication Standards Institute (ETSI).

Furukawa, T., Honda, S., Tomiyama, H., and Takada, H. 2007. A hardware/software cosimulator with RTOS supports for multiprocessor embedded systems. In *Proceedings of International Conference on Embedded Software and Systems*. Daegu, Korea.

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

Gao, L., Karuri, K., Kraemer, S., Leupers, R., Ascheid, G., and Meyr, H. 2008. Multiprocessor performance estimation using hybrid simulation. In *Proceedings of the Design Automation Conference (DAC)*. Anaheim, CA.

Gerstlauer, A., Schirner, G., Shin, D., Peng, J., Dömer, R., and Gajski, D. D. 2006. System-On-Chip Component Models. Tech. Rep. CECS-TR-06-10, Center for Embedded Computer Systems, University of California, Irvine. May.

Gerstlauer, A., Shin, D., Peng, J., Dömer, R., and Gajski, D. D. 2007. Automatic, Layer-based Generation of System-On-Chip Bus Communication Models. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD) 26,* 9 (Sept.).

Gerstlauer, A., Yu, H., and Gajski, D. D. 2003. RTOS Modeling for System Level Design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*. Munich, Germany.

Ghenassia, F. 2005. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer.

Giridhar, C. 2005. Trendy phones incorporate sophisticated engineering, EDN Asia. http://www.edn.com/article/CA6290467.html.

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

Kempf, T., Dörper, M., Leupers, R., Ascheid, G., Meyr, H., Kogel, T., and Vanthournout, B. 2005. 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.

Labrosse, J. J. 2002. *MicroC/OS-II: The Real-Time Kernel*. CMP Books.

Mong, W. S. and Zhu, J. 2004. DynamoSim: a trace-based dynamically compiled instruction set simulator. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. San Jose, CA.

Montoreano, M. 2007. *Transaction Level Modeling using OSCI TLM 2.0*. Open SystemC Initiative (OSCI).

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

Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., and Hoffmann, A. 2002. A universal technique for fast and flexible instructionset architecture simulation. In *Proceedings of the Design Automation Conference (DAC)*. New Orleans, LA.

Reshadi, M., Mishra, P., and Dutt, N. 2009. Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. *ACM Transactions on Embedded Computer Systems*.

Schirner, G. and Dömer, R. 2008. Abstract communication modeling and analysis. *ACM Transactions on Embedded Computer Systems 8,* 1 (Aug.), 4:1–4:29.

Schirner, G., Dömer, R., and Gerstlauer, A. 2009. High-level development, modeling and automatic generation of hardware-dependent software. In *Hardware Dependent Software: Principles and Practice*, W. Ecker, W. Müller, and R. Dömer, Eds. Springer.

Schirner, G., Gerstlauer, A., and Doemer, R. 2007. Abstract, Multifaceted Modeling of Embedded Processors for System Level Design. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*. Yokohama, Japan.

VaST. VaST Tools and Models for Embedded System Design. http://www.vastsystems.com.

Virtutech. Simics - embedded systems simulation platform. http://www.virtutech.com.

Yi, Y., Kim, D., and Ha, S. 2007. Fast and accurate cosimulation of mpsoc using trace-driven virtual synchronization. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD) 26,* 12, 2186–2200.

Zabel, H., Müller, W., and Gerstlauer, A. 2009. Accurate RTOS modeling and analysis with SystemC. In *Hardware Dependent Software: Principles and Practice*, W. Ecker, W. Müller, and R. Dömer, Eds. Springer.

Received: May 2009