The Dissertation Committee for Parisa Razaghi
certifies that this is the approved version of the following dissertation:

Dynamic Time Management for Improved Accuracy and Speed in Host-Compiled Multi-Core Platform Models

Committee:

Andreas Gerstlauer, Supervisor

Aloysius K. Mok

Jonathan W. Valvano

Derek Chiou

Vijay Janapa Reddi

Dynamic Time Management for Improved Accuracy and Speed in Host-Compiled Multi-Core Platform Models

by

Parisa Razaghi, B.E.; M.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Dedicated to my dad.

Acknowledgments

I would like to express my sincere appreciation to my advisor, Prof. Andreas Gerstlauer, for his kind support. Without his insights and guidance, this research would not have been possible. I would also like to thank my committee members, Prof. Al Mok, Prof. Jonathan Valvano, Prof. Derek Chiou and Prof. Vijay Reddi, for their invaluable comments and suggestions. I am particularly indebted to Vijay for his encouragement and persuasion. In addition, I wish to thank my friends and colleagues who helped me along the way to completing this dissertation. I would especially thank Dr. Andrew Lenharth and Xinnian Zheng, for their invaluable knowledge and assistance.

Foremost, I would like to express the deepest appreciation to my father, my best teacher. Without his unwavering support and sacrifices, I would not have been able to pursuit my dreams. I would like to express my love to my mother, for her affection and encouraging me with her best wishes. I would also like to thank my brothers and sister for supporting and believing in me. Special thanks goes to my brother, Masoud, for his infinite kindness and first introducing me to this field when I was in Middle School. Finally, I am deeply thankful to my husband, Mehdi, for his love and helping me to follow my dreams.

Dynamic Time Management for Improved Accuracy and Speed in Host-Compiled Multi-Core Platform Models

Publication No. _____

Parisa Razaghi, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Andreas Gerstlauer

With increasing complexity and software content, modern embedded platforms employ a heterogeneous mix of multi-core processors along with hardware accelerators in order to provide high performance in limited power budgets. Due to complex interactions and highly dynamic behavior, static analysis of real-time performance and other constraints is challenging. As an alternative, full-system simulations have been widely accepted by designers. With traditional approaches being either slow or inaccurate, so-called host-compiled simulators have recently emerged as a solution for rapid evaluation of complete systems at early design stages. In such approaches, a faster simulation is achieved by natively executing application code at the source level, abstracting execution behavior of target platforms, and thus increasing simulation granularity. However, most existing host-compiled simulators often focus on application behavior only while neglecting effects of hardware/software interactions and associated speed and accuracy tradeoffs in platform modeling.

In this dissertation, we focus on host-compiled operating system (OS) and processor modeling techniques, and we introduce novel dynamic timing model management approaches that efficiently improve both accuracy and speed of such models via automatically calibrating the simulation granularity.

The contributions of this dissertation are twofold: We first establish an infrastructure for efficient host-compiled multi-core platform simulation by developing (a) abstract models of both real-time OSs and processors that replicate timing-accurate hardware/software interactions and enable full-system co-simulation, and (b) quantitative and analytical studies of host-compiled simulation principles to analyze error bounds and investigate possible improvements. Building on this infrastructure, we further propose specific techniques for improving accuracy and speed tradeoffs in host-compiled simulation by developing (c) an automatic timing granularity adjustment technique based on dynamically observing system state to control the simulation, (d) an out-of-order cache hierarchy modeling approach to efficiently reorder memory access behavior in the presence of temporal decoupling, and (e) a synchronized timing model to align platform threads to run efficiently in parallel simulation.

Results as applied to industrial-strength platforms confirm that by providing careful abstractions and dynamic timing management, our models can achieve full-system simulations at equivalent speeds of more than a thousand MIPS with less than 3% timing error. Coupled with the capability to easily adjust simulation parameters and configurations, this demonstrates the benefits of our platform models for early application development and exploration.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In todays embedded systems, software content is growing continuously to deal with increased complexities and tight development cycles. During early design space exploration, system developers are interested in evaluating an application behavior on a particular architecture. However, system-wide interactions and dynamic behavior in complex parallel systems make static analysis challenging. Efficient full-system simulations therefore play an important role in the design process.

Multi-core processors have become popular both in general-purpose as well as in embedded computing in order to achieve continued high performance while managing power budgets [8]. In practice, such multi-core processors are integrated into a multi-processor platform in order to provide a heterogeneous multi-processor and multi-core systems-on-chip (MPCSoCs) that meets all real-time and design constrains. The complexities of the MPCSoC design space have made traditional cycle- or instruction-accurate simulators inefficient. Cycle-based simulators are highly accurate, but very slow, especially in a multi-core or multi-processor context. Instead, high-level simulators can establish a fast simulation with an acceptable level of degraded accuracy.

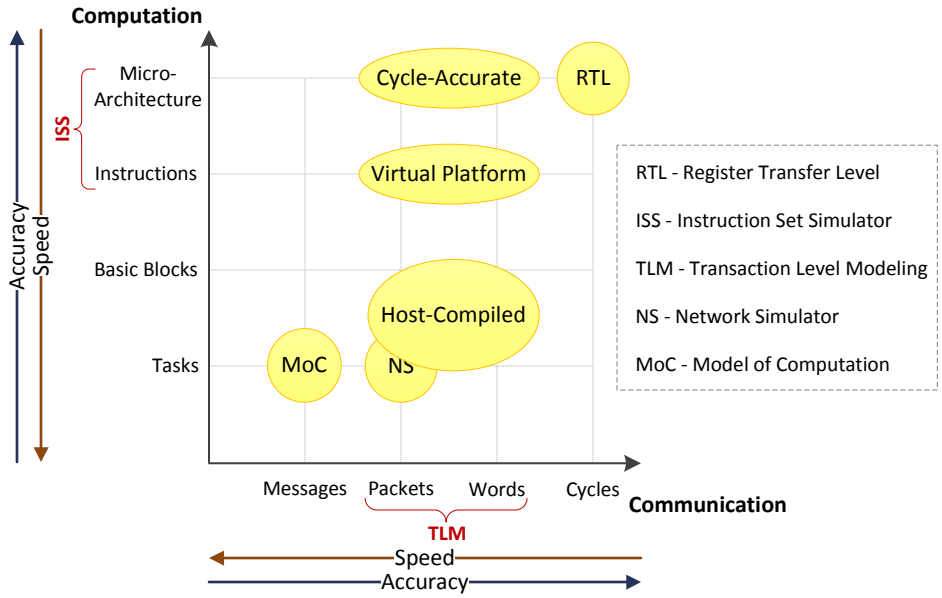


Figure 1.1: Modeling space [28].

1.1 Modeling Space

Figure 1.1 classifies existing simulation approaches based on the modeling abstraction level [28]. In this figure, system models are organized along communication and computation parts executing at different levels of granularity.

Models of computation (MoCs) [44, 48] provide the fastest possible simulation platform by functional execution of tasks, which exchange messages over high-level communication channels. At the other end of the spectrum are models described at the *register transfer level* (RTL), which contain micro-architectural details to provide cycle-accurate simulation results. Due to a high degree of detail and fine granularity required for modeling interactions

among system components, however, their simulation performance is very low.

In order to provide fast simulation while maintaining an acceptable level of accuracy, a range of intermediate simulation approaches have been developed. From a communication perspective, *transaction-level modeling* (TLM) [11] has been introduced to abstract on-chip communication from pins and cycles to the level of word or packet transactions in order to increase the simulation speed while still providing accurate timing estimates. Traditional micro-architectural *instruction set simulators* (ISS) [3, 6, 76, 54] execute the binary code of applications on a cycle accurate model of a target processor. Such approaches coupled with TLM can provide accurate full system simulation but still tend to be slow, especially in a multi-core or multi-processor context.

By contrast, *virtual platform* (VP) [82, 4] prototypes that employ binary translation coupled with abstract modeling of system peripherals can establish fast functional simulation, but provide little to no timing information. Similarly, *network simulators* (NS) [37, 93] provide fast simulation of the entire network by abstracting both communication and computation units.

More recently, *source-level* and *host-compiled* (HC) [27] simulators have emerged as an alternative that aims to address the need for fast and accurate simulation. In pure source-level approaches, application code is natively compiled and executed on a host machine to achieve the fastest possible functional simulation. For accuracy, the source code is further back-annotated with target-specific timing information obtained through estimation or measure-

ment. To achieve full host-compiled simulation, back-annotated source code is then wrapped into abstract models of operating systems and processors, which integrate into existing TLM backplanes on top of standard system-level design languages (SLDLs), such as SpecC [25] or SystemC [30].

Host-compiled simulators are generally used for system-level design space exploration of MPCSoC platforms. In a typical system-level design flow, system-level synthesis tools explore mapping of MoCs onto a target platform using host-compiled platform models for overall performance evaluation of the mapped systems at early stages of design process. As such, host-compiled models are meant to complement existing low-level ISS/RTL models, which are still needed for detailed micro-architectural exploration or final sign-off.

1.2 Host-Compiled Simulation

Figure 1.2 shows the structure of a typical host-compiled simulator with a layered organization as introduced for single-core models in [81], which we have expanded to a multi-core version as the basis of our research.

The application running on each processor is given in a multi-threaded C code form. Typically, mounting the application on the simulator should not be more complicated than porting it to a different OS. In doing so, the source code needs to be converted into an application model by translating tasks/threads into SLDL processes and OS and inter-processor communication (IPC) primitives into equivalent canonical, high-level application programming interfaces (APIs) provided by the simulator. This can be done

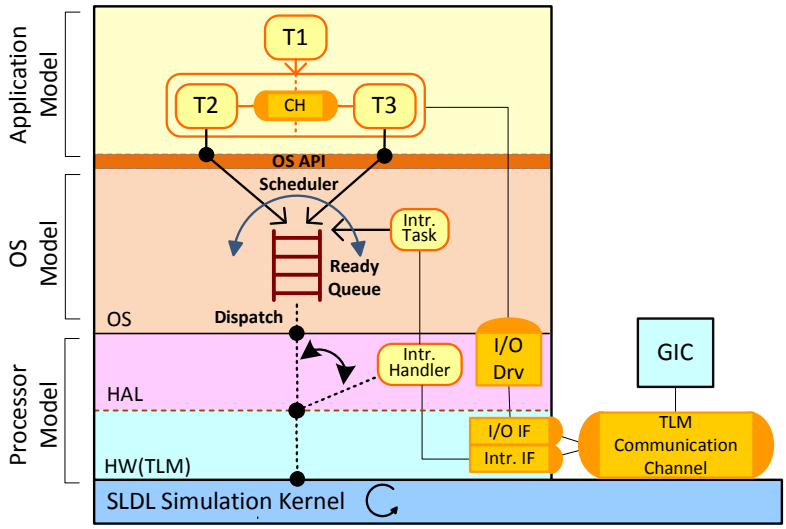


Figure 1.2: High-level, host-compiled simulation platform.

automatically [22] or by providing a thin wrapper for standard OS APIs, such as POSIX. In addition, for timing accuracy, the application code is further back-annotated with execution delays estimated or measured based on a selected target platform, which can also be performed automatically [15]. In the end, mounting an application on a host-compiled simulator is fully transparent to the user. The same application code running on top of the simulator will later be synthesized and compiled to run on a real target platform by automatically removing back-annotated delays and transforming high-level OS calls into an execution on top of a real OS [80].

At the core of the simulation engine, an OS model implements the simulator's high-level canonical OS interface for multi-tasking and IPC. Internally, the OS model replicates an abstract OS kernel, which emulates the execution of application tasks on top of underlying SLDL kernel. The OS model

thereby schedules, queues, dispatches and executes the application tasks according to a chosen scheduling policy, which is configurable to emulate different scheduling strategies. Underneath, a hardware abstraction layer (HAL) in conjunction with a hardware layer constitute the high-level processor model. The HAL integrates the software into the processor hardware models and includes necessary description of I/O drivers and interrupt handlers. Combined with interrupt processing in the OS layer, this replicates and emulates an accurate interrupt handling mechanism. The hardware layer also provides interrupt and bus interfaces to the external communication infrastructure. Generally, the bus interface can be developed at an arbitrary level of abstraction, depending on a desired level of accuracy; here, a transaction level model (TLM) of communication is used to establish a fast simulation environment. A high-level, generic interrupt controller (GIC) model collects interrupts from the hardware side and manages their distribution to the processor model.

Finally, at the base, the complete simulator is implemented over a standard system level design language (SLDL) that provides the required concurrency, timing and event handling infrastructure on a host machine.

1.3 Thesis Scope

Most of the existing approaches in the host-compiled domain have focused on source-level simulation of the application level only, including accurate back-annotation processes. However, such source-level models only replicate execution of sequential codes without considering any interference from

other tasks or HW/SW interactions. Yet, management of intra- and inter-processor interactions in the OS kernels, device drivers and interrupt handling chains of complex multi-processor and multi-core platforms can carry a large overhead. As such, OS- and system-level interactions can contribute significantly to overall system performance. At the same time, execution of associated detailed code in traditional instruction set simulations can lead to a large simulation overhead. In reality, designers may only care about the effect on application performance, and they are not concerned with the OS internals, for example. This provides an opportunity and need to abstract such details and develop fast and accurate host-compiled OS and processor models that faithfully replicate such effects without including any of the associated simulation overhead. To achieve this goal, the main focus of this dissertation is on multi-core platform modeling with a specific emphasis on host-compiled OS and processor models.

In this dissertation, we target embedded and mobile SoC platforms. We assume that back-annotated source-level application models are given. Instead, we specifically focus on modeling the effects of interactions among multiple tasks on top of shared resources, which includes models for the OS scheduler, inter-process communication, communication with the external world through high-level bus transactions, interrupt handling chains, and multi-core cache hierarchies. Modeling of other OS effects, such as virtual memory systems that are not typically found in embedded and mobile platforms are subject to future work.

1.4 Thesis Statement

Thus far, existing host-compiled OS and processor modeling approaches have only focused on single-core platforms and suffer from inherent speed and accuracy tradeoffs. In this dissertation, we demonstrate that both accuracy and speed of host-compiled platform (OS and processor) models can be significantly improved by dynamically managing simulation granularities in the platform model itself. This is achieved by (i) an automatic timing granularity adjustment, which monitors the system state and automatically controls the timing model of the simulation platform to provide both fast and accurate results, and (ii) a multi-core, out-of-order cache hierarchy modeling approach that incorporates a delayed reordering of aggregated requests to provide accurate memory behavior in coarse-grained platform simulations. In conjunction with the above approaches, (iii) a synchronized timing model fully exploits potential of modern multi-core host workstations for parallel host-compiled simulations.

1.5 Contributions

In the following, we briefly summarize the contributions presented in the subsequent chapters.

1.5.1 Host-Compiled Multi-Core Platform Simulation

Host-compiled simulation approaches are based on a modeling abstraction concept. Hence, careful abstraction of low-level details is crucial for fast

and accurate results. In this dissertation, we first develop *multi-core OS and processor models* inspired from well-established single-core host-compiled models in order to provide a comprehensive and flexible multi-core platform infrastructure. We further examine conventional host-compiled platform models to assess the accuracy of abstract models compared to detailed and fully accurate simulators. We finally present an *analytical model of error bounds*, which shows that, generally, errors are a direct function of simulation granularities. However, contrary to common intuition, it is further shown that under certain circumstances, errors in discrete preemption models can potentially exceed the bounds set by the timing granularity by a large amount. Related contributions are as follows:

- A host-compiled multi-core platform simulator, which incorporates a parametrizable SMP OS model supporting a wide range of scheduling schemes and policies, and a high-level multi-core processor model with integrated models of configurable host-compiled interrupt handling chains.
- A quantitative study of accuracy and speed tradeoffs in host-compiled simulations, including an investigation of error sources in host-compiled platform models.
- An analytical model of error bounds in preemptive discrete-event simulations, demonstrating that timing errors can potentially exceed the bounds set by the predetermined simulation granularity.

1.5.2 Automatic Timing Granularity Adjustment

In host-compiled simulation, higher speed is achieved by coarse-grained simulation of discrete events, which degrades timing accuracy of preemptive simulations. The central contribution of this dissertation is an approach for largely eliminating the accuracy and speed tradeoff of preemptive discrete-event simulation of host-compiled platforms. We proposes an *automatic timing granularity adjustment* (ATGA) approach in which the platform models internally monitor the state of the system and automatically control the timing granularity of the simulation. When applied to an OS model, simulation speed and accuracy is independent of the granularity of back-annotated delays, which frees designers from having to settle on a particular, difficult to evaluate and predict tradeoff. Instead, the OS kernel itself accumulates or breaks delays into a number of smaller steps as needed, automatically providing the best timing granularity for accurate and fast results. Specific contributions are:

- A predictive timing model management algorithm, which introduces a method for adjusting the granularity of simulation to eliminate preemption errors in fully predictable applications, such as periodic task sets.

- A comprehensive timing model management method for automatic timing granularity adjustment in sporadic applications with hardware interactions, which is achieved by integration of a fallback algorithm into our basic predictive timing model approach.

- An enhanced algorithm that applies finer control over when to switch to a fine-grain fallback mode in order to deal with unpredictable external events.

- A multi-core ATGA method, which further considers inter-core interactions between regular application or special interrupt tasks running on different cores.

1.5.3 Multi-Core, Out-of-Order Cache Simulation

In the ATGA approach, the OS model internally accumulates task delays and only synchronizes global time when a task switch is required. In a multi-core configuration, this temporally decouples cores and allows them to go ahead of the simulation time. As such, accurate multi-core cache hierarchy modeling becomes challenging, since memory references from different cores can be globally committed out-of-order. To combat this problem, we propose a *multi-core, out-of-order cache* (MOOC) modeling approach, which incorporates a delayed reordering of aggregated requests to provide an accurate cache hierarchy simulation in the presence of temporal decoupling. The proposed reordering approach can be integrated into any general, high-level multi-core cache model, which enables accurate yet fast cache simulation of multi-core processors. The main contributions are as follows:

- A multi-core cache management approach, which, when integrated into a temporally decoupled multi-core simulation, incorporates a reordering

technique for fast yet accurate cache commitment of globally out-of-order memory accesses.

- An extended reordering approach, which, when coupled with the OS model, further considers task preemption effects for accurate reordering of memory requests.

1.5.4 Synchronized Timing Model for Parallel Simulation

With the advent of multi-core processors in common workstations and PCs, parallel discrete-event simulators have received increased attention as a solution for providing faster simulation of virtual platforms. However, the simulation performance is still limited by the amount of parallelism available in the platform model itself. In this dissertation, we propose a novel modeling technique that manages and synchronizes the execution of discrete concurrency in multi-core models to boost parallel simulation performance. We introduce a *synchronized timing model* which is coupled with our host-compiled models in order to efficiently control the available parallelism between the platform models and the underlying parallel simulator. In doing so, we make the following contributions:

- A synchronized timing model, which uses static alignment of temporal quanta to issue platform threads to the underlying simulator kernel in the same simulation cycle for fully utilized parallel simulation.

- A dynamic quantum selection method, which, when integrated into the

ATGA approach, efficiently aligns platform threads to achieve the fastest possible parallel simulation.

1.6 Methodology

As mentioned before, for the purposes of OS and processor modeling, we assume that previously back-annotated source-level code is given. In this dissertation, we manually back-annotate and mount application models onto our simulator. In order to evaluate the accuracy of our host-compiled OS and processor models, we thereby assume ideal back-annotation, where we measure and back-annotate average execution times at the function level from a complete run of the application on a reference platform model or a real board. This allows us to isolate errors in our platform models from other error sources, such as back-annotation errors, which would otherwise further decrease accuracy of the overall host-compiled simulation.

In order to evaluate the accuracy of our host-compiled OS and processor models, we use a reference ISS that simulates one instruction per cycle. Note that accuracy numbers reported in this dissertation are largely independent from the reference execution used for back-annotation. The reference provides a baseline for fully accurate simulation and back-annotation of sequential pieces of code. By contrast, our focus is on errors in the models of multi-tasking and HW/SW interactions and interferences, which are to large extend orthogonal to the timing model assumed for sequential code. This is further evidenced by the fact that consistent accuracy results are observed

when we use a real board as a reference for back-annotation and comparison.

Timing errors are measured as response times (denoted as $R_{i,j}$) of single iterations (j) of individual tasks (i) in an application as per the following equation:

$$error_{i,j}\ [\%] = \frac{|R_{i,j}(HC\ Simulator) - R_{i.j}(ISS_{ref})|}{R_{i,j}(ISS_{ref})} \times 100 \qquad (1.1)$$

The average error is then calculated over all iterations of all tasks running on the system:

$$avg_{error}[\%] = \frac{\sum_{i=1}^{I} \sum_{j=1}^{J_i} error_{i,j}}{\sum_{i=1}^{I} J_i} \qquad (1.2)$$

To evaluate the simulation speed of our models, we measure the number of simulated instructions on the reference ISS. Based on the total simulation time of our complete host-compiled models, we present speed numbers as simulation throughput measured in millions of simulated instructions per real second (MIPS) using the following equation:

$$Speed\ [MIPS] = \frac{Simulated\ Instructions\ (ISS_{ref})/10^{+6}}{Simulation\ Time} \qquad (1.3)$$

1.7 Thesis Outline

The remainder of this dissertation is organized as follows. Chapter 2 reviews relevant prior work on a few distinct areas. We mainly focus on existing system-level simulators and prior research on improving associated accuracy and speed tradeoffs. We then focus on the related work on high-level cache simulations. Finally, we discuss the prior work on parallel simulation of multi-core virtual platforms.

Chapter 3 presents an analysis of error bounds in task preemption models of a conventional host-compiled simulation. Traditionally, the assumption is that timing errors are bounded by annotated discrete timing granularities. However, the presented analysis shows that errors can exceed such limits significantly depending on the system utilization. Such a large error bound can consequently limit the usefulness of host-compiled simulations for evaluating real-time performance.

Chapter 4 presents the design and implementation of our abstract SMP OS model. This chapter further proposes a novel automatic timing granularity adjustment (ATGA) approach, which avoids the task preemption errors presented in Chapter 3. In such an approach, an extended OS model is capable of continuously monitoring system state to automatically and dynamically adjust simulated timing granularity.

Chapter 5 presents our work on a multi-core processor model, which incorporates a high-level interrupt handling method and a multi-core cache hierarchy channel in order to accurately replicate external interactions and evaluate overall real-time performance, respectively. We introduce a multi-core out-of-order cache modeling approach, which incorporates a delayed reordering of aggregated requests to provide an accurate cache hierarchy simulation along with our ATGA approach.

Chapter 6 extends the infrastructure from Chapter 4 and utilizes a novel synchronized timing model for efficient parallel simulation of multi-core virtual platforms. In the proposed approach, a supplementary timing model

is coupled with conventional virtual platform models in order to efficiently execute such models on multi-core hosts.

Chapter 7 concludes this dissertation, raises open questions, and proposes directions of future research.

Finally, Appendix 1 elaborates on the design and integration of more complex multi-core scheduling algorithms such as Pfair and G-EDF into our host-compiled OS model.

Chapter 2

Related Work

In this chapter, we briefly review existing host-compiled simulators and prior approaches for improving simulation accuracy and speed, high-level cache modeling approaches, and recent research on parallel virtual platform simulation.

2.1 Host-Compiled Simulation

Conventional ISS-based software simulators using micro-architectural or interpreted simulation [3, 6, 76, 54] can reach cycle accuracy at a speed of several kHz. At the other end of the spectrum, virtual platform simulators using dynamic binary translation can provide significant speedups (reaching simulation throughput of several MIPS), but only focus on functional simulation with no or very limited timing accuracy [82, 4, 7, 64, 31, 85]. Instead, hybrid approaches [18, 86, 49] employ functional model/timing model decoupling techniques to parallelize the simulation of various aspects for fast functional simulation while still providing cycle-accurate results. Although the aforementioned types of simulators offer multi-core support, including CPU modeling at different levels of abstraction ranging from instruction-accurate

models to fully cycle-accurate and micro-architectural ones, the need to simulate cross-compiled applications running on top of the complete binary code of an operating system kernel makes these simulator inefficient for fast and early integration and evaluation of complete systems.

Instead, source-level simulators aim to provide a fast yet accurate simulation platform by integrating instrumented source code of applications with a coarse-grain timing model that is obtained from the target architecture [56, 83, 14, 51]. For accurate performance evaluation, several approaches back-annotate the code with timing estimates that are obtained by compiling to an intermediate representation [39, 10, 91, 15].

Source-level approaches can provide accurate simulation of single-task application behavior, but lack support for modeling of parallel applications and architectures. Host-compiled simulators further extend source-level simulation to include abstract models of the software execution environment [27]. Originally, host-compiled simulators only focused on modeling of OS effects [29, 69, 36, 57, 47]. Later, those approaches were expanded into complete processor models that include timing accurate interrupt chains and TLM-based bus interface [26, 5, 9, 81]. Such host-compiled simulators have been shown to run at speeds beyond 500 MIPS with more than 95% timing accuracy. However, host-compiled simulation of complete multi-core platforms is still missing.

Improving the accuracy of high-level simulation while maintaining high performance has been the focus of many researchers. Krause *et al.* [45] present a hybrid ISS and RTOS modeling approach to combine cycle-accurate appli-

cation simulation with fast OS scheduling and context switching. Khaligh *et al.* [77] present an adaptive TLM simulation kernel, which changes the level of accuracy during simulation to the level expected by designers. Stattlemann *et al.* [88] propose an approach that precisely models the execution time of access conflicts in shared resources by using a proactive quantum allocator in a temporally decoupled simulation. Schirner *et al.* [78] introduce a result-oriented method for accurate simulation of interrupts on host-compiled processor models by applying optimistic prediction and correction. In all cases, however, fundamental, statically determined speed and accuracy tradeoffs remain.

2.2 Cache Simulation

Existing cache simulation techniques can be categorized into two common approaches: trace-driven or execution-driven simulations. *Trace-driven* cache simulators use a collected stream of memory accesses of applications to replicate the cache behavior [34]. This approach can be fast, but the simulator needs to deal with large trace files and data. In contrast, *execution-driven* simulators combine an executable cache model and simulated application instructions to capture memory accesses on-the-fly [17]. With the advent of MPCSoCs, cache simulators have also focused on simulating cache coherency and cache hierarchies [43, 33].

There have been several attempts for cache modeling in host-compiled or source-level simulation. Pedram *et al.* [66] present a high-level, search-based cache model integrated into a TLM processor simulation platform. Posadas

et al. [70] propose a faster approach by introducing a lookup table-based data cache model. Both include approaches for instrumenting application source code to update the cache state and adjust the back-annotated delays. Stattelmann *et al.* [89] introduce a hybrid source-level cache simulator, which uses application binary codes to annotate memory accesses. However, all these approaches suffer from a lack of multi-core cache hierarchy modeling with associated speed and accuracy challenges.

2.3 Parallel Virtual Platform Simulation

With the advent of multi-core PCs, parallel discrete-event simulation (PDES) has received widespread attention [24, 62]. Existing PDES approaches can be categorized by their inter-thread synchronization mechanisms and are divided into two common approaches: conservative or optimistic parallel simulations. A *conservative* PDES maintains sequential consistency by only issuing threads that are ready in the same simulation cycle, i.e. the same simulation time and delta cycle. Conservative PDES strictly follows causality constraints, and all threads are synchronized on simulation cycle advances. By contrast, in an *optimistic* simulation, no specific synchronization is required between threads, and threads are allowed to run ahead of other threads in simulated time. Instead, a *roll-back* mechanism is required to restore accurate simulation of any messages violating causality at synchronization points.

Accordingly, parallel discrete-event simulation of standard system-level design languages (SLDLs) has been proposed as a solution for improving full-

system simulation performance [21]. Chopard *et al* [19] propose a conservative SystemC simulator which partitions SystemC threads into clusters simulated by instances of SystemC schedulers running on distributed processing nodes. A master node then synchronizes SystemC threads at each update phase. Ezudheen *et al.* [23] instead present a single parallel SystemC scheduler that executes all SystemC threads in parallel on an SMP host. In this approach, SystemC processes are partitioned and mapped to a single host thread, and threads are issued if the runnable processes are in the same simulation cycle. In both approaches, however, designers need to manually partition SystemC modules. In similar approaches [84, 20], the evaluation phase of the SystemC or SpecC scheduler is parallelized, which synchronizes threads using a barrier at the end of evaluation phase. In these approaches, instead of a fixed mapping, the simulator or underlying OS will profile the execution of threads and balance work queues. All the approaches replicate a conservative parallel simulation. By contrast, Chen *et al.* [16] present a mixed conservative and out-of-order parallel simulation, in which inter-thread dependencies are statically determined and threads can be issued and move to different simulation cycles as long as such a dependency does not exist. In all such approaches, however, the speedup is limited by the amount of parallelism exposed by platform threads itself.

In addition to parallelized SLDL simulation kernels, several approaches utilize general features of virtual platforms to develop a specialized parallel simulator. Mello *et al.* [55] introduce a new programming style to utilize

temporally decoupled simulation of platform threads in a parallel simulator. The platform threads are synchronized when they receive special *sync* messages at simulation quantum boundaries. Pessoa *et al.* [67] then evaluate effects of communication locality on overall performance of such a parallel simulation approach. Similarly, Khaligh *et al.* [77] propose an approach in which each platform thread is mapped to a dedicated simulator. Platform threads are then temporally decoupled to provide fast simulation by locally incrementing the simulation time without actually synchronizing the global time. Again, a shared barrier is used to synchronize all simulators at simulation quantum boundaries. However, such methods only provide a basic infrastructure where choosing the right synchronization points and quantum boundaries is left to the user, which requires designers to be aware of the structure of the underlying simulation kernel.

Chapter 3

Tradeoff Analysis in Host-Compiled Simulations

In host-compiled simulation approaches, a faster simulation is achieved through source-level execution of application models. For performance evaluation, the source-code is then instrumented with target-specific execution delays and controlled by an abstract model of target platform. Hence, precise delay annotation and careful abstraction of platform models play an important role in an accurate and fast host-compiled simulation.

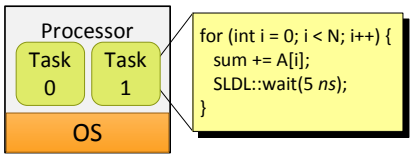
In this chapter*, we present a quantitative study and an analysis of accuracy and speed in convention host-compiled simulations. After presenting an example of the execution sequence for a host-compiled simulation in Section 3.1, we first examine the behavior of conventional host-compiled simulators under different timing granularities to study the challenges involved in fast and accurate platform modeling (Section 3.2). We further classify different sources of errors in such approaches. In Section 3.3, we present an analysis

*Contents of this chapter previously appeared in the following papers:
P. Razaghi and A. Gerstlauer. Host-compiled multicore RTOS simulator for embedded real-time software development. *DATE*, March 2011.
P. Razaghi and A. Gerstlauer. Predictive OS modeling for host-compiled simulation of periodic real-time task sets. *ESL*, March 2012.

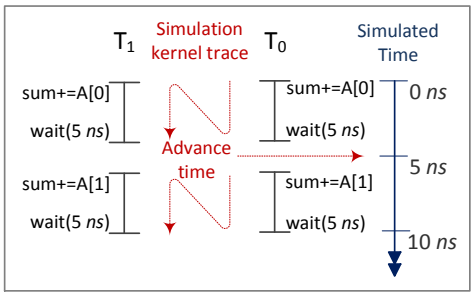
of error bounds in preemptive models that shows that errors can exceed limits set by timing granularities. At the end, we propose an approach to avoid such errors and provide an accurate simulation.

3.1 Host-Compiled Platform Simulation Example

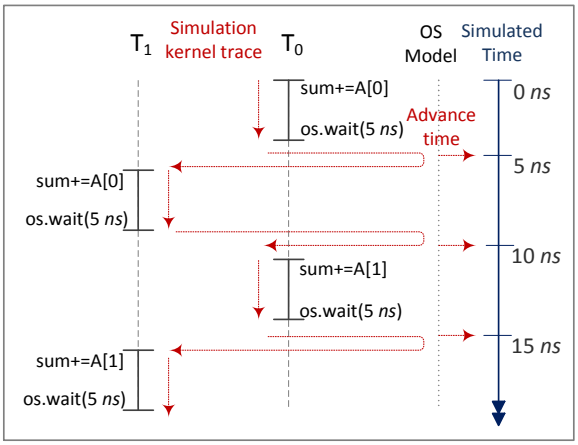
In this section, we demonstrate how a platform model manages the execution of a source-level application. Figure 3.1 (a) shows a simple application running two concurrent tasks, in which tasks' execution delays are back-annotated into the source code at basic-block granularities. Execution delays are modeled as *"wait for time"* calls to the underlying SLDL kernel. As such,



(a) Source-level application model.



(b) Source-level simulation trace.



(c) Host-compiled simulation trace.

Figure 3.1: Example of host-compiled simulation.

the overall timing behavior is emulated by the SLDL simulator. Figure 3.1 (b) shows the simulation trace of the application. At the beginning of the simulation, the SLDL kernel selects to execute task T_0. At the point where *wait (5 ns)* is called, the simulator switches to task T_1 until *wait(5 ns)* is reached there again. At this point, the simulator internally advances the simulated time to the next point (5 ns) and resumes the execution of T_0. As such, the SLDL simulator emulates the parallel execution of the application tasks.

So far, a target platform architecture is not considered during the simulation. To replicate sequential task execution on a single-core target processor, an abstract OS model is used to directly manage the timing model. Figure 3.1 (c) shows the simulation trace of the same application paired with an OS model. At the start of the simulation, the OS model suspends task T_1 and lets T_0 run. Task T_0 calls *wait(5 ns)* from the OS interface to notify the OS about its execution delay. The OS then internally advances the simulated time using the underlying SLDL kernel, suspends task T_0 and unblocks task T_1 to resume its execution. As such, tasks are interleaved at boundaries of *"wait for time"* primitives to replicate a sequential execution. In the following sections, we present how this behavior can limit the accuracy and speed of preemptive simulation of host-compiled models.

3.2 Quantitative Study of Accuracy/Speed Tradeoffs

In this section, we compare the execution behavior of randomly generated artificial task sets running on a conventional host-compiled simulator

and on a virtual reference platform to study the speed and accuracy of host-compiled models.

Target Platform. We simulate our target application on a dual-core MIPS34Kc Malta target platform running a 2.6.24 Linux SMP kernel, which realizes a partitioned queue scheduling scheme and is configured with preemption and high resolution timers. We compare our host-compiled model of this platform against an instruction-accurate reference ISS running the actual Linux binary [64].

Application Model. We use the setup presented in [12] to generate our artificial, periodic task sets. Task periods are uniformly distributed over [10, 100] ms, while task utilizations are distributed over [0.001, 0.1], [0.1, 0.4], and [0.001, 0.4], i.e. in the small (S), large (L) or mixed/medium (M) range of execution delays. For each task set, we generate tasks until a maximum is reached or the core utilization falls into the range [0.3, 0.6), [0.6, 0.8) or [0.8, 1) for light, medium, or heavy task loads, respectively. Tasks priorities are assigned inversely to their periods following a rate-monotonic scheduling strategy. Actual task delays are measured on the reference simulator and back annotated into our model at different levels of timing granularity.

Sources of Errors. Model error is measured as the average absolute difference in individual task response times over all tasks and task iterations. Generated task sets and resulting modeling accuracies are summarized in Table 3.1 and Table 3.2. Results show that with a timing granularity of 10 μs and 100 μs the average timing error across all task sets is less than 0.4% and

Table 3.1: Average error of small task sets under different timing granularities.

Task Set	**S1**		**S2**		**S3**		**S4**		**S5**	
Core ID	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1
# of Tasks	7	6	11	10	9	9	12	15	13	16
Total Util.	.33	.3	.47	.45	.56	.5	.7	.69	.84	.87
Avg Task Util.	.047	.05	.043	.045	.062	.056	.058	.046	.064	.054
Avg Err $(1\mu s)$	.58%	.30%	.39%	.73%	.64%	.38%	1.02%	.88%	1.10%	1.12%
Avg Err $(10\mu s)$	.54%	.29%	.33%	.68%	.57%	.35%	.88%	.81%	.98%	.98%
Avg Err $(100\mu s)$	.94%	.59%	1.51%	1.78%	1.17%	0.58%	2.74%	1.77%	2.04%	3.57%
Avg Err $(1000\mu s)$	8.56%	3.55%	10.0%	10.1%	8.58%	4.09%	16.8%	12.6%	13.2%	15.6%

Table 3.2: Average error of medium and large task sets under different timing granularities.

Task Set	**M1**		**M2**		**M3**		**M4**		**L1**		**L2**		**L3**	
Core ID	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1
# of Tasks	4	4	4	4	4	3	3	3	3	4	3	2	3	3
Total Util.	.54	.56	.69	.64	.71	.7	.86	.69	.63	.44	.64	.63	.88	.92
Avg Task Util.	.137	.139	.173	.16	.176	.235	.286	.23	.209	.109	.212	.315	.294	.306
Avg Err $(1\mu s)$	.24%	.06%	.25%	.55%	.35%	.16%	.12%	.09%	.14%	.09%	.08%	.14%	.07%	.11%
Avg Err $(10\mu s)$	.21%	.04%	.25%	.53%	.23%	.15%	.11%	.09%	.16%	.08%	.08%	.13%	.04%	.13%
Avg Err $(100\mu s)$	.32%	.42%	.26%	1.97%	1.05%	.44%	.31%	.12%	.26%	.23%	.50%	.14%	.43%	1.02%
Avg Err $(1000\mu s)$	3.1%	5.1%	1.4%	19.3%	11.4%	3.2%	2.8%	.81%	2.4%	2.9%	4.7%	.63%	4.3%	9.1%

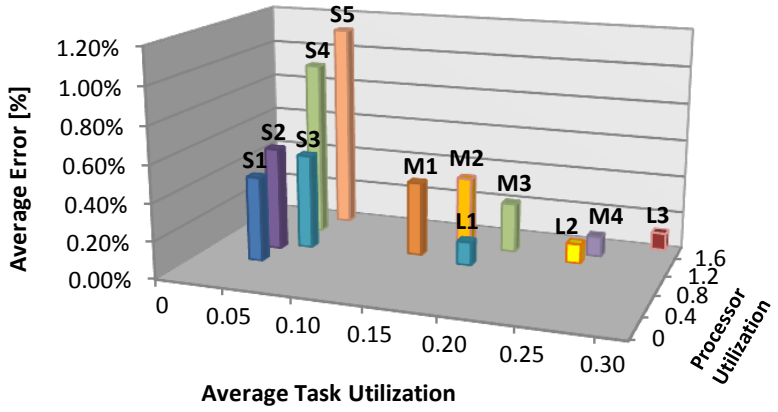


Figure 3.2: Average error in average response time of 1μs task sets.

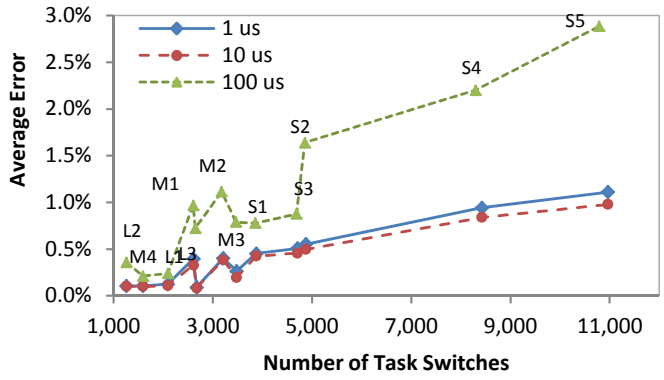


Figure 3.3: Average timing error over number of task switches.

1%, respectively. In general, timing error grows with increasing timing granularity. Figure 3.2 depicts the average error for all task sets under the timing granularity of 1 μs. The graph shows that for sets with smaller tasks and higher CPU loads a higher error is measured.

Context-switch overhead: The measurement of context switch overhead (Figure 3.3) shows that the timing error is a function of the number of task switches, and it is higher for task sets with a large number of small tasks. This

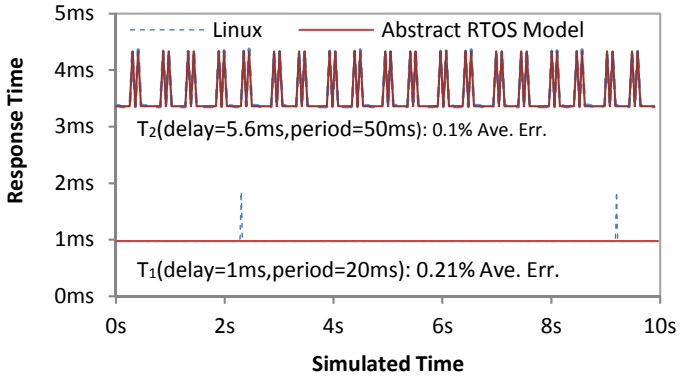


Figure 3.4: Response time traces for M1 task set.

Table 3.3: Simulation time (speed) per task set.

Task Set	1μs	10μs	100μs	1000μs
S1	4.1s (240 MIPS)	0.36s (2800 MIPS)	0.07s (14.3 GIPS)	0.03s (33 GIPS)
S2	5.0s (200 MIPS)	0.59s (1700 MIPS)	0.08s (12.5 GIPS)	0.05s (20 GIPS)
S3	6.1s (160 MIPS)	0.60s (1700 MIPS)	0.08s (12.5 GIPS)	0.04s (25 GIPS)
S4	12.0s (80 MIPS)	0.85s (1200 MIPS)	0.14s (7.1 GIPS)	0.04s (25 GIPS)
S5	10.5s (90 MIPS)	0.95s (1000 MIPS)	0.16s (6.2 GIPS)	0.08s (12 GIPS)
M1	7.6s (130 MIPS)	0.61s (1600 MIPS)	0.10s (10 GIPS)	0.03s (33 GIPS)
M2	7.5s (130 MIPS)	0.78s (1300 MIPS)	0.12s (8.3 GIPS)	0.03s (33 GIPS)
M3	7.6s (130 MIPS)	0.86s (1200 MIPS)	0.12s (8.3 GIPS)	0.04s (25 GIPS)
M4	9.3s (110 MIPS)	0.89s (1100 MIPS)	0.16s (6.2 GIPS)	0.03s (33 GIPS)
L1	5.9s (170 MIPS)	0.74s (1300 MIPS)	0.12s (8.3 GIPS)	0.03s (33 GIPS)
L2	7.4s (130 MIPS)	0.72s (1300 MIPS)	0.10s (10 GIPS)	0.02s (50 GIPS)
L3	10s (100 MIPS)	1.08s (920 MIPS)	0.13s (7.7 GIPS)	0.04s (25 GIPS)

is due to context switch delays, which are not included in the models.

Back-ground tasks: Another source of error is the non-ideal behavior of the real Linux kernel. Figure 3.4 plots the response times of the two highest-priority tasks in the M1 task set on the Linux kernel and the host-compiled simulator. As can be seen, on the Linux kernel, the highest priority task is interrupted at regular intervals by additional, unknown background activities.

Speed Evaluation. Table 3.3 shows the simulation runtimes of the models for each task set. We run each task set for 10 s of simulated time. At a nominal rate of 100 MIPS simulated by the reference ISS, this corresponds to 1000 million NOP instructions on each core for artificial delay loops and any idling. Simulations each runs for about 30 s of wall time on the reference ISS. By contrast, our model simulates such a non-functional, delay-only setup in faster than real time with a throughput of more than 1000 MIPS per core (or 2000 MIPS for the whole dual-core system) at timing granularities of 10 μs.

To summarize, we plot average accuracy and speed over all task sets

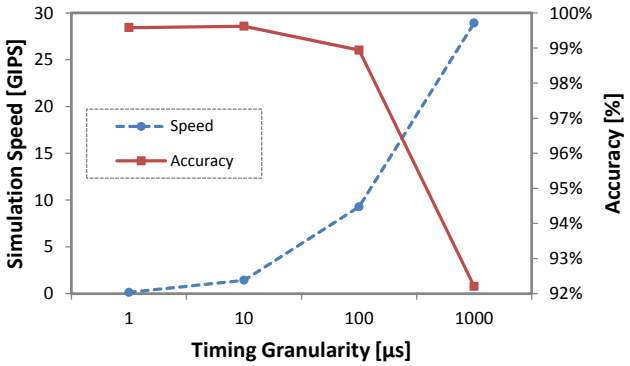


Figure 3.5: Accuracy and speed comparison for the artificial task sets.

for different timing granularities in a single graph shown in Figure 3.5. As can be seen, there is a fundamental tradeoff between accuracy and speed. A faster simulation is achieved by a coarser-grained simulation, but it comes with a loss in accuracy. This means that designers require to carefully select a proper granularity to meet the desired accuracy while still maintaining the benefits of a fast simulation.

3.3 Analytical Error Model for Preemptive Simulations

As we saw in Section 3.1, the OS model simulates task execution delays using underlying SLDL primitives whenever the running task calls a *"wait for time"* method from the OS API. In this way, a scheduler can be called after advancing the simulation time to allow for preemption of the current task by any higher priority task that became available in the meantime. Figure 3.6 shows modeling errors in such approaches. In this example, two tasks are running on the system and their execution delays are divided into equal intervals.

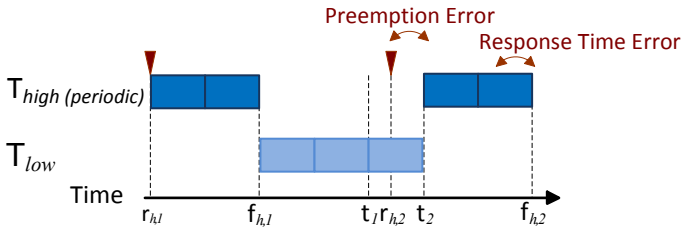
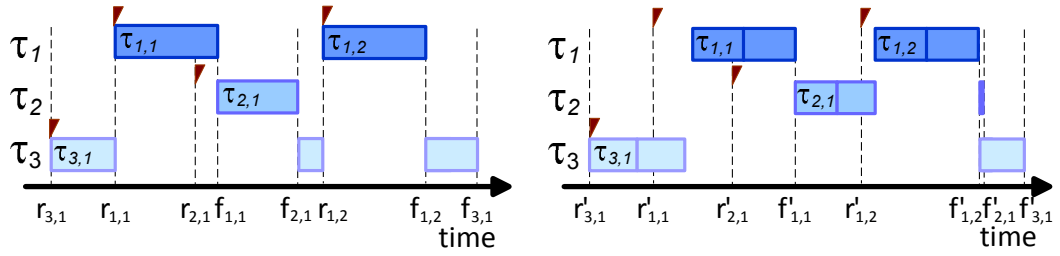


Figure 3.6: Preemption error model.

Task T_{high} is running periodically, therefore task T_{low} is scheduled after the finish time of τ_{high} jobs. At time t_1, the OS scheduler executes T_{low}, since it is the only ready task in the system. At time $r_{h,2}$, T_{high} becomes ready but the scheduler is not called until time t_2. Consequently, the start time of T_{high} is delayed and the lower priority task is preempted at a wrong time. As such, errors in the preemption model are a direct function of the back-annotated application-level timing model. Large granularities result in fast simulation, but may lead to preemption points being shifted by a large delay.

In the following, we focus on analyzing preemptive scheduling behavior. Without loss of generality, we consider an ideal model that abstracts away other effects, where an application is composed out of a set of periodic real-time tasks $\Gamma = \{\tau_1, \tau_2 \ldots \tau_n\}$, ordered by decreasing priorities P_i. Each task τ_i is described by its period T_i and execution time C_i. The j^{th} instance (job) of task τ_i is denoted by $\tau_{i,j}$, and its response time $R_{i,j}$ is measured as the time elapsed between its release time $r_{i,j}$ and the time $f_{i,j}$ when it finishes execution of one iteration. For host-compiled simulation, task execution delays are assumed to be modeled by timing values back-annotated at a granularity



(a) Theoretical order. (b) Simulation order.

Figure 3.7: Preemption error models in host-compiled simulation.

of δ_i ($C_i = \delta_i * n$). To model preemptions, the OS scheduler is called at the end of each δ_i interval (or when a task makes an explicit OS kernel call).

We can analyze the accuracy of host-compiled simulation by evaluating the response time of each task and measuring the percentage of error using the following equation:

$$err_i = \frac{|R_i(model) - R_i(ideal)|}{R_i(ideal)} \tag{3.1}$$

Figure 3.7(a) shows the theoretical execution of three periodic tasks in which the first job $\tau_{3,1}$ of task τ_3 is preempted by the higher-priority job $\tau_{1,1}$ at times $r_{1,1}$ and $r_{1,2}$. While τ_1 is in its first iteration $\tau_{1,1}$, a medium-priority task τ_2 is released (at time $r_{2,1}$) and gets executed once $\tau_{1,1}$ finishes (at time $f_{1,1}$). Subsequently, $\tau_{3,1}$ resumes its execution only after both higher priority jobs finish (at time $f_{2,1}$).

In Figure 3.7(b), the host-compiled simulation of the same task set is shown. In this model, execution delays of tasks are divided into discrete intervals. Since the OS scheduler is only called at the end of each advance in

time, the start of $\tau_{1,1}$ is delayed until the end of the current time interval of $\tau_{3,1}$. Consequently, the start of $\tau_{2,1}$ is also shifted by an equal amount to the delayed time $f'_{1,1}$ at which $\tau_{1,1}$ finishes. As a result, the next job of τ_1 now starts in the last time interval of $\tau_{2,1}$, and $\tau_{2,1}$ gets preempted by $\tau_{1,2}$ as soon as its last time interval expires. Therefore, $\tau_{2,1}$ can not finish executing its last block of code, complete its job and return control to the OS kernel until it is resumed at time $f'_{1,2} = f'_{2,1}$ when $\tau_{1,2}$ finishes[†]. In other words, $f'_{2,1}$ is now determined by $f'_{1,2}$.

As shown in this example, we can consider three sources of errors in response times. In a first scenario, the start time of a job of a higher priority task τ_i is delayed by a lower priority task τ_j running at its release time. The largest delay is achieved when the higher priority job is released simultaneously with the start of a new time interval of the lower priority job. As such, the maximum error can be bounded from above by:

$$err_i \leq \max_{j=i+1}^{n}(\delta_j)/C_i \tag{3.2}$$

In a second scenario, the start time of a job of a lower priority task $\tau_{j,l}$, which is released when a job of a higher priority task $\tau_{i,k}$ is running, is determined by the finish time of the higher priority job. Hence, if the start time of $\tau_{i,k}$ is shifted due to scenario one or two, the start time of $\tau_{j,l}$ is shifted equally. As such, the maximum error in this scenario is equal to the maximum

[†]Note that if the last code block is instead moved to before the last time advance, a job can conversely finish too early by an equal amount.

error derived for scenario one or two of all higher priority tasks:

$$err_j \leq \max_{i=1}^{j-1}(err_i) = \max_{i=1}^{n}(\delta_i)/C_j \tag{3.3}$$

Finally, the third scenario happens when the start time of $\tau_{j,l}$ is delayed due to scenario one or two such that a job of a higher priority task $\tau_{i,k}$ becomes active while $\tau_{j,l}$ is executing its last time interval, as is the case for $\tau_{2,1}$ in Figure 3.7(b). In this scenario, the finish time of $\tau_{j,l}$ is determined by the finish time of all higher priority tasks released while any higher priority task is running, i.e. until $\tau_{j,l}$ can get resumed. As such, the amount of error depends on the system load and, assuming a well-behaved system, is only limited by the task's next deadline:

$$err_j \leq (T_j - C_j)/C_j \tag{3.4}$$

As demonstrated by this analysis, the error in simulated task response times in scenarios one and two is a function of the modeled timing granularity. However, the possibility of large errors in the third scenario severely limits host-compiled simulators for evaluating real-time performance.

3.4 Summary

In this chapter, we studied the accuracy and speed of conventional host-compiled simulators on a suite of artificial task sets. We observed simulation speeds of more than 1000 MIPS with less than 1% timing error for 10 μs timing granularity. Overall, this shows the efficiency of host-compiled simulators for

supporting rapid and early embedded software development and exploration. However, the inherent speed and accuracy tradeoff and large preemption error bounds of special cases make this approach inefficient for real-time performance evaluation.

In the next chapter, we present an OS model that internally keeps the state of tasks running on the system and employs a novel timing model to automatically adjust the granularity and call its scheduler at right preemption points. In such a way, accuracy is improved while performance is maintained.

Chapter 4

OS Modeling

Multi-core processor architectures have opened new software development challenges, such as ensuring correct communication and synchronization between tasks running concurrently on different cores. In traditional multi-processor setups, tasks are assigned to different processors in an asymmetric multi-processing (AMP) fashion. Each processor has an independent address space and is managed by a dedicated OS. With each processor acting like a conventional single-core machine, this approach allows for easy adaptation of legacy applications. By contrast, in a multi-core setup, each processor can in turn contain multiple cores that share a common set of resources and are managed by a single OS. Tasks execute on different cores under a shared memory model in a symmetric multi-processing (SMP) context. With ever growing complexities and software content, a crucial component for the performance of real-time systems is the operating system (OS) kernel.

In this chapter*, we introduce a high-level, highly flexible and configurable host-compiled OS model to enable early, fast and accurate software

*Contents of this chapter previously appeared in the following paper:
P. Razaghi and A. Gerstlauer. Automatic timing granularity adjustment for host-compiled software simulation. *ASPDAC*, January 2012.

exploration in an SMP context. Our proposed model enables designers to easily and rapidly explore different scheduling parameters through straightforward adjustment of application and OS parameters. We further propose an *automatic timing granularity adjustment* (ATGA) approach, in which the OS model continuously monitors system state to automatically adjust simulated timing granularities and eliminate task scheduling errors while maintaining fast simulation speed. In such an approach, designers need not be concerned with manually selecting a proper granularity for optimizing the speed and accuracy tradeoff. Instead, the ATGA model automatically, continuously and dynamically adjusts to changing system conditions in order to achieve an optimized simulation.

This chapter is organized as follows: In Section 4.1, we first demonstrate that how a host-compiled application is integrated into the OS and complete simulator. Next, in Section 4.2, we elaborate the internal structure of our OS model. Thereafter, we focus on the proposed timing management model in Section 4.3. Finally, we present a preliminary evaluation of our approach and models in Section 4.4.

4.1 Application Setup and Integration

In host-compiled simulators, application code is captured at the source level in order to achieve a fast simulation and eliminate low-level implementation details. In the following, we present the simple and canonical parallel programming model used to develop applications and integrate them into our

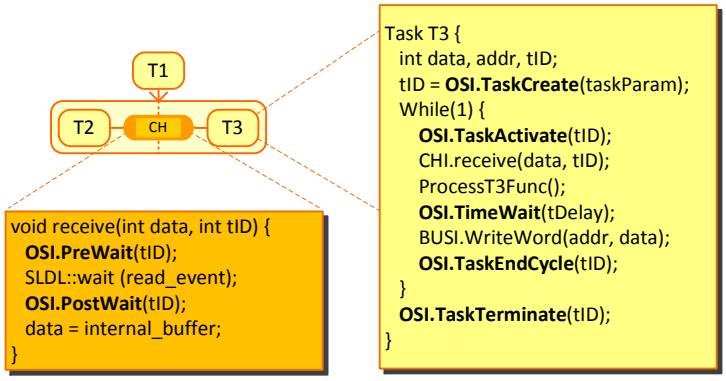


Figure 4.1: Application model for host-compiled simulation.

host-compiled simulator. In this approach, a designer only need to describe the functionality of application tasks. To describe inter-task communication, the simulator provides a comprehensive library of standard communication primitives and channels. In Figure 4.1, a model of a typical application task and an internal implementation of a communication channel are shown, which are further detailed in the rest of this section.

4.1.1 Task Modeling

As discussed in Section 1.2, a host-compiled simulator is developed over a standard SLDL. Accordingly, application tasks are modeled as high-level, hierarchical SLDL processes, which are connected to the simulator via the underlying OS application program interface (API), shown in Figure 4.2. Task behavior is described by conventional C functions and their target-specific execution delays are back-annotated into the code once at compile time. The source code is instrumented with the required timing information using `TimeWait()`

```
1   /* OS initialization and startup */
2   void Init(OSPARAM param);
3   void Start(void);
4   /* Task management */
5   int TaskCreate(TASKPARAM param);
6   void ParStart(int taskID); /* fork */
7   void ParEnd(int taskID); /* join */
8   void TaskActivate(int taskID);
9   void TaskSleep(int taskID);
10  void TaskResume(int taskID);
11  void TaskEndCycle(int taskID);
12  void TaskTerminate(int taskID);
13  /* Delay modeling and event handling */
14  void TimeWait(long long nSec, int taskID);
15  void PreWait(int taskID);
16  void PostWait(int taskID);
17  void PostNotify(int taskID, int blockedTaskID);
18  /* Interrupt handling */
19  void IntrTrigger(int intrID);
20  int CreateIntrHandler(int coreID);
21  void IEnter(int coreID, int handlerID);
22  void IReturn(int coreID);
```

Figure 4.2: High-level OS interface.

methods of the OS API.

During the system startup phase, the `Init()` method initializes the OS

model data structures and defines the OS parameters, including the number of supported cores and the default simulation quantum. The `Start()` method is then used to enter multi-core scheduling after all tasks have been attached to the model.

During the task creation phase, tasks are added to the OS by calling `TaskCreate()`, which allocates an internal representation inside the OS model. Furthermore, this method allows application designers to explore a wide range of tasks properties and behavior. The currently supported parameters are listed as follows:

- *Type:* each task can be defined exclusively as an aperiodic, a periodic, or a kernel special task model;
- *Period:* valid only for periodic tasks;
- *Priority:* the static priority level of tasks;
- *Time Slice:* time interval that a task is allowed to execute without preemption by other tasks with the same priority;
- *Affinity:* a bitmap representing the set of cores allowed to execute the task;
- *Initial Core:* the initial core that is allowed to run the task at start time.

The `TaskCreate()` method returns a unique ID, which is passed to the OS kernel in all following task-related API calls.

At the start of simulation, task threads are spawned via the SLDL. They then register themselves with the OS via a call to the `TaskActivate()` method at the beginning of their execution. This allows the OS model to collect all threads and enter them into the scheduler. At the end of their execution,

tasks remove themselves from the OS kernel by calling `TaskTerminate()`. If supported by the underlying SLDL, tasks can fork children and temporarily remove themselves from OS scheduling (`ParStart()`) until all children are collected on the SLDL level (`ParEnd()`).

Finally during the execution, a task can remove itself from the active core temporarily by either calling `TaskEndCycle()` or `TaskSleep()`. The former moves the calling periodic task into idle mode until its next release time. The latter puts the task in sleep mode until another task calls a corresponding `TaskResume()` method.

4.1.2 Channel Library

Inter-task communication is implemented by one-way or two-way message passing channels. Channels are described using the underlying SLDL event/notify primitives, which are wrapped into OS APIs that will allow the OS to accurately control the execution order of tasks connected to the channel. Designers will typically not have to deal with event handling directly. Instead, the simulator provides a reimplementation of a rich library of communication channels and primitives that are properly hooked into the OS model.

As shown in Figure 4.1, each *"wait for event"* statement is encapsulated by `PreWait()` and `PostWait()` methods. `PreWait()` simply removes the running task from the OS kernel and lets the scheduler start the next ready task on the current core. As soon as the event is captured, `PostWait()` returns the blocked task into the ready state and waits until the task is scheduled by the

OS kernel.

In point-to-point communication channels, where sender and receiver of messages are fixed during the simulation, *"notifying an event"* is followed by a call to an OS `PostNotify()` method. If the just unblocked task has a priority higher than the current one, `PostNotify()` will immediately perform a task switch and call the OS scheduler. In this way, a more accurate task scheduling is modeled, since high priority unblocked tasks are not required to wait until the next point that the scheduler is called.

4.2 Abstract OS Modeling

We extend existing approaches to develop a multi-core SMP OS model that manages the scheduling and dispatching of application tasks across available queues and cores. In doing so, the OS model wraps around the basic SLDL event handling mechanism, replacing SLDL primitives with calls to the OS interface instead and ensuring that at any time only as many SLDL threads as there are cores are active. Figure 4.3 depicts an overview of our abstract multi-core OS model, which is designed to perform three main functionalities: task management, multi-core scheduling emulation, and coordination of the simulated timing model.

During its execution, each simulated task can be in five states, and tasks move to different states by calling a corresponding OS API method. In order to emulate the state of the system, the OS model maintains tasks in five internal queues: a *Ready* queue holds tasks that are ready to execute

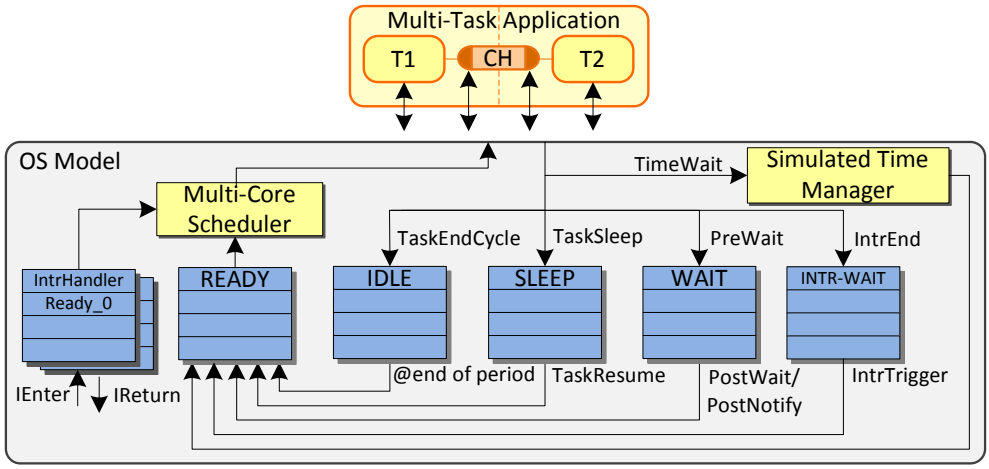
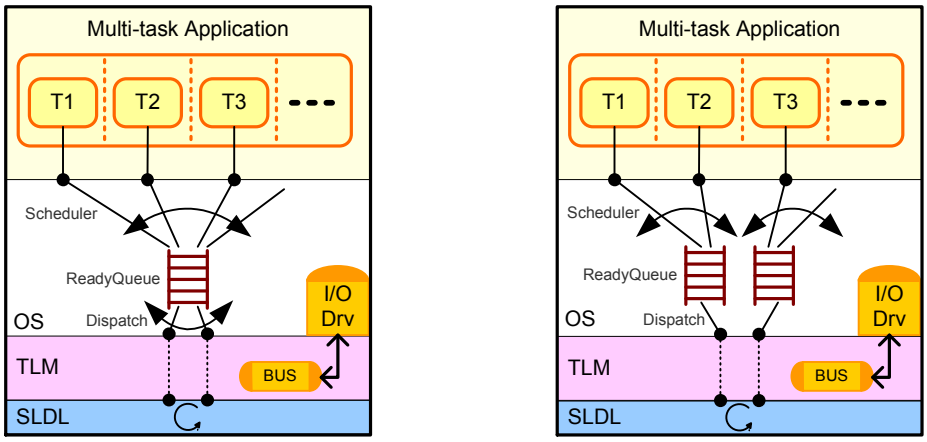


Figure 4.3: Abstract OS model internals.

and is sorted based on a user-defined scheduling policy. An *Idle* queue holds periodic tasks that have called the kernel's `TaskEndCycle()` method. The *Idle* queue is ordered based on the release time of each task's next iteration. Idle tasks are retrieved from the head of queue and placed in the *Ready* queue by the OS kernel at the start time of their next period. A *Sleep* queue holds tasks that have been suspended until they are resumed again. Tasks waiting for an event are suspended and transferred to a *Wait* queue. As part of modeling the top half of the OS interrupt handling chain, an *IntrWait* queue holds special interrupt tasks until a core-specific interrupt handler calls the `IntrTrigger()` method to move them into the *Ready* queue. Since the core-specific interrupt handlers are treated as special high-priority tasks by the OS scheduler, a separate *IntrHandlerReady* queue is dedicated to each core. Interrupt handlers are activated and moved into a corresponding queue when the `IEnter()` method is called by the processor interrupt interface. After



(a) Global queue scheme. (b) Partitioned queue scheme.

Figure 4.4: Host-compiled SMP OS models.

triggering interrupt tasks, interrupt handlers deactivate and remove themselves from the *IntrHandlerReady* queue by calling the `IReturn()` method.

In the context of SMP scheduling, there are two major task scheduling schemes distinguished by the number of *Ready* queues associated with each core: partitioned and global scheduling schemes. In a *partitioned scheme*, each core has a separate *Ready* queue and tasks are initially assigned to a fixed queue. The scheduler picks tasks for a core only from the associated queue. In a *global scheme*, the scheduler maintains only a single *Ready* queue and tasks can be freely assigned to the next available core. A global queue can lead to a better utilization but is less scalable and may result in degraded performance due to cache pollution when tasks move between cores too frequently [46]. Our OS model supports both scheduling schemes, such that designers can explore the right structure for a particular application. Depending on the scheduling

```
Function Scheduler (int coreID):
 1    queueID := GLOBAL_SCHEDULING ? ø : coreID
 2    oldTask := runningTask[coreID]
 3    runningTask[coreID].RemainingTimeSlice − = (CurrentTime() - runningTask[coreID].StartTime)
 4    runningTask[coreID].State := Ready
 5    if runningTask[coreID].RemainingTimeSlice ≤ ø then
 6      runningTask[coreID].RemainingTimeSlice := runningTask[coreID].TimeSlice
 7      FIFO(ReadyQueue[queueID], runningTask[coreID])
 8    else
 9      LIFO(ReadyQueue[queueID], runningTask[coreID])
10    endif
11    Dispatch(coreID)
12    Wait4Sched(oldTask)
```

Figure 4.5: Multi-core OS scheduler.

scheme, a single *Ready*, *Idle*, *Sleep*, *Wait* and *IntrWait* queue is either shared among all cores as depicted in Figure 4.3 and Figure 4.4 (a), or multiple such queues are replicated one per core as shown in Figure 4.4 (b). In the following, we present further details on the internals of the OS' scheduler implementing fixed-priority, FIFO, and round-robin scheduling policies. We have further implemented more sophisticated scheduling policies such as Pfair [87] and G-EDF [1], details of which are discussed in Appendix 1.

The core component of the OS model is a replicated, generic multi-core scheduler, the body of which is shown in Figure 4.5. The scheduler is

an internal function of the OS model and is called by the OS API methods whenever a task switching is possible or required. The main functionality of the scheduler is to retire the currently active task on a core, if any, and place it in a proper place in the right *Ready* queue. We utilize a time slice notion to model FIFO or round-robin (RR) scheduling among tasks that have the same priority. When the OS runs the scheduler for a specific core, it calculates the remaining time slice of the current active task on the desired core. This is done by subtracting the time consumed by the task, which is computed as the difference between the current simulated time and the time the task was last put onto the core (line 3 in Figure 4.5). Then, the current task is moved to the corresponding *Ready* queue based on the new value of the time slice. If the time slice reaches zero, the task is added at the end of its priority list where it will be scheduled after all current ready tasks with the same priority. In addition, the task's remaining time slice value is reset back to its configured value. Otherwise, the task will be placed back at the beginning of the priority list from where it will be scheduled immediately again, right before any other ready tasks with the same priority. Consequently, in RR scheduling the value of the time slice defines the portion of time that every task is allowed to be executed without any preemption by tasks of the same priority, while setting an infinite time slice value will result in a FIFO scheduler.

At the end of the scheduler, a `Dispatch()` function will be called to assign a new task to the current core. Figure 4.6 shows the implementation of `Dispatch()`. As a first step, if there is an active interrupt on the core, a

```
Function Dispatch (int coreID):
1    if !Empty(IntrHandlerReadyQueue[coreID]) then
2      IntrHandlerID := PeekFirst(IntrHandlerReadyQueue[coreID])
3      runningTask[coreID] := IntrHandlerID
4      IntrHandlerList[IntrHandlerID].State := RUN
5      SendSched(IntrHandlerID)
6    else
7      OSDispatch(coreID)
8    endif
```

Figure 4.6: Multi-core task dispatcher.

corresponding interrupt handler (see Section 5.2) will be assigned to that core (lines 2-5) instead of a regular task. The aforementioned core-specific interrupt handler queues (*IntrHandlerReady*) thereby manage the priorities and selection among pending interrupts if multiple interrupt vectors are supported by the modeled processor.

If there are no pending interrupts, an OS-specific dispatcher will be called to assign normal tasks to the core. We present the implementation details of the `OSDispatch()` function for both global and partitioned queue models (Figure 4.7). In both cases, the function selects the highest priority task in the *Ready* queue and assigns it to run on the current core. Ready queues are sorted by task priorities, and tasks with the same priority are arranged based on time slices, as described above. In case of partitioned queues (Figure 4.7 (a)), a load balancing to optionally, e.g. at regular intervals,

```
Function OSDispatch (int coreID):
 1   queueID := coreID
 2   LoadBalance(coreID)
 3   if !Empty(ReadyQueue[queueID]) then
 4    runningTask[coreID] :=
 5          getFirst(ReadyQueue[queueID])
 6    runningTask[coreID].State := RUN
 7    runningTask[coreID].StartTime :=
 8                          CurrentTime()
 9    SendSched(runningTask[coreID])
10   else
11    runningTask[coreID] := Null
12   endif
```

(a) Partitioned-queue scheme.

```
Function OSDispatch (int coreID):
 1   ReadyQueue_Lock.Acquire()
 2   runningTask[coreID] := Null
 3   if !Empty(ReadyQueue[ø]) then
 4    activeTask := getFirst(ReadyQueue[ø], coreID)
 5    if activeTask ! = Null then
 6     runningTask[coreID] := activeTask
 7     runningTask[coreID].State := RUN
 8     runningTask[coreID].StartTime := CurrentTime()
 9     SendSched(runningTask[coreID])
10    endif
11   endif
12   ReadyQueue_Lock.Release()
```

(b) Global-queue scheme.

Figure 4.7: OS dispatcher.

migrate tasks between queues is performed before dispatching. In case of a global queue structure (Figure 4.7 (b)), a semaphore controls all accesses to the shared queue. Note that in both schemes, the tasks to be migrated or the first task allowed to run on a particular core are selected (in the queue's `getFirst()` method, line 4) based on the task affinity. In other words, the current core can be idle even if the *Ready* queue is not empty.

After selecting a new task, the dispatcher releases it by calling `SendSched()` to notify an SLDL event associated with the chosen task. After returning from

the `Dispatch()` call at the end of the scheduler, the current task on the given core in turn suspends itself on its own event. Leveraging SLDL events assigned to each task, this emulates actual context switches. Note that if no higher priority or other sibling task is available, the current task may simply dispatch itself and be immediately triggered again.

4.3 Timing Model Management

In addition to basic OS services, the OS model simulates task execution delays using underlying SLDL primitives whenever the running task calls a `TimeWait()` method. In conventional models, the granularity of delays is defined by the application code. The scheduler is only called after advancing the simulation time to allow for preemption of the current task by any higher priority task that became available in the meantime. As described in Section 3.3, this limits the possibility of accurately modeling preemptions. In the following section, we introduce a novel timing model, which predicts possible preemption points and automatically adjusts the granularity of user-defined delays in order to call the scheduler at right preemption points. This approach is only applicable to fully predictable applications, such as periodic systems. Thereupon, we extend such a timing management approach to support compete SoCs running sporadic tasks with hardware interactions.

```
Function PredictNextPreemptionTime (task runningTask):
1    predictedDelay := SIMULATION_QUANTUM
2    scheduledCoreID := GetSchedCoreID(runningTask)
3    queueID := GLOBAL_SCHEDULING ? ∅ : scheduledCoreID
4    for all idleTask in IdleQueue[queueID] do
5      if idleTask.Priority ≥ runningTask.Priority and AllowedRun(idleTask, scheduledCoreID) then
6        predictedDelay := idleTask.NextPeriodTime - CurrentTime()
7        return predictedDelay
8      endif
9    endfor
```

Figure 4.8: OS predictive mode.

4.3.1 Predictive Timing Model

The key idea for removing the preemption error is to predict the next possible preemption point and invoke the scheduler at the proper time. Figure 4.8 shows the algorithm for predicting the next preemption time among periodic tasks. Since the *Idle* queue is sorted based on the tasks' next release times, the preemption point is defined by the first task with a priority higher than the currently running one. Note that for a global scheduling scheme, the `AllowedRun()` method checks if a core is allowed to run a task based on the task's user-defined affinity.

Figure 4.9 shows the pseudo code of the `TimeWait()` method in the predictive timing model. It first computes the adjusted time delay by calling the method to predict the next preemption point (line 4). Since the exact

```
Function TimeWait (long long nsec, task runningTask):
 1   remainedDelay := nsec
 2   scheduledCoreID := GetSchedCoreID(runningTask)
 3   while remainedDelay > ø do
 4     adjustedDelay := PredictNextPreemptionTime(runningTask)
 5     if ( !Empty(WaitQueue(scheduledCoreID)) ) then
 6       adjustedDelay := defaultDelayGranularity
 7     endif
 8     adjustedDelay := Min( adjustedDelay, remainedDelay)
 9     remainedDelay − = adjustedDelay
10     SLDL::wait(adjustedDelay)
11     Scheduler()
12   endwhile
```

Figure 4.9: Predictive timing model.

preemption point is unknown whenever a task is waiting for an external event, the OS kernel in this case falls back to a user-defined default timing granularity (lines 5 to 7). After advancing the simulation time by the adjusted delay (line 10), the OS scheduler is called to perform a context switch and block the current task until it is scheduled again (line 11). This loop continues until the user-defined delay is consumed. As can be seen, the designer does not need to settle on a granularity for back-annotated delays. The OS kernel itself breaks delays into a number of smaller steps as needed, in order to automatically provide the best timing granularity for accurate results.

4.3.2 Automated Timing Granularity Adjustment (ATGA)

In this section, we describe our automatic timing granularity adjustment (ATGA) approach, which is an expansion of the predictive timing model. In this approach, the OS kernel switches between *predictive* and *fallback* modes to call the scheduler at the right preemption points. In the following, we demonstrate the details of each mode and the mechanism that the OS model uses to automatically control the underlying timing model.

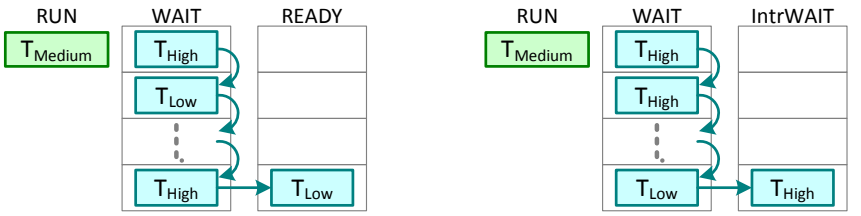
Generally, timing errors happen when a task is running and, while advancing simulation time, a higher priority tasks becomes ready without the scheduler getting a chance to immediately preempt the current one. This situation can occur in the following cases: (a) a periodic task reaches its next iteration time, (b) the interrupt handler triggers an interrupt task, or (c) a blocked or sleeping task returns to the *Ready* state when the running task notifies an event or resumes it. In such cases, the start of the newly released task is delayed until the expiration of the current time granule.

In *predictive mode*, the OS model monitors the state of periodic tasks running on the system and uses this information to predict the next possible preemption point specifically for situations in case (a). If the back-annotated granularity is larger than the predicted interval, the OS kernel adjusts the delay to invoke the scheduler at the predicted time (as discussed in Section 4.3.1).

Conversely, the exact next preemption point is unknown for cases (b) and (c), i.e. whenever a task is waiting for an internal or external event. In

these cases, the OS kernel falls back to a user-defined default timing granularity. In this *fallback mode*, the OS divides back-annotated delays into very fine granules until all events are captured and no task remains in the *Wait* queues.

The adjustive OS model lets designers select coarse-grain back-annotated delays while achieving fast and still accurate results. However, when the OS switches to fallback mode, the performance of the simulation decreases dramatically. As such, we develop techniques that exercise finer control over when to invoke fallbacks. Figure 4.10 illustrates two inter-task communication examples in which the OS kernel does not need to switch to fallback mode even though some tasks are waiting for an event. In Figure 4.10 (a), a inter-task communication chain is shown in which a set of tasks are blocked waiting for other tasks in the chain. The task at the end of the chain has a higher priority than the running task, but is blocked by a lower priority *Ready* task. Since the *Ready* task cannot be scheduled while the current task is running, remaining in predictive mode will not change the execution order of tasks. Similarly, in Figure 4.10 (b) the same chain is shown where the task at the end of the chain is a low priority task that is blocked on an external event, i.e. an interrupt.



(a) Blocked by a *Ready* task.

(b) Blocked by an interrupt task.

Figure 4.10: Inter-task communication examples.

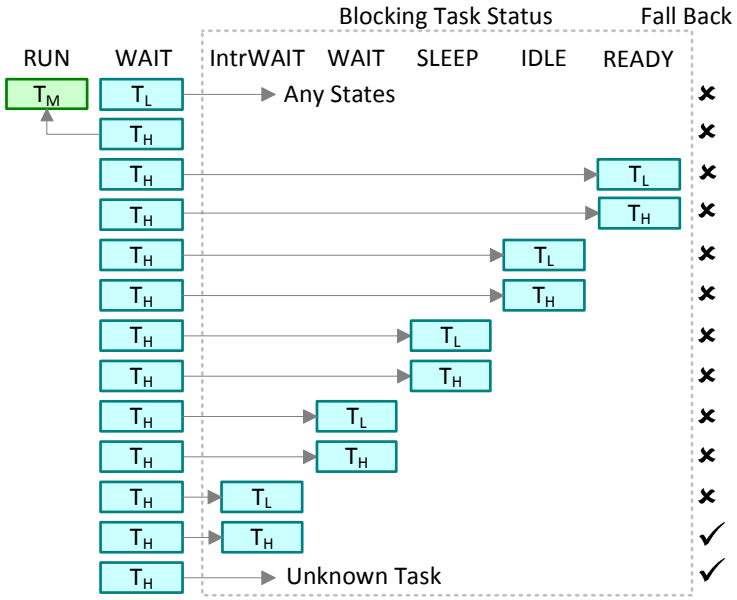


Figure 4.11: Fallback mode conditions.

Even when the interrupt occurs and assuming small interrupt task delays, unblocking the lower-priority task can never preempt the running task, i.e. the fallback mode can also be ignored in this situation.

As illustrated by these examples, only the task at the end of the *Wait* chain needs to be examined to determine the fallback condition. Figure 4.11 lists all possible situations and required fallback conditions. Generally, the OS switches to the fallback mode due to unpredicted events. Therefore, the OS moves to fallback when a task with a higher priority is in the *Wait* queue and is blocked by an unknown task or a task in the *IntrWait* queue. In all other situations, granularity of the simulation will not affect the execution order of the application tasks. Lower-priority tasks in the *Wait* queue or tasks waiting for

another task in the *Wait* queue can never affect execution of the current task. Likewise, the case of a lower-priority ready task has already been discussed, and a higher-priority task in the *Ready* queue should never exist. Situations in which a high-priority task is blocked on a periodic task in the *Idle* queue can be handled by switching to predictive mode and simulating the system at the predicted granule level. Similarly, if a higher-priority task is waiting for the currently running or a sleeping task, the preemption and context switch can be performed directly in the event notification or `TaskResume()` kernel method, right at the point when it is called by the running task. Lastly, assuming a low execution delay of interrupt tasks, we can postpone any such task if it will only trigger a low priority task in the *Wait* queue. With a minor interrupt timing error, we can ignore fallback condition. All the aforementioned conditions are checked in the enhanced `FallbackMode()` function, which is shown in Figure 4.12. In order to determine inter-task dependencies, we annotate all IPC primitives in the provided channel library to record the ID of the sending task that a receiving task is blocked on. The OS only turns to fallback mode if a higher priority task is blocked by an *Unknown* task (line 6) or is waiting for an external event (line 7).

In a multi-core context, the OS model further monitors the state of application tasks that are driven by an interrupt that is handled by a different core, such that the OS can adjust the predicted time or switch to fallback mode accordingly. In such cases, if the core handling the interrupt is responsible for releasing a high-priority interrupt-driven application task on the other core,

```
Function FallbackMode (task runningTask):
 1    currentCoreID := GetSchedCoreID(runningTask)
 2    queueID := GLOBAL_SCHEDULING ? ∅ : currentCoreID
 3    for all waitingTask in WaitQueue[queueID] do
 4      if waitingTask.Priority ≥ runningTask.Priority and AllowedRun(waitingTask, currentCoreID) then
 5        blockingTask := waitingTask.blockingTaskID
 6        if (blockingTask == Unknown) or
 7              (blockingTask.Priority ≥ runningTask.Priority
   and blockingTask.State == IntrWait) then
 8          return true
 9        endif
10      endif
11    endfor
12   return false
```

Figure 4.12: OS fallback mode.

it may have to adjust its predicted time or go to fallback mode even if it otherwise would not. In a situation in which the the other core's interrupt-driven application task is in *Idle* state, the interrupt handling can be delayed until the next release time of this periodic task without actually changing the execution sequence. Similarly, if the priority of the interrupt-driven task on the other core is lower than the priority of the task currently running there, the OS model can simply adjust the predicted time to become the next wake-up time on that other core. The scheduler wake-up time is determined by the OS model internally whenever the simulation time is advanced, i.e. it is

```
Function IntrDependencyCheck (task runningTask):
 1    currentCoreID := GetSchedCoreID(runningTask)
 2    for all otherCoreID with otherCoreID ! = currentCoreID do
 3      for all intrTask in IntrWaitQueue[otherCoreID] do
 4        if intrTask.HandledCore == currentCoreID or intrTask.HandledCore == Unknown then
 5          (adjDelay, fallback) := (∞, true)
 6          blockedTask := intrTask.blockedTaskID
 7          if blockedTask.State == IDLE then
 8            adjDelay := min(adjDelay, (blockedTask.NextPeriodTime - CurrentTime()))
 9            fallback := false
10          elsif blockedTask.Priority < runningTask[otherCoreID].Priorirty then
11            adjDelay := min(adjDelay, (runningTask[otherCoreID].NextWakeupTime - CurrentTime()))
12            fallback := false
13          endif
14        endif
15      endfor
16    endfor
17   return (adjDelay, fallback)
```

Figure 4.13: Inter-core interrupt dependency check.

guaranteed that no context switch can happen in the meanwhile. Note that the predicted time on the interrupt-handling core can not be advanced further than that. Even if there is no higher-priority task waiting on the other core right now, the task running there can choose to enter a wait state and thus change the scheduling mix at any time.

Figure 4.13 demonstrates the method that checks inter-core interrupt-dependencies and calculates a new predicted delay if needed. In order to keep track of interrupt dependencies, interrupt handler models send their core ID to interrupt tasks they trigger. The `IntrDependencyCheck()` method explores *IntrWait* queues on other cores to see if some interrupt tasks can be triggered by the current core's interrupt handler (line 4). If such a condition exists, the application task waiting for that interrupt is determined via the receiving task ID recorded using similar channel library annotations as described earlier (line 6). Finally, a new adjusted delay is calculated and fallback mode conditions are determined as described above (lines 7-13). At the end, the adjusted delay and the fallback mode check are reported back to the OS model.

Using the new timing model, user-defined timing granularities are internally divided to provide accurate task preemption. However, simulation speed is still limited by back-annotated granularities, which usually depend on various other factors, such as application code modularity. We further introduce an accumulative timing approach to achieve highest possible speed even with fine-grained back-annotated delays, while maintaining overall accuracy. In this approach, temporal accumulation and decoupling is integrated into and controlled by the OS kernel itself.

In accumulative mode, the OS kernel lets the running task execute its code and accumulate back-annotated delays without calling the scheduler and advancing simulation time. Each task has a local counter to keep track of the amount of the delay that needs to be simulated. As long as this delay is less

than the next predicted preemption point, the task continues to accumulate back-annotated delays. It only consumes delays whenever a task preemption point needs to be reached.

Timing accumulation and adjustment is only effective in predictive mode. Designers still need to decide on a timing granularity for fallback mode, which can affect speed and accuracy tradeoffs. Similarly to accumulation during predictive execution, we integrate an event-driven timing method into the fallback mode. In this setup, accumulated delays are executed even under fallback conditions, but an OS-internal event is introduced to be able to asynchronously interrupt long time consumption periods.

The pseudo code of the final `TimeWait()` method is shown in Figure 4.14. In order to achieve highest possible speed even with fine-grained back-annotated delays, the OS accumulates back-annotated delays of the current task until the next predicted preemption point is reached or the OS needs to switch into fallback mode (lines 1-4). The OS then advances the simulation time using the predicted delay, optionally utilizing an event-driven fallback check, and calls the scheduler to perform a context switch, if necessary (lines 7-15). This loop continues as long as the accumulated delay is greater than the predicted delay or the OS is still in fallback mode.

To realize fallback mode, an OS-internal event (*schedulerEvent*) is introduced that enables asynchronously interrupting long time consumption periods (line 9 in Figure 4.14). Since the fallback mode is only entered when a high-priority task is waiting for an interrupt, the *schedulerEvent* is triggered by

```
Function TimeWait (long long nsec, task runningTask):
 1    runningTask.AccDelay + = nsec
 2    FB := FallbackMode(runningTask)
 3    (adjustedDelay, ID_FB) := IntrDependencyCheck(runningTask)
 4    predictedDelay := min(PredictNextPreemptionTime(runningTask), adjustedDelay)
 5    while runningTask.AccDelay > predictedDelay or FB or ID_FB do
 6      runningTask.NextWakeupTime := CurrentTime() + predictedDelay
 7      if FB or ID_FB then
 8        startTime := CurrentTime()
 9        SLDL::wait(predictedDelay, OS::scheduleEvent)
10        predictedDelay := CurrentTime() - startTime
11     else
12        SLDL::wait(predictedDelay)
13      endif
14      runningTask.AccDelay − = predictedDelay
15      Scheduler()
16      FB := FallbackMode(runningTask)
17      (adjustedDelay, ID_FB) := IntrDependencyCheck(runningTask)
18      predictedDelay := min(PredictNextPreemptionTime(runningTask), adjustedDelay)
19    endwhile
```

Figure 4.14: ATGA timing model.

interrupt handlers in the HAL whenever an interrupt occurs (see Section 5.2). This in turn will abort the `wait()` statement in the SLDL kernel, at which point control is returned to the OS model to perform a corresponding schedul-

ing check.

All in all, this timing model provides error-free task scheduling at the highest possible speed by accumulating and dividing user-defined timing granularities to internally maintain an independently controlled, improved strategy to advance simulation time.

4.4 Experiments and Results

In this section, we evaluate our OS model and demonstrate how our novel timing management approach improves the speed and accuracy tradeoff.

4.4.1 Predictive OS Evaluation

In order to evaluate simulation speed and accuracy of our predictive timing model, we simulate a set of periodic tasks and compared the simulation performance of our OS model to a conventional one under different timing granularities. Accuracy is then analyzed by comparing results to the execution of tasks on a reference ISS [64] modeling a single-core MIPS Malta platform running a Linux 2.6 kernel configured with preemption and high resolution timers.

The application consists of randomly generated periodic task sets as introduced in Section 3.2. We run each task set for 10 s of simulated time. At a nominal rate of 100 MIPS simulated by the reference ISS, this corresponds to 1000 million no-operation (NOP) instructions.

Table 4.1: Artificial periodic task set characteristics and simulation results.

Task Set	S1	S2	S3	S4	S5	M1	M2	M3	M4	L1	L2	L3
Number of Tasks	7	11	9	12	13	4	4	4	3	3	3	3
Avg. Task Weight	.047	.043	.062	.058	.064	.136	.173	.176	.286	.21	.212	.294
CPU Utilization	.33	.47	.56	.70	.84	.54	.70	.71	.86	.63	.64	.89
Avg. Err. (1μs)	.53%	.48%	.48%	.79%	.88%	.41%	.08%	.45%	.08%	.17%	.18%	.14%
Avg. Err. (10μs)	.54%	.53%	.49%	.82%	.89%	.42%	.08%	.44%	.08%	.18%	.18%	.13%
Avg. Err. (100μs)	.95%	1.63%	.96%	3.47%	2.98%	.83%	.11%	.95%	.13%	.21%	.28%	.34%
Avg. Err. (1000μs)	5.8%	7.3%	5.6%	15.9%	12.8%	5.02%	.32%	.99%	1.2%	1.8%	.90%	2.4%
Avg. Err. P-OS	.53%	.48%	.48%	.79%	.88%	.41%	.08%	.45%	.08%	.17%	.18%	.14%
Speed [GIPS] (1μs)	0.55	0.36	0.31	0.25	0.20	0.33	0.25	0.25	0.21	0.28	0.27	0.19
Speed [GIPS] (10μs)	4.5	3.6	2.8	2.3	2.1	2.8	2.3	2.4	1.9	2.4	2.4	2.0
Speed [GIPS] (100μs)	20	20	25	17	6	14	14	14	12	12	17	10
Speed [GIPS] (1000μs)	100	50	50	50	25	100	100	100	500	100	50	50
Speed [GIPS] P-OS	107	32	33	19	18	62	107	54	85	99	103	87

We analyze the accuracy of the predictive timing management approach by comparing the response times of periodic tasks in the reference ISS with our host-compiled simulator. Delays were back-annotated into host-compiled models directly from measurements taken when running on the ISS. Model error was measured as the average absolute difference in individual task response times over all tasks and task iterations. Table 4.1 summarizes the task set properties and compares the accuracy and performance of our predictive OS (P-OS) model with that of a conventional one at four different back-annotation granularities. We can observe that the highest possible accuracy is achieved using our P-OS model. This is equivalent to a conventional model at 1μs granularity, which loses a large amount of accuracy at coarser granularities. Note that although we would expect to see zero errors on the predictive model, remaining errors are caused by OS context-switch overheads and non-ideal behavior of a real Linux system not included in our OS model

(as studied in Section 3.2). In terms of simulation performance, an average simulation speed of 67 GIPS is achieved on the P-OS model. This is 233 time faster than the original OS model at a granularity of 1μs and similar to the original model at 1ms granularity.

In the conventional OS model, designers are responsible for choosing the timing granularity to achieve acceptable accuracy and performance. However, selecting the proper granularity is not straightforward. For example, using the granularities of 1μs and 10μs, the same accuracy is provided while the former simulates 10 times faster than the latter. In addition, the lack of a reference platform for many applications makes it impossible to find a reliable granularity. Figure 4.15 plots the tradeoff between average accuracy and simulation speed over all task sets. As can be seen, decreasing the timing granularity results in a higher accuracy but comes at a loss in simulation performance. By contrast, our predictive model provides both fast and accurate results regardless of the timing granularity.

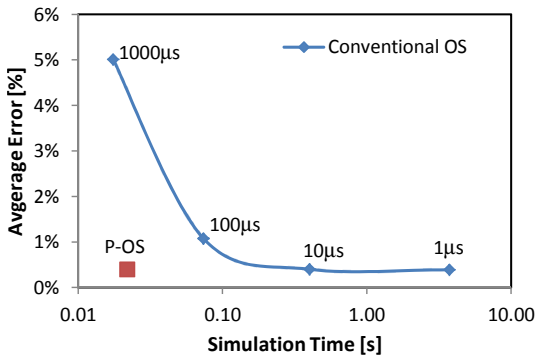


Figure 4.15: Accuracy and speed tradeoffs in the artificial task sets example.

Table 4.2: Simulation results for automotive task set.

Task	Exec. Delay	Period	Weight
susan(edge)	1.36s	4.7s	0.29
susan(smooth)	3.50s	38s	0.09
qsort	1.15s	45s	0.03
basicmath	37.26s	85s	0.44
Simulation	**Avg. Err.**	**Max. Err.**	**Speed**
OS ($1\mu s$)	0.20%	26.7%	115 MIPS
OS ($1ms$)	1.44%	26.8%	820 MIPS
OS ($10ms$)	1.64%	27.0%	825 MIPS
OS ($100ms$)	3.45%	28.5%	830 MIPS
P-OS	0.006%	0.022%	840 MIPS

In order to evaluate our approach under more realistic conditions with HW/SW interactions, we also simulate a task set composed out of a subset of applications from the automotive category of the MiBench suite [32]. Benchmarks are converted to execute periodically and concurrently based on rate-monotonic scheduling policy, where task Susan(edge) was modified to interact with an FPGA by streaming its outputs over the system bus. The resulting tasks set is simulated for 500s both on the reference ISS and in host-compiled form (with back-annotated ISS delays).

Table 4.2 summarizes benchmark features and compares the accuracy and the simulation performance for the predictive and the conventional OS models. Assuming a nominal CPI of 1 and not counting idle cycles, the P-OS model simulates at 840 MIPS. This is 7 times faster than the conventional model at 1μs, where a granularity of 100ms is required to achieve the same speed. However, the third error scenario is triggered in this setup, leading to high average and maximum errors in conventional models at any granularity.

Table 4.3: Accuracy and speed measurements for cellphone example.

	Conv. 1μs	**Conv.** 10μs	**Conv.** 100μs	**Conv.** 1000μs	**ATGA** FB 1μs	**ATGA/Acc** FB 1μs	**ATGA** ED	**ATGA/Acc** ED	**ISS**
Avg. Err. (MP3)	0.73%	0.79%	1.40%	9.65%	0.73%	0.74%	0.73%	0.74%	0%
Avg. Err. (JPEG)	7.33%	7.33%	7.33%	7.35%	7.33%	7.32%	7.33%	7.32%	0%
Avg. Err. (MP3+JPEG)	4.18%	4.20%	4.49%	8.45%	4.18%	4.18%	4.18%	4.18%	0%
Simulation Speed [MIPS]	340	790	930	1080	554	684	621	892	0.13
Simulation Time [s]	0.61s	0.26s	0.22s	0.19s	0.37s	0.30s	0.33s	0.23s	1580s

By comparison, P-OS model accuracy remains above 99%.

4.4.2 ATGA Evaluation

To further demonstrate the benefits of the ATGA approach, we apply our OS models to an industrial-strength, ARM7-based cellphone example running concurrent MP3 decoding, JPEG encoding and control tasks. The MP3 decoder runs as a periodic task with the highest priority in the system. It uses a hardware accelerator to perform real-time audio decoding. The JPEG encoder runs as an interrupt-driven task with medium priority. The control task performs user-interface actions and runs at the lowest priority. Tasks communicate with external hardware and the rest of the system via an AHB bus and 14 interrupts. In this setup, the ATGA OS model can utilize predictive mode whenever the JPEG or control task are running and the MP3 is idle. On the other hand, fallback mode is triggered whenever MP3 or JPEG tasks are waiting for an external hardware interrupt while a lower-priority task is running.

For accuracy analysis, we compare the execution of our host-compiled

simulator using the proposed OS models to a cycle-accurate ISS [53]. Task delays were back-annotated from measurements obtained from the ISS. Our testbench performs encoding of 55 MP3 frames and JPEG decoding of a 680×480 picture divided into 60 stripes. This translates into a total of 200 million simulated instructions. Model error was measured as the average absolute difference in individual frame and stripe delays over all iterations.

Table 4.3 compares the accuracy and performance of our proposed OS models with that of a conventional one at four different back-annotated granularities. ATGA and ATGA plus accumulation models are simulated with both 1μs and event-driven (ED) fallback mode. We can observe that, regardless of the granularity of the back-annotated delays, the highest possible accuracy is achieved using our ATGA approach. This accuracy is equivalent to a conventional model at 1μs, which looses accuracy at coarser granularities. Although one would expect close to 100% accuracy in ATGA models, remaining errors are due to back-annotation inaccuracies and missing of model of OS effects like timer interrupts and task context-switch overhead. As a long-running low-priority task, the JPEG encoder is adversely affected by such basic errors. On the other hand, unlike high-priority tasks such as the MP3, it is not subject to preemption errors. As such, its errors are independent of the timing granularity.

Our measurements show that the highest speed of 890 MIPS is achieved using the accumulative ATGA/Acc with an event-driven fallback mode. This model is 2.6x faster than but as accurate as the conventional one at 1μs gran-

ularity. A conventional model achieves this speed with significantly reduced accuracy at a granularity of 100μs. Results clearly show the benefits of our ATGA approach. Furthermore, both timing accumulation and event-driven fallback help to increase simulation speed without adversely affecting accuracy.

Figure 4.16 plots the average error and accuracy for different OS configurations. As can be seen, there is a fundamental tradeoff using conventional OS models. By contrast, our ATGA OS models provide both accurate and fast simulation regardless of the granularity of back-annotated delays.

4.5 Summary

In this chapter, we presented a configurable and high-level host-compiled multi-core OS model, which enables rapid and early embedded software development and design space exploration. We evaluated our models by replicating

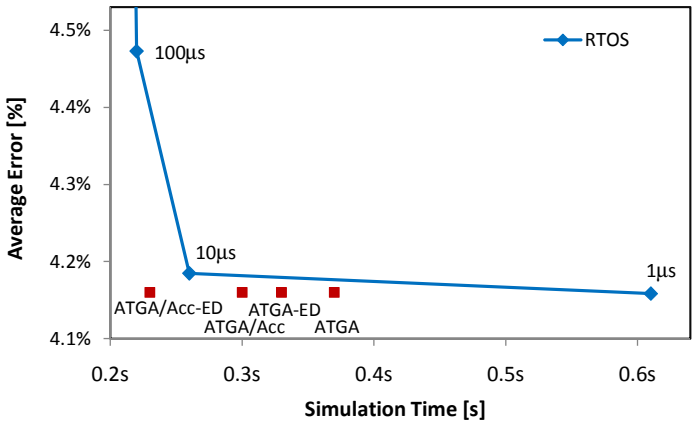


Figure 4.16: Accuracy and speed tradeoffs for cellphone example.

a fixed-priority scheduling policy on a partitioned scheme to follow a simulated Linux reference kernel. In Appendix 1, we further evaluate our OS model with integrated Pfair and G-EDF scheduling policies, and we compare our models to a real execution of an extended Linux kernel [12] replicating such scheduling policies on top of a multi-core Atom reference board.

We have further presented the integration a set of dynamic scheduling policies We further presented an automatic timing granularity adjustment (ATGA) approach for accurate and fast host-compiled simulation. In this approach, the OS model continuously monitors system states and automatically and dynamically accumulates and adjusts back-annotated granularities in order to provide error-free task scheduling. Results show that high accuracy is achieved while maintaining fastest possible simulation speed regardless of user-defined delay granularities. In addition, by eliminating preemption errors and consequently unpredicted error bounds, this makes host-compiled simulators suitable for rapid and early evaluation of the real-time performance of embedded systems.

Chapter 5

Processor Modeling

In the preceding chapter, we focused on the OS model internals and examined the timing management model proposed for adjusting the simulation granularity to achieve both fast and accurate results. Beyond the OS model, to provide feedback about timing-accurate HW/SW interactions, we develop a host-compiled processor model, which emulates both TLM bus accesses and a high-level multi-core interrupt handling chain. In addition, for more accurate real-time performance evaluation, dynamic cache effects are considered in the processor model as well. However, in the context of coarse-grained simulation, fast yet accurate modeling of multi-core cache hierarchies poses several challenges. We therefore propose a novel generic multi-core cache modeling approach that incorporates a delayed reordering technique to replicate an accurate multi-core cache behavior even under a coarse-grained simulation.

This chapter* is organized as follows: In Section 5.1 and Section 5.2, we present the overall structure of our complete host-compiled platform sim-

*Contents of this chapter previously appeared in the following papers:
P. Razaghi and A. Gerstlauer. Multi-core cache hierarchy modeling for host-compiled performance simulation. *ESLsyn*, May 2013.
P. Razaghi and A. Gerstlauer. Host-compiled multi-core system simulation for early real-time performance evaluation. *ACM TECS*, accepted for publication, March 2014.

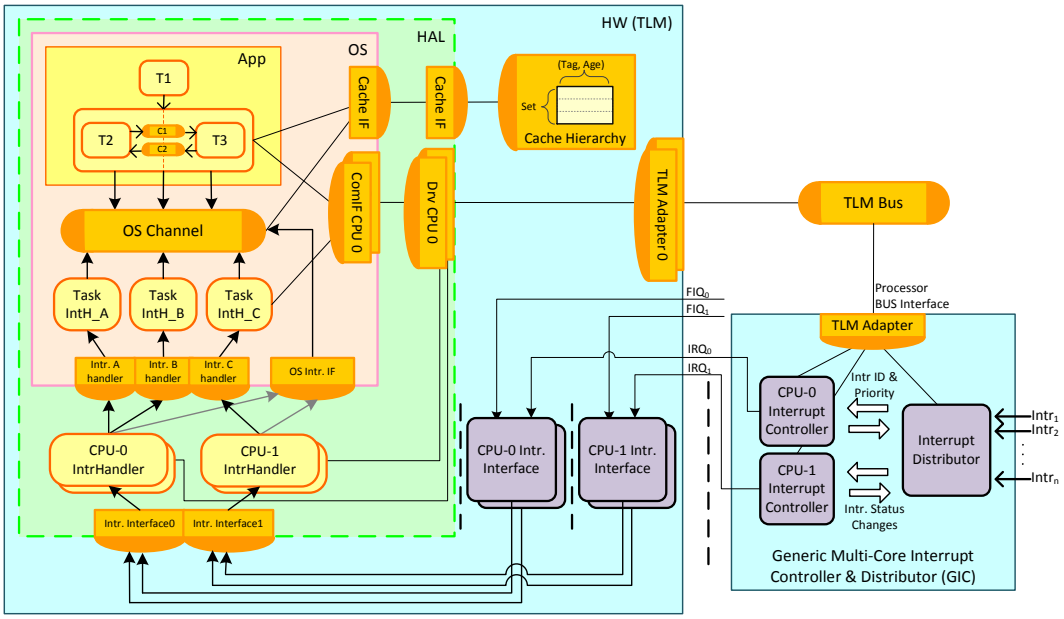


Figure 5.1: High-level multi-core processor model.

ulation backplane. In Section 5.2.3, we then show an example of a general execution sequence. In Section 5.3, we further propose a novel muti-core, out-of-order cache hierarchy modeling approach. Finally, in Section 5.4, we examine different aspects of our complete simulator using a set of experiments.

5.1 Processor Model Overview

Figure 5.1 depicts the connections across different layers of our platform models. At the innermost layer, a user application is directly connected to the OS layer and accesses the scheduling services via the provided OS API. In addition to the OS services detailed in Section 4.2, the OS layer provides

high-level communication primitives for sending and receiving inter-processor application-level messages via the HAL. Together, the OS and HAL thereby realize models of drivers that transform application-level messages all the way down to corresponding transaction-level bus accesses plus interrupt-driven or polling-based synchronizations, if required. Finally, the HW layer models external bus communication via a TLM bus channel. The HW layer also emulates monitoring of processor interrupt signals and associated processor exceptions. The HAL combined with the HW layer constitute the processor model. In this way, the simulator can be integrated into any standard TLM backplane for co-simulation in an overall multi-processor system environment.

5.2 High-Level Interrupt Handling Mechanism

In addition to the overall structure of the processor model, Figure 5.1 depicts extra hardware components, inter-layer interfaces, and specialized OS-level tasks to replicate a general multi-core interrupt handling chain in our host-compiled model. From the hardware side, core-specific interrupt requests (IRQx) are generated by a generic multi-core interrupt controller (GIC) model, which manages the distribution of interrupt signals across processor cores. The internal structure of the interrupt controller will be discussed next in Section 5.2.2.

5.2.1 Processor and OS Interrupt Model Integration

Inside the processor model, the HW layer contains core-specific SLDL processes (*IntrInterface*) that are sensitive to changes on the external interrupt inputs (IRQx). Whenever an interrupt request is asserted, the corresponding *IntrInterface* notifies the OS kernel by calling the `IEnter()` method of the OS API, exported to the HW layer via the HAL. `IEnter()` then moves the desired interrupt handler into the corresponding core's *IntrHandlerReady* queue (as shown in Figure 4.3) and triggers the OS-internal *schedulerEvent* to inform the OS of the recently activated interrupt. Depending on fallback versus predictive mode, the OS will terminate the current time-wait primitive and call the scheduler to perform a context-switch (as illustrated in Section 4.3). As discussed previously, the OS scheduler always first checks the *IntrHandlerReady* queue for active interrupt handlers in order to model processor suspension in response to external interrupt events.

In this approach, interrupt handlers are modeled as special, high priority tasks associated with each core. Interrupt handlers are created and added to the OS kernel by the HAL via the `CreateIntrHandler()` methods of the OS API. When an interrupt handler is scheduled by the OS kernel, it communicates with the GIC via a TLM bus channel to determine and acknowledge the interrupt source. It next triggers a special interrupt task associated with the interrupt source via a call to the `IntrTrigger()` method of the OS API. Before the interrupt handler returns to the OS kernel, it removes itself from *IntrHandlerReady* queue by calling the `IReturn()` method. Finally,

user-supplied code in the interrupt tasks can communicate with external hardware, with application tasks or with the OS model, e.g. to spawn additional processing tasks.

5.2.2 Multi-Core Interrupt Controller

We develop a generic multi-core interrupt controller (GIC) model that manages interrupt distribution among the processor cores and generates interrupt request (IRQ) signals associated with each core. The high-level GIC model is a configurable, generic multi-core interrupt controller that is modeled after typical real-world components, such as the ARM Generic Interrupt Controller Architecture [2]. Our model supports up to 32 edge-triggered hardware interrupts and is able to manage the interrupts for an arbitrary number of processors/cores. A user can program the interrupt controller to define interrupt priority and target core for each interrupt source. The GIC model replicates a 1-N model for handling interrupts, i.e. it ensures that only one processor handles a captured interrupt.

Internally, our high-level GIC model is composed out of one centralized interrupt distributor and per-processor CPU interfaces (as shown in Figure 5.1). The distributor monitors incoming interrupts and dispatches the highest priority asserted interrupt to the associated CPU interface, which is determined by the programmed target core list. The CPU interface thereby asserts the IRQ signal to its connected core and takes care of the communication between the processor and GIC.

To manage interrupt detection and distribution, the GIC model identifies each interrupt by an ID and maintains the interrupts' state transitions. When an interrupt is asserted by a connected hardware, the GIC moves that interrupt to the *Pending* state. The distributor can then detect all pending interrupts and send the highest priority one to the corresponding CPU interface. A pending interrupt moves to the *Active* state whenever it is handled by the corresponding core, i.e. the associated interrupt handler reads the "Acknowledge" register. An active interrupt can move to *Inactive* (initial state) or *Active & Pending* states when the interrupt handler writes to the "End of Interrupt" register or another interrupt signal with the same ID is captured, respectively.

In such a detailed GIC model, three context switches are required in the simulator for an external interrupt to propagate until the actual interrupt handler is executed. Such a detailed model therefore carries a large simulation overhead. We develop an alternative, lightweight GIC model that reduces the overhead by a factor of two. This model only contains core-specific processes, which identify the highest priority pending interrupt for the connected processor and communicate with the processors and the associated interrupt handlers directly. To achieve a faster interrupt routing and consequently a faster simulation, the priority of an interrupt is only defined by its ID and is not programmable. Furthermore, this model does not replicate a 1-N implementation. Hence, designers need to be careful when programming the GIC to ensure that each input interrupt is only mapped to a single core.

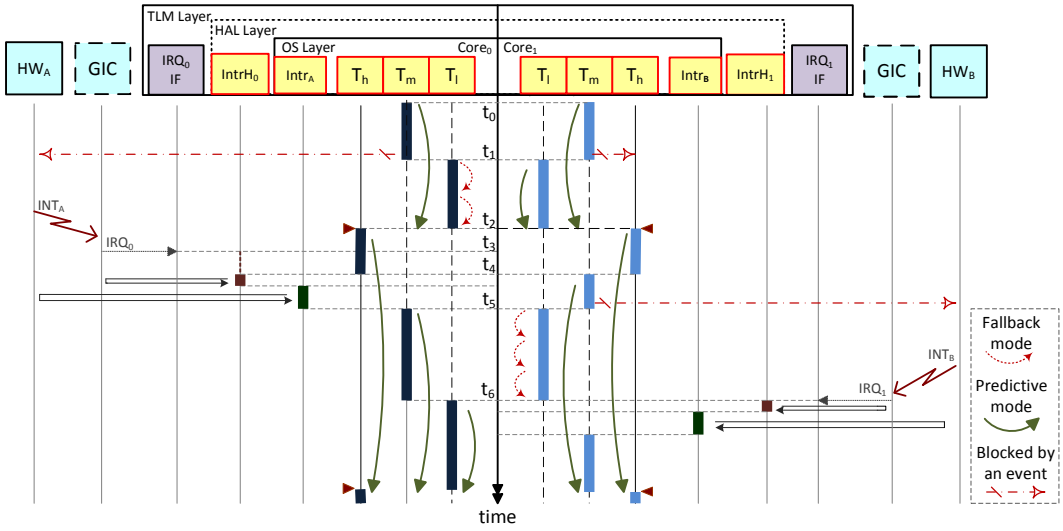


Figure 5.2: Host-compiled simulation trace.

On the whole, the presented processor and interrupt controller models and inter-connections provide an infrastructure that enables accurate modeling of interactions between hardware components and multi-core processors.

5.2.3 Host-Compiled Simulation Trace Example

To illustrate the execution sequence of the integrated host-compiled model, we show a simulation trace of two task sets running on a dual-core platform under a partitioned scheduling strategy (see Figure 5.2). Each set contains three tasks with high, medium and low priorities (named T_h, T_m, and T_l respectively). Sets are mapped to run on separate cores. The highest priority task is modeled as a periodic task. Tasks may communicate with each other or external hardware via interrupt signals.

At the beginning of the simulation, T_h is in *Idle* state and T_m is therefore

scheduled on both cores. At time t_1, T_m running on $core_0$ is blocked on an external event, and T_l is scheduled on that core. From this point forward, the OS switches to fallback mode on $core_0$, since a higher priority task is waiting for an interrupt. At the same time, T_m on $core_1$ waits for a response from T_h, and T_l is scheduled accordingly. Both cores can be in predictive and fallback modes independently, and the OS remains in predictive mode for $core_1$, since it is aware that neither T_m nor T_h can be scheduled before the next release time of the periodic task T_h (time t_2).

At time t_3, the hardware sends an INT_A interrupt to the GIC, which is programmed to route that interrupt to $core_0$. Accordingly, $core_0$'s interrupt interface activates the corresponding interrupt handler. However, we can delay the execution of the interrupt handler until the finish time of T_h. Such a situation allows the OS to stay in predictive mode, where it will not call the scheduler until time t_4, the finish time of T_h. The interrupt handler then communicates with the GIC and activates the corresponding interrupt task for further interactions with the hardware. Although such an interrupt modeling approach introduces a small timing error in the execution of T_h, the interrupt task and the interrupt handler, the simulation is kept fast while these errors are typically negligible. Finally, the interrupt task $Intr_A$ releases T_m, such that T_m and subsequently T_l execute in predictive mode until the start of the next period of T_h.

Now consider the simulation trace on $core_1$: after releasing T_m again once T_h has run, T_m blocks on an external interrupt at time t_5 while T_h is

Idle. Task T_l is therefore scheduled and the OS switches to fallback mode in order to continuously monitor for possible interrupts. As such, when the interrupt request is captured by $core_1$ at time t_6, the OS schedules the activated interrupt handler immediately and provides a fully accurate interrupt handling sequence.

5.3 Multi-Core, Out-of-Order Cache Modeling

In this section, we propose a novel high-level, multi-core cache hierarchy modeling approach, which accurately models cache behavior of multi-core processors, simulated by a host-compiled simulator. In the following, we first present an overview of our generic cache model and its integration process into the host-compiled simulator. Next, we demonstrate a novel memory access reordering technique, which is designed for fast yet accurate cache behavior simulation in a temporally decoupled execution context.

5.3.1 Base Approach

For accurate performance evaluation, we need to consider performance penalties due to cache misses and update static back-annotated delays during simulation. For this purpose, we develop a high-level model of a cache channel that emulates the system memory behavior by updating its internal states on every memory access. Note that we only need to model hit/miss behavior of the cache, i.e. we are not concerned about the data that is stored in the cache. Instead, the simulation host takes care of maintaining coherent data values.

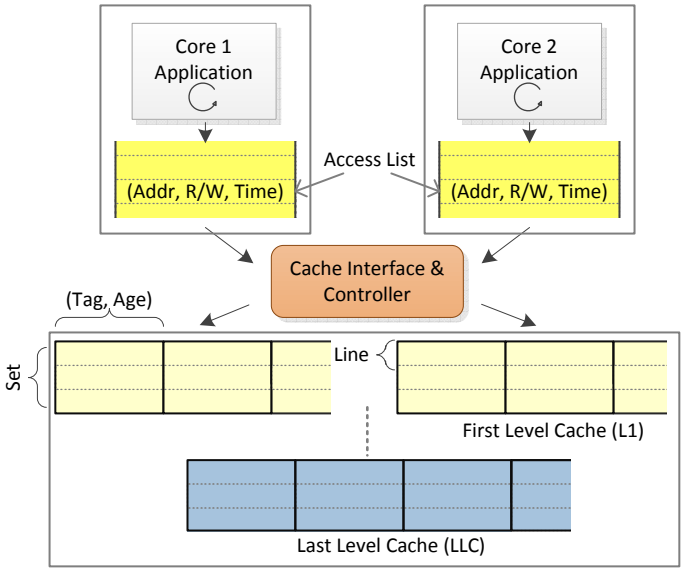


Figure 5.3: High-level cache hierarchy model.

In our cache model, each cache line is composed out of an address tag, an age counter to implement a replacement policy, and a coherency flag to store the current state of each line compared to other cores' caches (Figure 5.3). Associated with each core, an access list stores locally ordered memory references reported by the application running on that core. Each location in this list contains a memory address, access mode (i.e Read or Write), and an access time. Using this information, a cache controller is able to manage the cache state updating process and to report back total miss cycles. Accordingly, the simulator can internally adjust back-annotated task delays by adding delay for extra memory cycles.

As mentioned before, a user application is integrated into the simulator at the source level. As such, low-level information including task execution

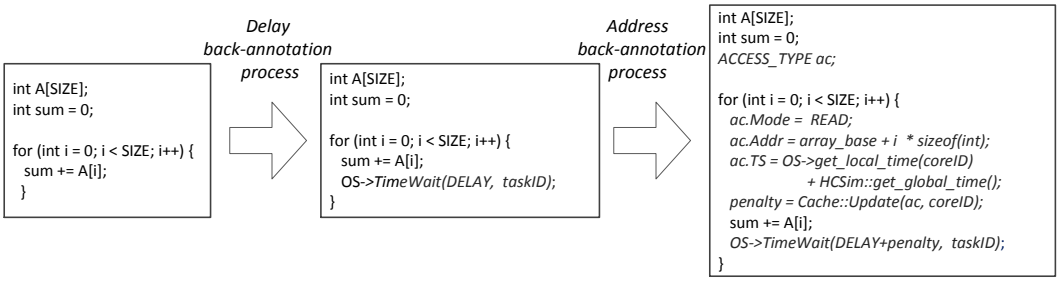


Figure 5.4: Source code back-annotation example for cache simulation.

delays and memory references need to be back-annotated into the source code. In this dissertation, we assume that designers use existing techniques to obtain task execution delays and accessed memory addresses based on a selected target platform [70, 91, 10]. Figure 5.4 shows source code instrumentation steps to consider cache effects during host-compiled simulation. The original application is a simple loop over an integer array. The execution delay is given to the simulator by calling the `TimeWait()` method of the OS API. In this way, the OS kernel internally advances simulated time to model task and core execution times.

In the address back-annotation step, every memory access is reported to the simulator by simply committing the access information including a 3-tuple of (address, mode, time stamp) into the cache. The time stamp is an absolute time, which is determined as the sum of the global simulated time and the core's local time. Note that absolute time stamps are required for temporal decoupling simulation and are used in our reordering technique (see Section 5.3.2). At the end, the cache returns any miss penalties incurred during the access, which are in turn added to the task's execution delay.

Table 5.1: Cache configurable parameters.

Parameters	Description
Levels of hierarchy	Core-private L1 Core-private L2, or shared L2 within a package Shared L3 within a package
Cache structure	Cache size, Line size Associativity Write back, Write through (write allocation)
Replacement policy	LRU: default policy
Miss latency	Number of cycles for miss penalty of each level

We design our cache model such that a designer can easily explore a wide variety of cache architectures and evaluate their effects on system performance. Table 5.1 lists all configurable options that our cache model offers. Primarily, the number of packages and cores per package need to be defined to generate the overall structure of the cache hierarchy. In addition, the number of levels of the cache hierarchy and interconnections across these levels are configurable. For example, the L2 can be defined a shared cache between all cores on a package or as a core-private cache. Furthermore, the structure of each cache including total and set size, replacement and write policies are parametrizable. By configuring the miss latency of all levels, the cache channel can provide the total miss penalty on every cache update. Finally, to keep the cache hierarchy coherent, a standard MSI-based cache coherency protocol is implemented.

5.3.2 Decoupling and Cache Simulation

The ATGA approach presented in Section 4.3 accumulated tasks execution delays locally and only advances the global simulated time whenever a local task switches or a global synchronization is required. In such an approach, cores are temporally decoupled until a synchronization point is reached. In general, temporal decoupling is a well-known mechanism to improve the simulation speed by increasing the granularity of simulation. In such an approach, a thread can go ahead of the simulation kernel time without advancing global time. However, decoupling techniques may decrease the accuracy of cache behavior due to the coarse-grained synchronization of parallel threads. In multi-core cache simulation, different cores may commit their memory accesses to the cache globally out-of-order. In the following, we propose a delayed reordering technique that provides an accurate temporally decoupled cache hierarchy simulation.

To illustrate the general concept behind the reordering technique, we show the execution sequence of a dual-core platform in Figure 5.5. At the beginning of the simulation, $core_1$ starts the execution of its application code and reports three memory accesses at local times $1(ts_1)$, $2(ts_2)$, and $12(ts_3)$. At local time 12, $core_1$ notified the simulator to consume the accumulated delay. Accordingly, the host-compiled simulator internally advances the simulated time by calling the underlying SLDL *wait(12)* method. As such, $core_1$ is suspended and $core_2$ gets a chance to run its application code, which reports its two memory accesses at local times $10(ts_4)$ and $20(ts_5)$. As can be seen, ts_4

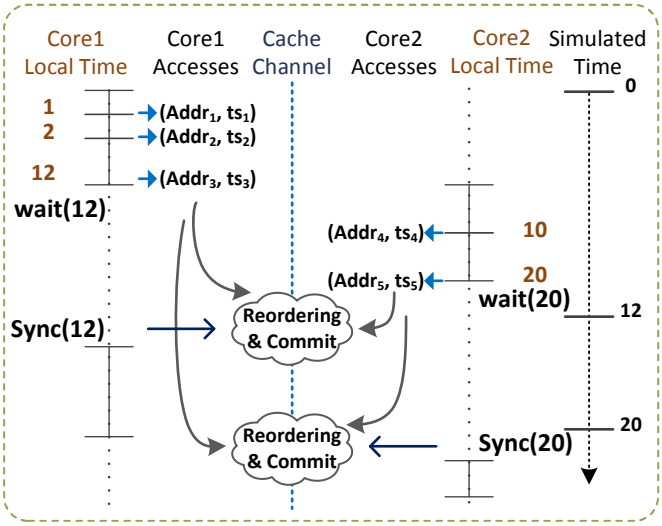


Figure 5.5: Cache reordering example.

is reported after ts_3, while to be committed before ts_3. When $core_2$ request to consume the accumulated delay in a similar way, the simulator internally advances the global simulated time by 12 units and returns to $core_1$. By this time, all cores have already collected their memory accesses. Hence, $core_1$ calls a *Sync(12)* method to commit the out-of-ordered accesses with their correct sequence into the cache channel.

To enable delayed reordering, instead of directly committing accesses to the cache as they occur, each core keeps all referred memory addresses in an ordered list. Although each access has a time stamp, which allows the cache to detect the global sequence of all accesses, the simulator needs to invoke the cache synchronization and reordering method when all cores' accesses have been collected and it can be determined that they are safe to commit.

```
Function: TimeWait(time_t nsec)
1    cur_core.local_time += nsec
2    sync_time = simulation_quantum
3    while cur_core.local_time ≥ sync_time do
4      SLDL::wait(sync_time)
5      cur_core.local_time -= sync_time
6      cache_time = Cache::Sync(sync_time, cur_core)
7      cur_core.local_time += cache_time
8    endwhile
```

Figure 5.6: Timing model for out-of-order cache modeling.

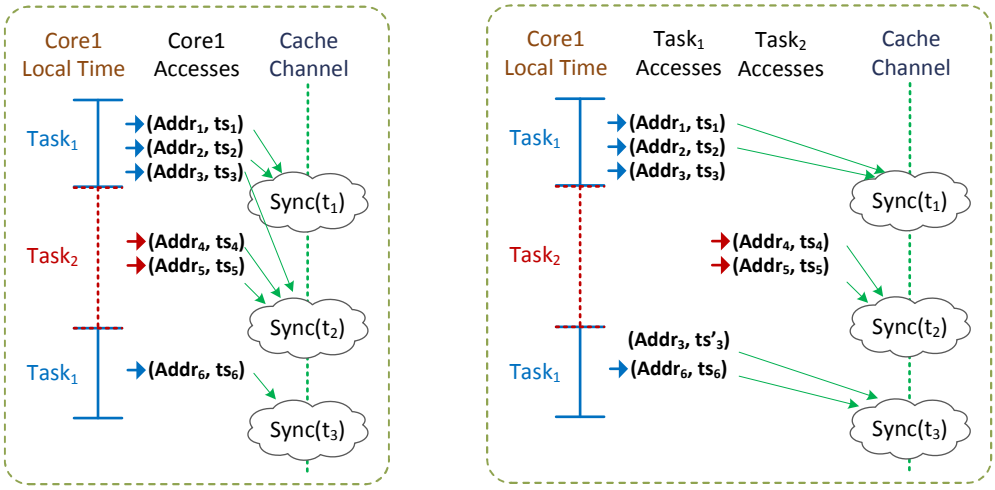
An efficient safe point for committing collected memory accesses is after advancing the simulation time. In this way, the underlying SLDL simulation kernel lets other cores run their tasks and collect all memory accesses up to that point in simulation time. Figure 5.6 shows the function that manages the simulated time. Task delays are back-annotated into the code via a call to this `TimeWait()` method. The current core first updates its local time. If the accumulated delay is greater than the simulation quantum, the global simulated time is advanced by the underlying SLDL *wait* primitive (line 4). After advancing the simulation time, the `Sync()` function in the cache is called (line 6). Finally, the local time is updated by possible extra delays caused by cache miss penalties of committed accesses (line 7).

```
Function: Sync(time_t sync_time, int cur_core)
1    while true
2      for all cores do
3        min_core = ArgMin_i(access_list[i].first.ts)
4        if (access_list[min_core].first.ts > sync_time) break
5        a = access_list[min_core].pop()
6        min_core.local_delay += Cache::Update(a, min_core)
7      endfor
8    endwhile
9    return cur_core.local_delay
```

Figure 5.7: Cache synchronization and reordering algorithm.

5.3.3 Reordering mechanism

The pseudo code of our cache synchronization and reordering technique is shown in Figure 5.7. The reordering algorithm is divided into three steps: In the first step, the core containing the access with the smallest time to commit is determined by exploring all core access lists (line 3). In the second step, any corresponding memory access with a time stamp smaller than or equal to the safe to commit time (synchronization point) is used to update the cache state (line 4 and 5). Finally, based on the cache behavior, the core's local delay is updated to record extra delays related to miss penalties (line 6). This loop continues until all accesses from all cores with a time stamp smaller than the end time are committed to the cache model.



(a) Preemption error example. (b) Preemptive reordering model.

Figure 5.8: Cache preemptive reordering example.

5.3.4 Preemptive Reordering mechanism

The presented reordering method works based on the assumption that only one task is running on each core. Figure 5.8 (a) shows the situation in which memory accesses are still committed out-of-order due to a task preemption. As the execution trace shows, $Task_1$ issues three memory accesses, but only two accesses are committed to the cache by the `sync()` function. The third one is then left in the list and when $Task_2$ preempts the current task, the `sync()` function commits a memory access from previously executed task. Figure 5.8 (b) shows a preemptive reordering model, which associates an access list per application task instead of per core. However, the time stamp of remaining items still need to be updated by the amount of time that the associated task was preempted.

To efficiently managing time stamps, each access list locally stores delays related to cache penalties and preempted times. In this order, the `sync()` function calculates the absolute time stamp using the following formula:

$$absolute\ time\ stamp = \begin{cases} Accumulated\ Cache\ Penalties & \\ & + \\ Preempted\ Time & \\ & + \\ Task\ Executed\ Delays & \end{cases}$$

Where *accumulated cache penalties* is the count of all cache delays correlated to miss penalties, *preempted time* is sum of all time task waiting for scheduling, and *task executed delays* is sum of task's all execution times, i.e *"wait for time"* times plus the accumulated delay.

In summary, with incorporating the delayed reordering of aggregated requests, we are able to provide the highest possible accuracy, while increased simulation performance benefits from temporal decoupling introduced by our ATGA timing model.

5.4 Experiments and Results

We evaluate different aspects of our simulator using a set of successive experiments. First, we evaluate the accuracy of our high-level interrupt handling models and the accuracy and the simulation performance of our OS and processor models on a suite of artificial task sets and compute-intensive multi-threaded benchmarks. We then also explore different architectures of an industrial-strength example to show the benefits of our simulation platform

for early design space exploration. At the end, we examine the accuracy and speed of our reordering cache modeling approach. All host-compiled simulations are performed on a 2.67 GHz Intel Core i7 workstation using the SpecC version of our simulator.

5.4.1 Experimental Setup

To evaluate our processor model, we use the Open Virtual Platform (OVP) [64] as a reference ISS for evaluating the accuracy. OVP consists of an instruction-accurate simulator (OVPSim), fast processor models, and behavioral peripheral modeling and programming APIs, which enable full-system modeling and simulation. Furthermore, we use Imperas, Verification, Analysis and Profiling (M*VAP) tools to measure the execution cycles of applications in kernel and user space [40]. Since OVP is based on a timing model of one cycle per instruction (CPI=1), execution delays are determined as the product of the number of executed instructions and the target processor clock period.

The reference platform consists a quad-core Cortex-A9 ARM processor running a Linux 2.6.39 SMP kernel at a frequency of 1 GHz. OVP is configured to run at instruction-level granularity. The OVP peripheral modeling library is used to implement extra hardware components. Correspondingly, loadable kernel modules are developed to integrate drivers for the hardware into the platform and application. Hardware accesses are implemented using interrupts and memory-mapped I/O.

For timing-accurate interrupt modeling, we measure the number of in-

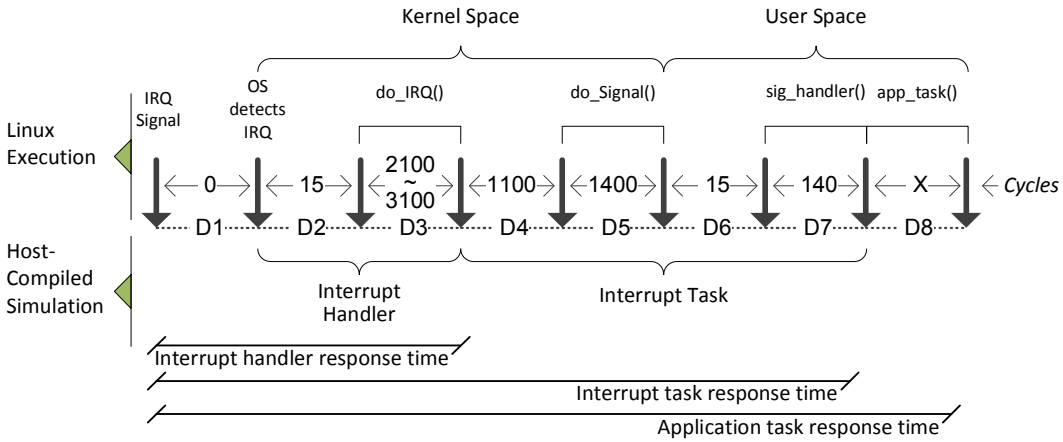


Figure 5.9: Modeling of Linux interrupt handling.

structions executed by Linux when handling an interrupt. Figure 5.9 shows the sequence of interrupt events in Linux and their mapping to the host-compiled simulation model. Since we modeled the system bus and GIC as untimed, high-level peripherals, the Linux kernel starts handling interrupts immediately after an interrupt is signaled. In our setup, the interrupt handler notifies an application process using Linux real-time signals. The execution trace shows that the interrupt handler delay varies between 2,100 to 3,100 cycles[†], depending on the system state and the internal implementation of signal queues in the Linux kernel. The Linux kernel then delivers the notified signal to the corresponding process by evoking an associated signal handler (`do_Signal()` in kernel space and `sig_Handler()` in user space). Note that in Linux, these signal handlers are executed in the context, i.e. at the same priority and time

[†]Since OVP considers CPI=1, the number of cycles and instructions are interchangeable for the experiments presented in this section.

as their associated processes. Finally, the signal handler determines the interrupt source and calls a corresponding application service routine, which in turn communicates with the main process by releasing a POSIX semaphore to signal that the interrupt has been received.

To accurately model the Linux interrupt handling chain, we map the functionality of interrupt and signal handlers to interrupt handlers and interrupt tasks in our host-compiled model. Matching Linux behavior, interrupt tasks are thereby assigned the same priority as their associated user tasks. Finally, we back-annotate interrupt handler and task models with corresponding delays ($D2 + D3$ and $\sum(D4..D7)$, respectively).

5.4.2 Interrupt Handling Evaluation

To verify the accuracy of the interrupt handling model, we compare the response time of interrupt handlers, interrupt tasks, and corresponding interrupt-driven application tasks to the reference simulation. Response times are defined as the delay between signaling an interrupt and completion of the corresponding handler or task (see Figure 5.9). Table 5.2 lists average absolute errors in response times for two different experiments, in which six interrupt-driven tasks are running on a single-core ARM processor. In experiment E1, all interrupts are triggered simultaneously with a fixed period of 10 ms. By contrast, in experiment E2, interrupts are generated at different rates in order to stress the experiment under random conditions. The interrupt controller was programed to assign different priorities to each interrupt signal. Experiments

Table 5.2: Interrupt handling response time errors.

Interrupts	**E1: Identical Interrupt Load**				**E2: Random Interrupt Load**			
ID/Priority	Period	Intr. H	Intr. T	App. T	Period	Intr. H	Intr. T	App. T
$1/low$	$10ms$	0.86%	0.01%	0.01%	$15ms$	2.84%	0.27%	0.16%
2	$10ms$	0.20%	0.03%	0.02%	$12ms$	2.17%	4.97%	0.68%
3	$10ms$	0.39%	0.04%	0.03%	$11ms$	1.32%	5.36%	0.52%
4	$10ms$	0.84%	0.05%	0.03%	$8ms$	1.01%	9.12%	0.93%
5	$10ms$	1.03%	0.02%	0.01%	$7ms$	0.69%	7.01%	0.62%
$6/high$	$10ms$	0.23%	0.50%	0.10%	$5ms$	0.17%	4.01%	0.29%
Avg. Error		0.60%	0.11%	0.03%		1.37%	5.12%	0.53%

were run for a total simulated time of 5 sec.

Results show that for randomized interrupt behavior, average response time errors can be as high as 10% with maximum errors reaching 50% in some instances. Upon closer inspection, these errors stem from interference of the high-priority timer interrupt not being modeled in our setup. By synchronizing all interrupts with the fixed 10 ms rate of the Linux timer, such interferences are eliminated and both average and worst-case errors drop below 1%. Overall, assuming timer and other interrupt handlers to be generally short, results prove the accuracy of the interrupt controller and interrupt handling models.

5.4.3 Processor Evaluation

To evaluate overall simulation accuracy and performance, we used random periodic task sets as generated in Section 3.2. We run each task set for 5 sec of simulated time. At a nominal rate of 1000 MIPS simulated by the reference ISS, this corresponds to 5 billion NOP instructions running on each core. Experiments are executed on a simulated dual-core and a quad-core platform.

Table 5.3: Artificial task set simulation results.

Set	S1	S2	S3	S4	M1	M2	M3	M4	L1	L2	L3	L4
Number of Tasks	13	18	29	58	8	8	8	16	7	5	6	12
Number of Cores	2	2	2	4	2	2	2	4	2	2	2	4
Avg. Task Weight	.05	.06	.06	.06	.14	.16	.19	.19	.19	.26	.28	.28
CPU Utilization	.65	1.03	1.7	3.4	1.1	1.3	1.5	3.0	1.3	1.3	1.7	3.4
Intr. High												
Error (periodic)	.33%	.26%	.31%	.33%	.11%	.05%	.02%	.04%	.02%	.02%	.04%	.13%
Error (intr-driven)	.37%	.23%	.21%	.23%	1.02%	.49%	.34%	.74%	.76%	.60%	.50%	.91%
Speed [GIPS]	24	40	60	53	46	54	63	53	47	46	70	62
Intr. Medium												
Error (periodic)	.15%	.11%	.13%	.13%	.11%	.05%	.01%	.01%	.06%	.02%	.04%	.04%
Error (intr-driven)	.13%	.06%	.16%	.19%	.12%	.12%	.10%	.10%	.13%	.19%	.10%	.11%
Speed [GIPS]	80	128	213	170	140	165	188	251	165	215	211	281
Intr. Low												
Error (periodic)	.15%	.11%	.13%	.13%	.07%	.05%	.01%	.01%	.02%	.02%	.04%	.05%
Error (intr-driven)	.23%	.17%	.25%	.27%	.13%	.11%	.13%	.13%	.08%	.07%	.03%	.05%
Speed [GIPS]	161	171	284	284	278	660	377	377	330	322	422	422
No. Intr.												
Error (periodic)	.15%	.11%	.13%	.13%	.07%	.05%	.01%	.01%	.03%	.02%	.04%	.04%
Speed [GIPS]	322	513	426	426	696	824	1,884	1,500	1,100	1,287	1,406	1,688

Each task set is executed under four different interrupt conditions: periodic tasks plus a high, a medium, or a low priority interrupt-driven application task running on each core, or only periodic tasks running on the cores. Task weights for interrupt-driven application tasks are fixed at a value of 0.01, and their generation frequency/load is proportional to their priority, i.e. a higher load for the interrupt with a higher priority. For accuracy measurement, task response times are compared to the reference ISS.

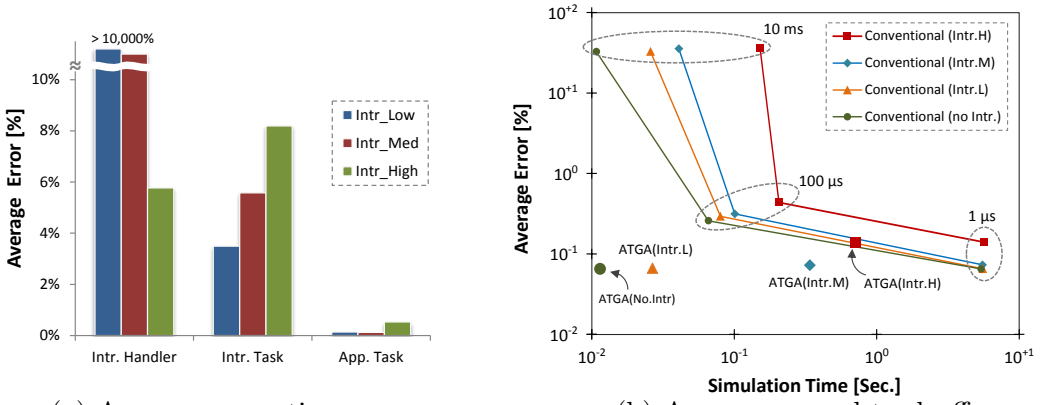
Table 5.3 lists the task set features and summarizes accuracy and speed results. Error is measured as the average percentage of absolute differences in individual task response times over all tasks and task iterations. To measure simulation performance, only the number of actually simulated instructions is considered. The number of simulated instructions is calculated based on

the total simulated time and the nominal NOP instructions executed on the reference ISS. The idle time is eliminated by considering the CPU utilization. In other words,

$$Speed = \frac{Simulated\ time * 1000\, MIPS * CPU\ utilization}{Simulation\ time}.$$

Results show an average timing error of 0.16% and an average speed of 400 GIPS over all task sets and experiments. In all cases, error variance remains below ±1.8%. Although we would expect to see zero errors using our ATGA approach, remaining errors are caused by non-modeled OS context-switch overheads, non-ideal behavior of a real Linux system (as studied in Section 3.2)), and errors in measured back-annotated delays. We can observe constant average errors for the periodic task sets in all experiments except when a high priority interrupt is running in the system. Reduced accuracy in these cases is caused by the error in estimated delays of interrupt and signal handlers, which have a larger effect when the interrupt load is high and tasks delays are small.

Further investigation of errors in the interrupt chain are shown in Figure 5.10 (a). High errors are measured for interrupt handlers in experiments with low and medium interrupt priorities. This is due to deliberate inaccuracies in the interrupt model. Since the OS model does not switch to fallback mode when the priority of the running task is higher than any interrupt-driven task, start times of interrupt handlers can be delayed until the next predicted scheduling point. This results in high response time errors of interrupt handlers. However, the total effect on response time of application-level tasks is



(a) Avg. response time errors. (b) Accuracy-speed tradeoffs.

Figure 5.10: Accuracy and speed analysis for artificial task sets.

negligible. Furthermore, larger errors are observed for high-priority interrupt and application tasks. Since high priority interrupts are generated at higher rates, non-modeled Linux back-ground tasks can introduce larger timing errors in such cases.

Finally, we compare our host-compiled simulator using ATGA approach against a conventional host-compiled simulation at user-defined simulation granularity. Figure 5.10 (b) compares the average simulation error and simulation time of ATGA and conventional models under different timing granularities. As can be seen, there is a fundamental accuracy and speed tradeoff in a conventional simulation, i.e. decreasing the timing granularity results in a higher accuracy but comes at a loss in simulation performance. Furthermore, there is a high variation in errors under large granularities. For example, with 10 ms simulation granularity, errors across task iterations vary between 0.1% and 1,200%. By contrast, our simulator automatically provides fast simula-

Table 5.4: ParMiBench accuracy and speed results.

Application	Simulated time (Error)			Simulation speed [MIPS]		
	CPU=1	CPU=2	CPU=4	CPU=1	CPU=2	CPU=4
Bitcount (112500 iter.)	582ms (0.58%)	291ms (0.80%)	147ms (1.13%)	3,600	2,600	2,600
Basicmath (500K num.)	287ms (0.16%)	144ms (0.34%)	72ms (1.88%)	2,300	2,600	2,250
Susan-edge (2.8MB pic.)	9.134s (0.36%)	5.016s (0.64%)	2.928s (0.62%)	3,950	3,800	3,800
Susan-corner (2.8MB pic.)	1.80s (0.32%)	0.959s (1.48%)	0.532s (0.18%)	4,800	4,800	4,800
Susan-smooth (2.8MB pic.)	11.60s (0.11%)	5.843s (0.21%)	2.946s (0.73%)	2,700	2,500	2,600
SHA (16 input files)	348ms (0.15%)	218ms (0.19%)	156ms (0.27%)	2,800	3,200	3,200
Dijkstra (160 nodes)	45.58s (0.02%)	22.79s (0.01%)	11.39s (0.06%)	6,500	6,900	6,900
Patricia (5000 IP address)	917ms (0.73%)	459ms (0.43%)	229ms (0.42%)	2,200	2,100	2,100
Stringsearch (16MB in file)	59.11s (0.20%)	29.55s (0.35%)	14.77s (0.79%)	2,900	2,900	2,900

tion with the highest possible accuracy. The ATGA simulation provides the fastest possible speed when no interrupt is running in the system and the OS kernel runs only in predictive mode. By contrast, task sets with high-priority interrupt-driven tasks require the OS model to remain in fallback mode and thus simulate much slower.

5.4.4 Inter-Task Communication Evaluation

Finally, to evaluate our SMP OS model under multi-core conditions, we apply our simulator to a set of compute-intensive multi-threaded applications from the ParMiBench suite [42]. ParMiBench applications are parallelized by data decomposition across threads that are synchronized via barrier channels. Task delays are back-annotated at the function level from measurements taken on the ISS. Pthread calls are mapped into corresponding OS model primitives, where a high-level inter-task channel is implemented to model POSIX-based barrier synchronization. Table 5.4 summarizes simulation accuracy and speed for a single, dual and quad core simulated platform. Results show that sim-

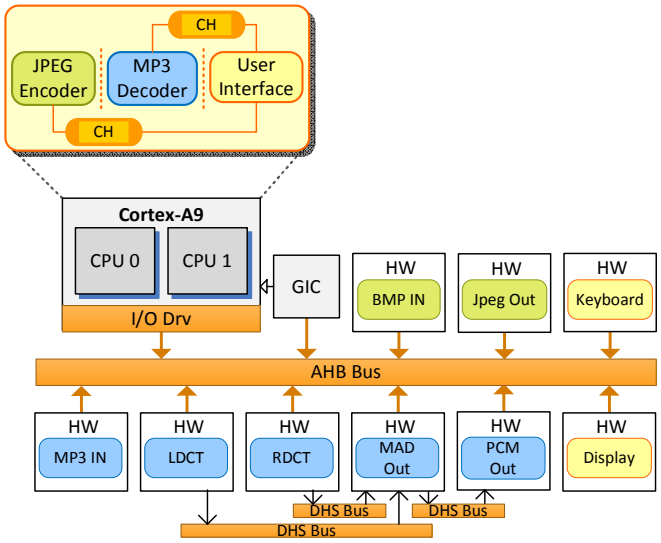


Figure 5.11: Motion-JPEG example architecture.

ulated application runtimes follow the reference simulation with an average error of 0.5%. Remaining errors are largely due to non-modeled execution delays in Pthread calls. Overall, when replacing artificial NOP instructions with real code, average simulation speed is degraded to 3,500 MIPS.

5.4.5 System Evaluation for Design Space Exploration

To demonstrate the benefits of our simulator for fast and accurate design space exploration, we apply our models to an industrial-strength Motion-JPEG (M-JPEG) example, running three concurrent MP3, JPEG and user interface tasks on a dual-core 650 MHz Cortex-A9 platform model. The overall architecture of the system is shown in Figure 5.11. The MP3 decoder uses hardware accelerators to perform audio decoding, and the JPEG encoder is completely implemented in software. Tasks communicate with external hard-

ware and the rest of the system via an AHB bus and 12 interrupts. MP3 decodes 13 frames at a bitrate of 384 kbit/s, and JPEG encodes 10 frames of a 30 frames/s movie with 352×288 resolution.

For accuracy analysis, we compare the execution behavior of MP3 and JPEG tasks simulated on our host-compiled models to the reference OVP ISS. Task delays are back-annotated at the function level from measurements taken on the ISS. Moreover, average Linux context-switch overhead is measured and back-annotated into the OS model. We explore a wide range of architectures by applying different OS and processor configurations, including mapping of M-JPEG tasks and interrupts to different cores. Error is measured as the average percentage of absolute differences in individual frame delays over all frames. The simulation speed is calculated based on the number of application and Linux kernel instructions simulated by the reference ISS excluding Linux boot-up times. Instruction counts number between 800 and 1,040 million instructions depending on the system configuration. Note that the application-only instruction count is 160 million instructions, which means that a significant performance benefit comes from the OS abstraction approach. Finally, in order to achieve fast simulation, a TLM of the AHB bus at a granularity of user/application transactions [79] and the lightweight model of the GIC are used.

Table 5.5 shows the average error and the simulation speed for the explored architectures simulated by the ATGA model and the conventional model under three different granularities. In dual-core architectures with a

Table 5.5: Motion-JPEG example simulation results.

Configuration	Average Frame Error				Simulation Speed [MIPS]			
	ATGA	Conventional			ATGA	Conventional		
		0.01μs	1μs	100μs		0.01μs	1μs	100μs
Single-Core								
1: C0: *Prty*(MP3>JPG>CTL)	0.53%	0.68%	0.66%	1.95%	1,000	50	1,170	1,500
2: C0: *FIFO*(MP3, JPG)>CTL	0.75%	0.62%	0.62%	0.62%	1,460	70	1,600	2,130
3: C0: *Prty*(JPG>MP3>CTL)	0.58%	0.65%	0.65%	0.65%	1,920	70	1,620	2,200
Task-attached interrupt model								
4: C0: CTL, C1: *Prty*(MP3>JPG)	0.48%	0.64%	0.60%	1.85%	1,000	50	1,200	1,400
5: C0: CTL, C1: *FIFO*(MP3, JPG)	0.67%	0.71%	0.71%	0.71%	1,740	70	1,500	2,200
6: C0: CTL, C1: *Prty*(JPG>MP3)	0.63%	0.64%	0.72%	0.72%	1,820	70	1,600	2,000
7: C0: *Prty*(MP3>CTL), C1: JPG	0.59%	0.72%	0.72%	0.72%	1,500	50	1,200	1,500
Core-attached interrupt model								
8: C0: *Prty*(MP3>JPG>CTL), C1: Intr	0.85%	0.93%	0.89%	3.75%	1,050	50	1,100	1,600
9: C0: *FIFO*(MP3, JPG)>CTL, C1: Intr	1.56%	1.92%	1.92%	33.1%	1,500	50	1,460	1,800
10: C0: *Prty*(JPG>MP3>CTL), C1: Intr	0.93%	0.35%	0.47%	35.9%	1,570	50	1,400	1,800
11: C0: *Prty*(MP3>CTL), C1: JPG,Intr	0.28%	0.46%	0.46%	11.9%	840	50	1,250	1,500
Average	0.71%	0.76%	0.77%	8.35%	1,400	57	1,373	1,785

task-attached interrupt model, application tasks are distributed among two cores and a task and its associated interrupts are mapped to the same core. By contrast, dual-core architectures with a *core-attached* interrupt model always handle and run all interrupts on core_1. Results show that with accurate interrupt modeling, the average error of MP3 and JPEG frame delays over all configurations is 0.71% at an average simulation speed of 1,400 MIPS. This translates into an average speed of 244 million application-only instructions per second. For configurations in which MP3 has a higher priority than JPEG, increasing the simulation granularity in a conventional approach degrades accuracy but increases simulation performance. By contrast, with the ATGA approach both fast and accurate results are achieved. This tradeoff is not

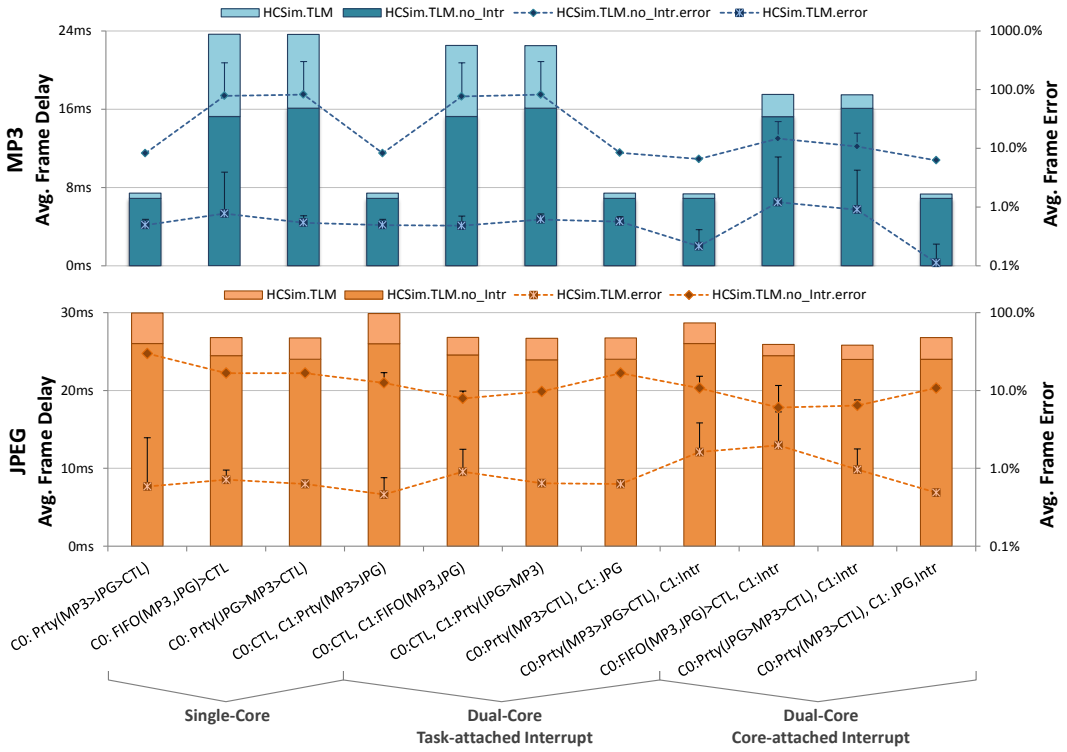


Figure 5.12: Design space exploration results.

observed in some of the other configurations. A further investigation shows that in these configurations, the OS rarely switches to fallback mode and the running task is never preempted by the next release time of a higher priority one. As such, even with the conventional model, accurate results are always achieved, i.e. the accuracy is independent of the simulation granularity. However, this is hard to predict. By contrast, the ATGA approach always provides guaranteed accurate yet fast results. Note that in some cases, the conventional simulation accuracy is higher, because errors caused by the simulation granularity compensate inaccuracies for in back-annotated delays.

Figure 5.12 summarizes the average frame delays and average frame delay errors plus max. error bars of MP3 and JPEG tasks. In order to demonstrate the importance of accurate interrupt modeling, frame delays and errors are reported both with and without modeling the interrupt handling chain (in the latter case, by bypassing the complete interrupt handling model). As can be seen, the best MP3 performance is achieved when a higher priority is assigned to MP3, or MP3 and JPEG are running on separate cores. In other configurations, average MP3 frame delay is close to its deadline boundary (i.e. 26.1 ms). Minimized JPEG delay is obtained from configurations with FIFO scheduling, when JPEG has higher priority or when it runs on a separate core. In FIFO scheduling, MP3 behaves like a low-priority task. The reason is that MP3 is often blocked waiting on hardware, while JPEG completely runs in software. As such, whenever the MP3 task is blocked, the JPEG gets the highest priority and MP3 can only resume its execution after JPEG finishes encoding of the current frame. All combined, our explorations confirm that shortest-job-first or rate-monotonic scheduling guarantee that MP3 and JPEG meet their performance requirements.

Overall, optimized MP3 and JPEG performance is achieved when tasks run on separate cores. The performance degradation of JPEG in the last configuration compared to the same *task-attached* execution is caused by extra time periods that JPEG is preempted by MP3 interrupts. Finally, by mapping all interrupts to a separate core (core_1), we only see slight performance benefits in MP3 and JPEG delays. Since interrupt handlers are not running in parallel

with their application tasks, putting them on a separate core does not reduce interrupt delays within a task but can minimize the influence of one task's interrupts on the other. This effect is more pronounced for the JPEG task, but interrupt handlers have small execution delays and do not provide large speedup benefit.

In the end, results also confirm that interrupts can have a significant influence on overall system performance. When bypassing the interrupt handling model, some configurations exhibit a very large error that is caused by a wrong execution order of MP3 and JPEG tasks. By contrast, when including the model of the interrupt chain, average errors remain within 2%. Overall, the high accuracy and fast simulation of our host-compiled simulator including accurate OS, processor and interrupt models provides an efficient platform for early design space exploration.

5.4.6 Cache Evaluation

We evaluate our cache modeling approach by simulating parallel matrix multiplication tasks running on a dual-core 1.6 GHz Atom platform with a core-private 24K, 6-way set associative L1 and a shared last level 512K, 8-way set associative L2 cache. We execute our experiments under three different simulation modes: a conventional simulation updates the cache state at the fine granularity given by back-annotated execution times, while the temporally decoupled (TD) ATGA approaches with and without multi-core, out-of-order cache modeling (MOOC) method commit accumulated accesses only at sim-

ulation quantum boundaries. To analyze the effect of course-grain temporal decoupling on cache simulation accuracy, we run our experiments under two different simulation quanta: $1\mu s$ and $1ms$.

In order to evaluate the cache behavior under different memory accesses patterns, we simulate two algorithms, a typical naïve matrix multiplication algorithm and a cache-aware, blocked algorithm with fixed 32x32 blocks. For each algorithm, we simulate a variety of matrix sizes ranging from small size matrices that would fit entirely in the L1 cache to large matrices that exceed the L2 size.

We first verify the accuracy of our cache hierarchy model on a single-core execution. For this purpose, we compare cache miss rates obtained from simulation with the actual execution on a single-core Atom board. We use Valgrind [60] to monitor the Atom cache behavior.

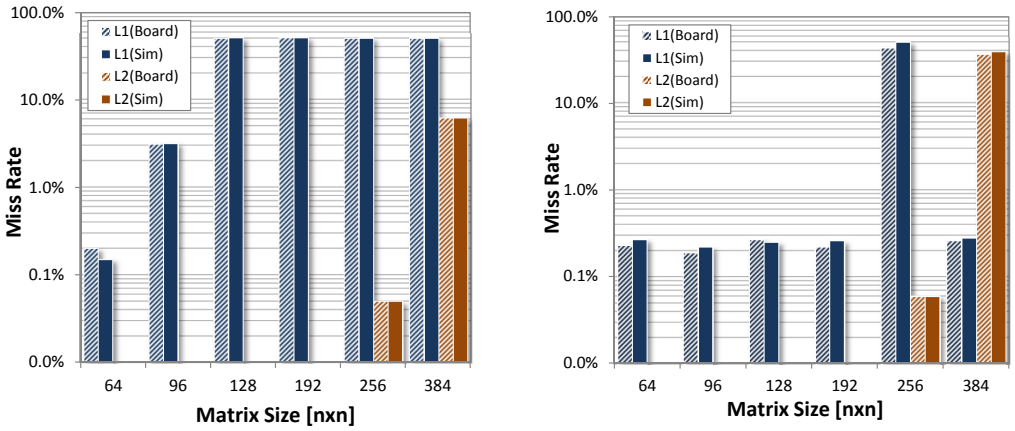
Table 5.6 shows the total number of accesses for the L1 cache and the L1 and L2 miss rates on the Atom board and compares these with simulation results. Residual errors are caused by the back-annotation process, which does not instrument all memory accesses. As summarized in Figure 5.13, the results depict that our cache simulator follows the reference cache behavior. Furthermore, low miss rates for large matrices confirm that a cache-aware implementation achieves a better performance.

Table 5.6: Cache accuracy results for matrix multiplication simulation on a single-core platform.

	Naïve Algorithm							Cache-Aware Algorithm						
Matrix	Total L1	L1 Miss Rate			L2 Miss Rate			Total L1	L1 Miss Rate			L2 Miss Rate		
Size	Accesses	Board	Sim.	Error	Board	Sim.	Error	Accesses	Board	Sim.	Error	Board	Sim.	Error
16	9,306	-	-	-	-	-	-	9,818	-	-	-	-	-	-
32	69,802	-	-	-	-	-	-	71,850	-	-	-	-	-	-
64	539,918	0.20%	0.15%	(-27.5%)	-	-	-	633,996	0.23%	0.27%	(16.9%)	-	-	-
96	1,750,378	3.12%	3.17%	(1.6%)	-	-	-	2,140,519	0.19%	0.22%	(17.3%)	-	-	-
128	2,127,242	50.1%	50.6%	(1.1%)	-	-	-	5,071,702	0.27%	0.25%	(-6.2%)	-	-	-
192	7,067,470	50.6%	50.9%	(0.7%)	-	-	-	17,123,704	0.22%	0.26%	(17.0%)	-	-	-
256	16,898,250	50.0%	50.3%	(0.6%)	0.05%	0.05%	(-0.22%)	40,572,930	43.0%	50.2%	(16.8%)	0.06%	0.06%	(0.02%)
384	56,578,186	50.3%	50.5%	(0.4%)	6.21%	6.21%	(0.03%)	136,988,302	0.26%	0.28%	(9.4%)	36.3%	38.9%	(7.25%)

Table 5.7: Measured and simulated execution time for matrix multiplication on single-core and dual-core platforms.

	Single-Core Platform						Dual-Core Platform					
	Naïve Algorithm			Cache-Aware Algorithm			Naïve Algorithm			Cache-Aware Algorithm		
Matrix	Exe. Time	Simulation Error		Exe. Time	Simulation Error		Exe. Time	Simulation Error		Exe. Time	Simulation Error	
Size	Board	w/ cache	w/o cache	Board	w/ cache	w/o cache	Board	w/ cache	w/o cache	Board	w/ cache	w/o cache
16	25μs	-6.2%	-6.2%	24μs	-5.0%	-5.0%	25μs	-7.5%	-7.5%	25μs	-7.5%	-7.5%
32	180μs	2.6%	2.6%	179μs	3.3%	3.3%	183μs	1.4%	1.4%	183μs	1.4%	1.4%
64	1,472μs	1.0%	0.6%	1,563μs	-4.5%	-5.2%	1,492μs	-0.3%	-0.7%	1,569μs	-4.8%	-5.6%
96	5,484μs	-0.5%	-8.8%	5,200μs	-3.2%	-3.9%	5,329μs	2.4%	-6.2%	5,179μs	-2.8%	-3.5%
128	29,631μs	-1.6%	-60.0%	12,335μs	-3.2%	-3.9%	29,638μs	-1.6%	-60.0%	12,320μs	-3.1%	-3.8%
192	101,630μs	-2.9%	-60.7%	41,156μs	-2.1%	-2.8%	101,597μs	-2.4%	-60.6%	41,082μs	-0.5%	-2.7%
256	247,855μs	-6.0%	-61.8%	259,449μs	-8.7%	-63.5%	248,944μs	8.1%	-61.9%	259,352μs	-7.5%	-63.5%
384	1,235,815μs	-10.9%	-94.6%	419,768μs	-20.4%	-23.8%	1,270,051μs	-8.5%	-74.8%	419,905μs	-20.4%	-23.8%



(a) Naïve algorithm.

(b) Cache-aware algorithm.

Figure 5.13: Accuracy results for L1 and L2 on the single-core platform.

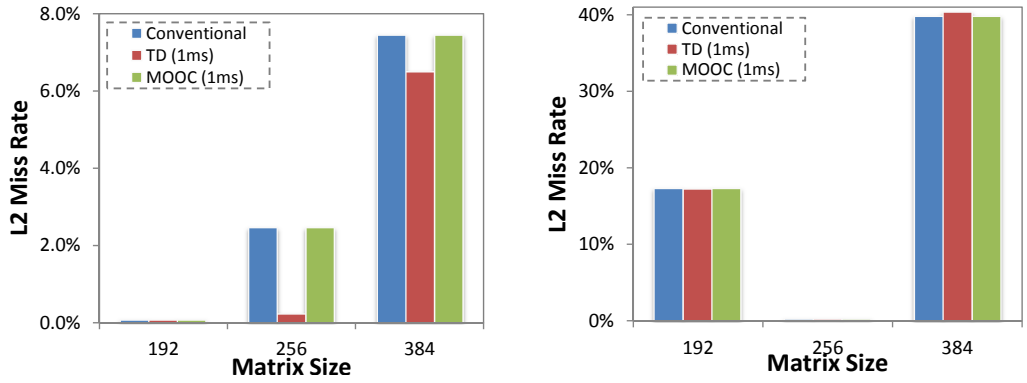
We further compare the simulated execution times with the reference execution on the board. In this experiment, we used a cache-memory calibration tool [13] to measure L1 and L2 miss latency cycles and annotated our cache model accordingly in order to accurately reflect the cache effects in the host-compiled simulation. The measured and simulated execution times for both the single-core and the dual-core platforms are shown in Table 5.7. Results show a significant improvement of the cache model on the execution time accuracy for matrices with high L1 or L2 miss rates. The average execution error in the presence of cache modeling was -3.8%, while neglecting the impact of cache behavior can generate up to 95% timing error. Note that as shown in Table 5.7, for a matrix size of 384 cache conflicts in the cache-aware algorithm lead to a high L2 miss rate which in turn results in a relatively high error, even with our cache model. This is due to lack of accurate models of system busses beyond the L2 in our setup.

Table 5.8: L2 miss rate and errors for conventional, MOOC, TD simulation of matrix multiplication on the dual-core platform.

	Naïve Algorithm						Cache-Aware Algorithm					
Matrix	L2	Miss	Miss Rate Error				L2	Miss	Miss Rate Error			
Size	Accesses	Rate	TD		MOOC		Accesses	Rate	TD		MOOC	
		Conv.	$1\mu s$	$1ms$	$1\mu s$	$1ms$		Conv.	$1\mu s$	$1ms$	$1\mu s$	$1ms$
16	0	-	-	-	-	-	-	-	-	-	-	-
32	0	-	-	-	-	-	-	-	-	-	-	-
64	1,536	-	-	-	-	-	2,870	-	-	-	-	-
96	112,896	-	-	-	-	-	8,064	-	-	-	-	-
128	4,262,398	-	-	-	-	-	21,504	-	-	-	-	-
192	14,454,982	0.08%	0.00%	-3.06%	0.00%	0.00%	76,200	17.3%	0.00%	-0.34%	0.00%	0.00%
256	33,830,910	2.46%	-73.3%	-90.4%	0.00%	0.00%	34,765,822	0.25%	0.00%	1.54%	0.00%	0.00%
384	114,482,686	7.43%	-16.0%	-12.6%	0.00%	0.00%	663,552	39.8%	0.00%	1.36%	0.00%	0.00%

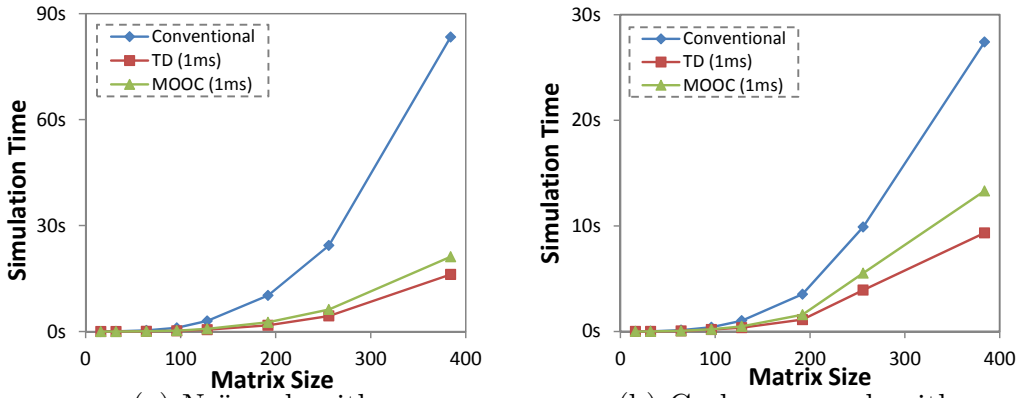
Finally, to demonstrate the efficiency of our MOOC approach in a temporally decoupled simulation, we compare L2 miss rates using different modeling setups. Table 5.8 shows the miss rates for TD and MOOC simulations. As expected, for the MOOC approach, miss rates were constant under different simulation quanta and were identical to the conventional simulation results. By contrast, the TD approach can exhibit large deviations due to out-of-ordered cache updates. Note that for the cache-aware algorithm, there is very little L2 interference between cores. As such, the ordering of accesses does not play a significant role and a naïvely decoupled simulation already provides good results. However, such behavior is hard to predict. By contrast, reordering approach provides consistently good results.

To summarize, Figure 5.14 compares L2 miss rates reported by different host-compiled simulation approaches. As can be seen, temporally decoupled simulation without reordering the memory accesses can result in large errors



(a) Naïve algorithm. (b) Cache-aware algorithm.

Figure 5.14: L2 miss rate for Conventional, MOOC, and TD simulations with quantum (1ms).

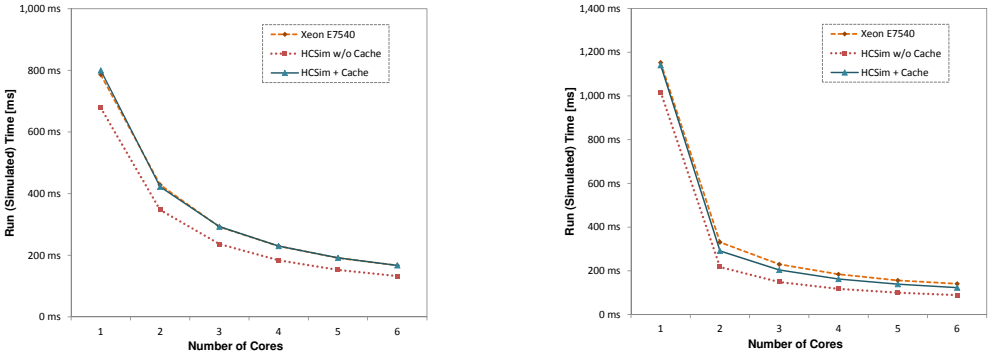


(a) Naïve algorithm. (b) Cache-aware algorithm.

Figure 5.15: Simulation time for Conventional, MOOC, and TD simulations with quantum (1ms).

for some configurations. Figure 5.15 plots the simulation time for the same experiments. Altogether, the integrated MOOC model provides a guaranteed accurate result while maintaining almost the full performance benefits of a temporally decoupled simulation.

Finally, to evaluate our cache model stressed under more realistic con-



(a) Naïve algorithm. (b) Cache-aware algorithm.

Figure 5.16: Galois host-compiled simulation w/ and w/o cache modeling (MOOC, quantum=1ms).

ditions, we apply our models to a multi-thread programming framework (Galois) [61] running a single source shortest paths (SSSP) algorithm on an input graph with 1,070,376 nodes. We run the application on a 6-Core 2.0 GHz Xeon board with core-private 32 KB, 8 Way L1 and 256 KB, 8 Way L2 caches and a shared 12 MB, 12 Way L3 cache. Task delays are measured using Vtune [41] on the reference board. Figure 5.16 compares the measured and simulated execution times for both FIFO- and Stack-based priority queues in Galois framework. We further run the experiment under different number of cores. Results show that with cache simulation average timing error of Fifo-based priority queue is improved to -0.8% from -18.8% without cache modeling. Similarly, for stack-based priority queue, the average error is decreases to -9.9% from the original -31.8%.

5.5 Summary

In this chapter, we presented a host-compiled multi-core system simulator designed for early real-time performance evaluation. At its core, the simulator consists of a configurable, abstract OS model, which emulates multi-core task scheduling. Furthermore, the OS model is embedded in a high-level multi-core processor model that replicates a generic multi-core interrupt handling chain and supports standard TLM interfaces for integration into co-simulation backplanes to provide a fast and accurate full-system HW/SW co-simulation platform. Experimental results demonstrate the efficiency of our simulator both on a suite of artificial task sets and an industrial-strength design example. Results show that compared to a reference ISS, simulations on the order of 1000 MIPS at less than 3% error can be achieved. Overall, experiments demonstrate the benefits of our configurable models for fast and accurate early design space exploration and software development.

We further presented a novel cache hierarchy simulation technique, which provides an accurate multi-core cache simulation for efficient system-level evaluation and exploration. Our approach introduces a multi-core, out-of-order cache model, which incorporates a delayed reordering of aggregated requests to provide an accurate cache simulation in the presence of temporal decoupling. Our results show that the highest possible accuracy is obtained by using our reordering technique, while increased simulation performance benefits from temporal decoupling.

Chapter 6

Parallel Simulation

In the preceding two chapters, we focused on improving the speed and accuracy tradeoff in host-compiled simulations through timing model management of platform models considering discrete-event behavior of SLDLs. In the presented approaches, however, single-threaded execution of traditional SLDL simulators still limits the simulation performance. With recent advances in parallel discrete-event simulators (PDES), parallel simulation of virtual platforms, such approaches have been widely accepted as a solution to provide fast simulation of multi-core and multi-processor platforms on parallel multi-core simulation hosts.

In this chapter, we propose a novel timing model management approach, which integrates with our previously presented solutions to align logical time advances in platform models along synchronized simulation times in order to fully utilize parallel resources available in host PCs. The rest of this chapter is organized as follows: In Section 6.1, we briefly discuss the behavior of parallel host-compiled simulation. In Section 6.2, we then present our synchronized timing model management solutions. Finally, in Section 6.3, we evaluate the performance benefits of our parallel timing models.

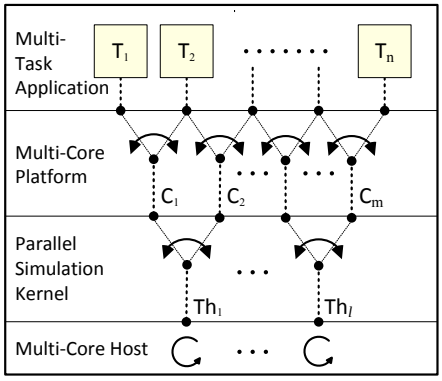


Figure 6.1: Parallel simulation of virtual platforms.

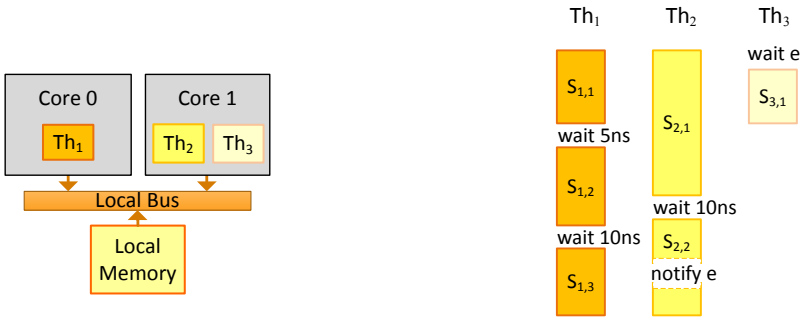
6.1 Parallel Host-Compiled Simulation

Abstract multi-core platform models paired with temporal decoupling approaches can provide faster simulation. However, in spite of the parallel nature of multi-core virtual platforms, the models are traditionally executed on top of a sequential simulator. Recently, PDESs have emerged as a solution for parallel simulation of virtual platforms, as shown in Figure 6.1. Basically, simulated platform resources are clustered and mapped to a dedicated core on a host PC by the underlying SLDL simulation kernel. The application threads are then managed and scheduled on the platform threads by an abstract OS model. In an ideal parallel simulation, the number of active host cores is equal or less than the total number of platform and application model threads at any simulation time.

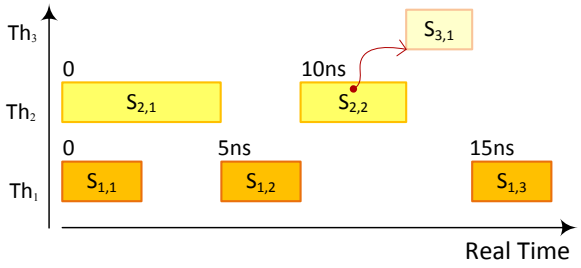
We explored different parallel discrete-event simulators in Section 2.3. A fully conservative PDES issues platform threads only if they are ready at the same simulation cycle [20]. As such, available host cores remain idle if

there are less active threads in a particular simulation cycle. In other words, the parallel simulation is strictly controlled by the timing behavior of platform models. By contrast, a predictive simulator issues multiple platform threads even if they are in different simulation cycles as long as there is no interaction between them [16]. However, such simulators require a static analysis of the dependencies, and as such, are not able to consider complex, dynamic platform behavior. Such problems can be solved by using a timing management model to align and synchronize concurrent threads, and when integrated into our OS model, this approach can further consider dynamic behavior of platforms for more efficient simulation.

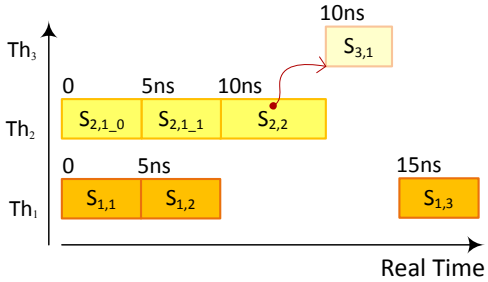
In the following, we present an example of platform simulation using existing PDSEs with and without an integrated, synchronized timing model management approach. Figure 6.2 (a) shows a simple dual-core platform model simulating three application threads, in which Th_1 is assigned to $core_0$ and threads Th_2 and Th_3 are running on $core_1$. Figure 6.2 (b) shows the conservative simulation of the example. As can be seen, except for the first simulation cycle, there is no chance that the platform threads can be issued in parallel. By contrast, Figure 6.2 (c) shows the same simulation with an integrated timing model management, where the timing manager splits the first segment of Th_2 in order to run both the second segment of Th_1 and Th_2 in the same simulation cycle. Figure 6.2 (d) shows the predictive simulation of the same example. In this simulation, the last segment of Th_1 cannot be issued, since in the general case, e.g. in the presence of dynamically issued dependencies (such



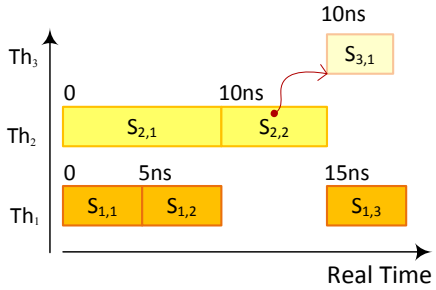
(a) A dual-core virtual platform example.



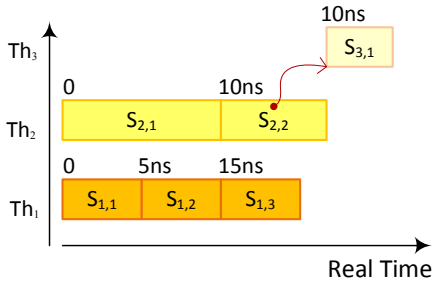
(b) Conservative parallel simulation.



(c) Conservative simulation w/ timing model management.



(d) Predictive parallel simulation.



(e) Predictive parallel simulation w/ timing model management.

Figure 6.2: Parallel simulation of a dual-core virtual platform using different parallelization approaches.

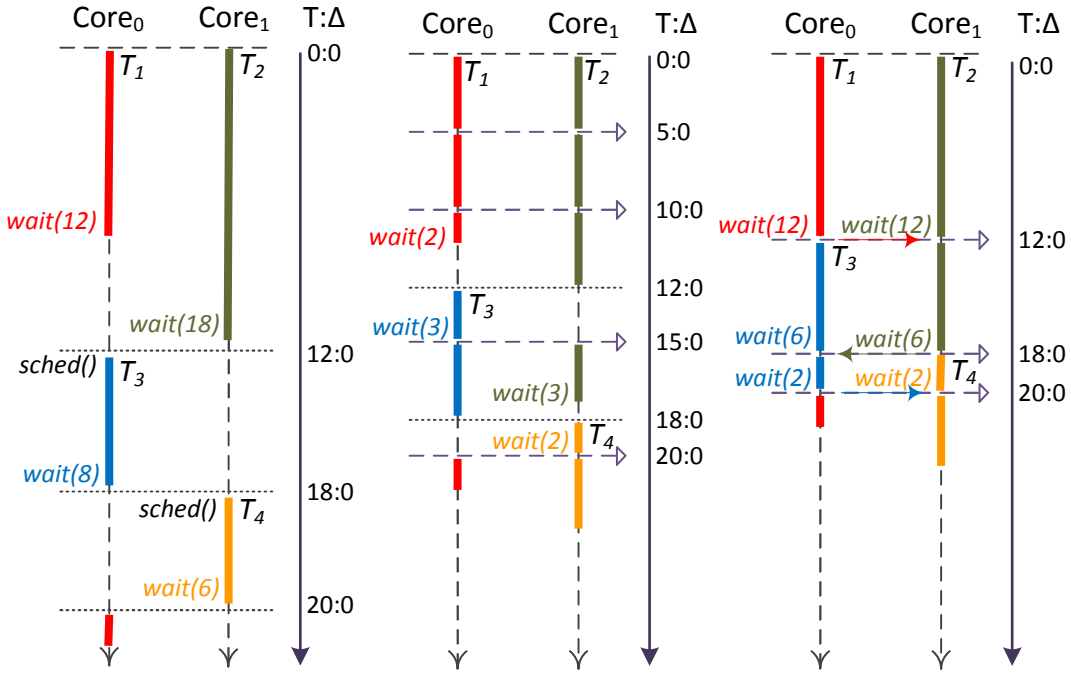
as a dynamically executed wait statement), a static analysis will not be able to determine that Th_1 is mapped to a separate platform core, and is independent of both Th_2 and Th_3. Figure 6.2 (e) then shows an ideal simulation, in which the platform timing model can manage run-time dependency checking to align and synchronize execution of concurrent threads in order to provide the most efficient parallel simulation.

6.2 Synchronized Timing Model Management

In the ATGA approach, the OS model manages the simulation timing model internally to only advance the simulation time and call the scheduler at right task switches points. In this approach, each core keeps a local time and can go a head of the simulated time until a task switch or a global synchronization such as a hardware interaction needed. We utilize the advantage of our integrated ATGA model to provide efficient parallel simulation via dynamic timing alignment of platform threads. We thereby propose three different schemes to perform such a timing synchronization for aligned parallel simulation:

6.2.1 Static Quantum Model

An ATGA-based OS model accumulates tasks execution delays internally, and decides to advance the simulated time when the accumulated delay exceeds predicted scheduling points. As such, platform cores can go ahead of the global time as long as there are no task switches. Figure 6.3 (a) shows the



(a) Traditional. (b) Static quantum. (c) Dynamic quantum.

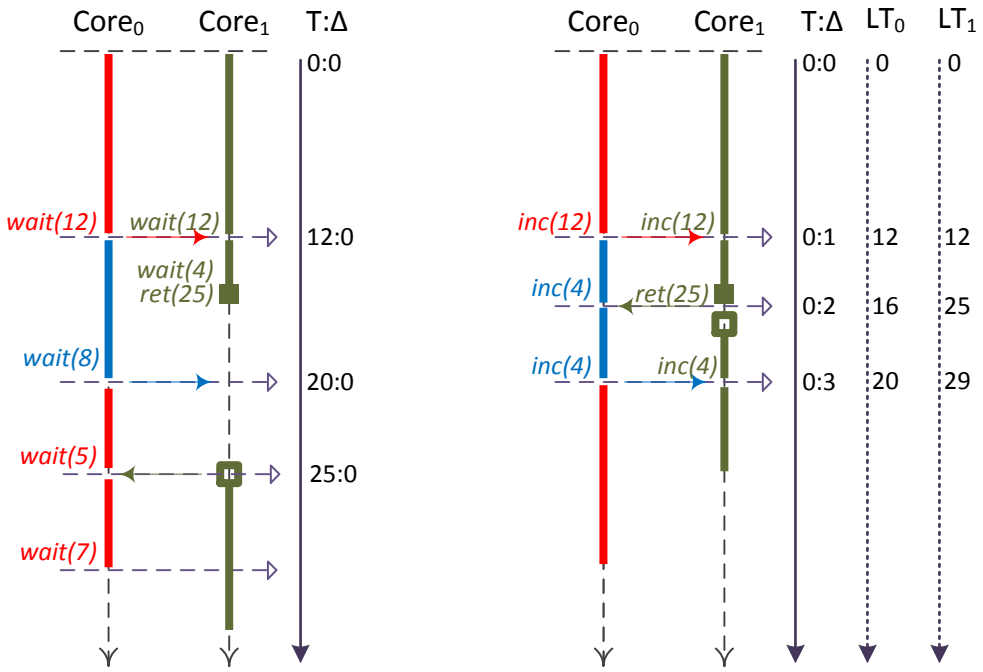
Figure 6.3: Synchronized timing models coupled with the ATGA approach.

parallel execution trace of a dual-core platform model. At the beginning of the simulation, both platform threads are issued on host cores, since they are ready at the same simulation cycle. At local time 12, the task running on $core_0$ requests to advance time and calls the OS scheduler. At this point, the associated host core becomes idle waiting for the other core to finish the execution of the assigned platform thread. After advancing the simulated time to 12, only the $core_0$ thread is ready and can be issued, while the other host core stays idle until the next simulation cycle. In such a timing model, there are several gaps in the execution trace, in which a host core is idle due to desynchronized platform threads. To solve this problem, we extend our ATGA approach by

simply aligning the predicted delay of different cores to fixed, statically selected quantum boundaries, such that platform threads are synchronized and can be issued concurrently. This can improve the performance in conservative parallel simulations, in which only threads in the same simulation cycles can be issued in parallel. As can be seen in Figure 6.3 (b), both platform threads are synchronized again at quantum boundaries of simulated times 15 and 20, and can be issued on host cores in parallel accordingly. However, platform threads can still be desynchronized at task scheduling points.

6.2.2 Dynamic Quantum Model

To efficiently manage thread alignment points, simulation quanta are determined dynamically by task-switches points. In the ATGA approach with static quanta, the simulation cycle advances to the next scheduling points when a task needs to be scheduled on a core. This can still lead to tasks becoming desynchronized on the underlying simulation kernel. To combat this problem, in an ATGA approach with dynamic quanta, only when a core needs to move to the next simulation cycle (i.e. a task scheduling point) will it inform other cores to terminate their running task and also move to the next, globally announced simulation cycle. As such, timing models of different cores are effectively synchronized with a dynamically selected, global quantum boundary. Figure 6.3 (c) shows the execution trace of our example using dynamic quantum boundary synchronizations. As can be seen, all threads are dynamically synchronized at task scheduling points to keep host cores busy.



(a) ATGA timing model. (b) ATGA$^+$ timing model.

Figure 6.4: Fully decoupled ATGA timing model.

6.2.3 ATGA$^+$ Timing Model

In the ATGA approach, the global time is advanced whenever a task scheduling is required or a quantum boundary has been reached. However, platform threads may still move to different simulation cycles when a core switches to idle state, e.g. while waiting for the start of the next period of any task assigned to the core. This can result in available host cores remaining idle while waiting for the next synchronization time. Figure 6.4 (a) shows the execution trace of a dual-core platform, in which the thread running on $core_1$ moves to and remains in idle state until the simulated time 25. As such, the host core remains idle until the other thread has reached this next

synchronization point, i.e. the start time of $core_1$ thread. To improve on this situation, we further enhance our ATGA approach by advancing the simulated time only when a hardware interaction is needed or an explicit inter-process synchronization is requested by platform threads. In such an approach, cores continue to accumulate task delays and increment their local time even if a task scheduling happens. This allows cores to remain synchronized longer at the same simulated time, where cores can only differ in their delta cycles, which have to be advanced on each scheduling event. In this way, the host machine is utilized more efficiently. As shown in Figure 6.4 (b) both cores are fully utilized, since platform cores are completely decoupled and $core_1$ was able to immediately jump to the next simulation cycle by only incrementing its local time.

6.3 Experiments and Results

To evaluate the performance benefits of our modeling approaches, we compare the simulation speed of different timing models to a reference single-thread simulation of the conventional ATGA model. All experiments are simulated using a conservative parallel SpecC simulator [20] on top of a Quad-Core 2.67 GHz Intel Core i7 workstation. We apply our timing models to randomly

Table 6.1: Periodic task sets characteristics.

Set	S1	S2	S3	M1	M2	M3	L1	L2	L3
Number of Tasks	42	31	47	16	16	16	12	11	13
Avg. Task Weight	0.06	0.06	0.06	0.15	0.16	0.15	0.22	0.27	0.23
CPU Utilization	2.35	1.68	2.74	2.43	2.83	2.62	2.61	2.98	3.01

Table 6.2: Parallel simulation results using ATGA timing model.

Sets	Parallel Simulation					
	CPU = 1		CPU = 2		CPU = 4	
	Sim. time (Util.)	Speedup	Sim. time (Util.)	Speedup	Sim. time (Util.)	Speedup
S1	22.46s (99%)	1	22.01s (120%)	1.02	21.31s (105%)	1.05
S2	14.61s (99%)	1	15.5s (122%)	1.15	13.65s (129%)	1.07
S3	27.14s (99%)	1	25.98s (121%)	1.04	24.84s (124%)	1.09
M1	17.28s (99%)	1	15.79s(126%)	1.09	15.1s(133%)	1.14
M2	25.25s (99%)	1	22.03s (123%)	1.25	20.57s (128%)	1.23
M3	20.86s (99%)	1	20.25s (124%)	1.03	18.44s (131%)	1.13
L1	23.74s (99%)	1	20.48s(133%)	1.16	18.26s (149%)	1.30
L2	26.6s (99%)	1	23.75s (130%)	1.12	22.22s (142%)	1.20
L3	26.54s (99%)	1	25.01s(129%)	1.06	22.99s (137%)	1.15

Table 6.3: Parallel simulation results using ATGA timing model w/ the synchronized static quanta ($Quantum = 100\mu s$).

Sets	Parallel Simulation					
	CPU = 1		CPU = 2		CPU = 4	
	Sim. time (Util.)	Speedup	Sim. time (Util.)	Speedup	Sim. time (Util.)	Speedup
S1	20.41s (99%)	1.10	14.73s (131%)	1.52	14.35s (142%)	1.57
S2	13.65s (99%)	1.07	11.02s (137%)	1.33	10.2s (151%)	1.43
S3	21.68s (99%)	1.25	18.55s (148%)	1.46	15.38s (180%)	1.76
M1	15.29s (99%)	1.13	11.9s (141%)	1.45	10.46s (155%)	1.65
M2	20.31s (99%)	1.25	15.17s (143%)	1.66	12.7s (170%)	1.99
M3	18.5s (99%)	1.13	13.48s (143%)	1.55	11.69s (166%)	1.78
L1	19.8s (99%)	1.20	15.45s (153%)	1.54	11.75s (193%)	2.02
L2	22.8s (99%)	1.17	17.15s (157%)	1.55	11.97s (205%)	2.22
L3	25.47s (99%)	1.04	18.48s (155%)	1.44	13.6s (196%)	1.95

generated periodic task sets (as described in Section 3.2) executed on a simulated quad-core platform. Task priorities are assigned inversely to their periods following a rate-monotonic scheduling scheme. Execution delays are modeled by a loop over basic arithmetic operations and are back-annotated at the granularity of 1 μs. We run each task set for 10 sec of simulated time. A summary of tasks' characteristics is shown in Table 6.1.

Table 6.4: Parallel simulation results using ATGA$^+$ model w/ the synchronized static quanta ($Quantum = 100\mu s$).

Sets	Parallel Simulation					
	CPU = 1		CPU = 2		CPU = 4	
	Sim. time (Util.)	Speedup	Sim. time (Util.)	Speedup	Sim. time (Util.)	Speedup
S1	21.67s (99%)	1.04	17.37s (164%)	1.29	11.67s (215%)	1.92
S2	17.33s (99%)	0.84	12.93s (160%)	1.13	6.7s (144%)	2,18
S3	29.55s (99%)	0.92	20.12s (165%)	1.35	11.88s (232%)	2.28
M1	17.09s (99%)	1.01	13.98s (164%)	1.24	7.05s (238%)	2.45
M2	23.74s (99%)	1.06	19.03s (163%)	1.33	8.84s (248%)	2.86
M3	20.06s (99%)	1.04	16.44s (161%)	1.27	7.86s (244%)	2.65
L1	19.12s (99%)	1.24	18.75s (172%)	1.27	8.54s (284%)	2.78
L2	26.69s (99%)	1	20.2s (170%)	1.32	9.19s (265%)	2.89
L3	26.92s (99%)	0.99	20.67s (171%)	1.28	9.32s (267%)	2.85

Table 6.2 shows the simulation time and the speedup for our conventional ATGA timing model. As can be expected, no performance gain is achieved by parallel simulation, since platform threads are rarely synchronized to execute in the same simulation cycle. By contrast, Table 6.3 shows that by a simple alignment of platform threads at quantum boundaries, an average speedup of 1.8 is achieved. However, the simulation performance is still limited by non-synchronized timing models in the presence of task switches and context-switch overheads.

Table 6.4 and Table 6.5 list the simulation speedup for the ATGA$^+$ approach with static quantum and dynamic quantum models, respectively. Overall, the ATGA$^+$ approach provides faster simulation by decreasing the overhead of advancing the simulated time. Results show that synchronized timing models with static and dynamic quantum selection can run 2.6× and 3.5× faster, respectively, compared to the single-threaded simulation of con-

Table 6.5: Parallel simulation results using ATGA w/ the synchronized dynamic quanta.

Sets	Parallel Simulation					
	CPU = 1		CPU = 2		CPU = 4	
	Sim. time (Util.)	Speedup	Sim. time (Util.)	Speedup	Sim. time (Util.)	Speedup
S1	16.99s (99%)	1.32	10.43s (170%)	2.15	8.77s (225%)	2.56
S2	11.76s (99%)	1.24	7.91s (167%)	1.85	5.36s (251%)	2.73
S3	19.75s (99%)	1.37	13.29s (170%)	2.04	9.55s (252%)	2.84
M1	13.32s (99%)	1.30	9.12s (161%)	1.89	5.53s (244%)	3.12
M2	17.08s (99%)	1.48	10.91s (171%)	2.31	5.76s (284%)	4.38
M3	15.73s (99%)	1.33	10.1s (168%)	2.07	6.13s (244%)	3.40
L1	17.29s (99%)	1.37	11s (171%)	2.16	5.23s (309%)	4.54
L2	19.38s (99%)	1.37	12.08s (173%)	2.20	6.57s (283%)	4.05
L3	19.88s (99%)	1.34	12.83s (172%)	2.07	6.32s (293%)	4.20

ventional models.

To summarize, we compare the average speedup of our timing models simulated on different number of host cores (shown in Figure 6.5). As can be seen, the ATGA$^+$ approach with dynamically aligned platform threads is able to fully utilize the host workstation regardless of the timing behavior of the original platform model itself.

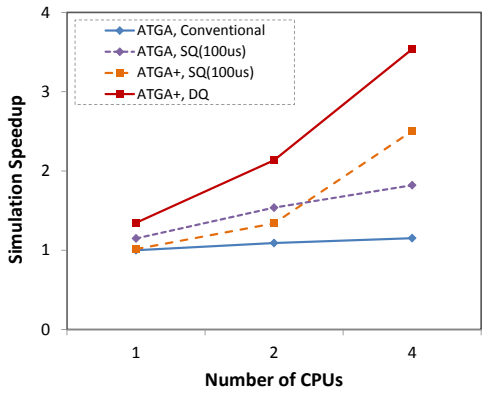


Figure 6.5: Average simulation speedups.

6.4 Summary

In this chapter, we examined the execution of conventional host-compiled platform models on top of typical parallel discrete-event simulators. Accordingly, we presented a synchronized timing model management designed to address the need for efficient parallel simulation of platform threads. Coupled with the ATGA model, platform threads are automatically aligned to quantum boundaries, in which the parallel simulator is able to issue them concurrently. We further enhanced the performance of our ATGA approach by temporally decoupling platform threads even beyond task scheduling points. Furthermore, with dynamically selecting the alignment boundaries, unnecessary synchronization points are eliminated to provide more efficient parallel simulations. Experimental results demonstrate the efficiency of our timing models on a suite of periodic task sets running on a simulated quad-core platform. Results show that compared to a reference single-thread parallel simulation of our conventional models, a speedup of up to 4.5× is achieved on a quad-core host PC, while the conventional model is only simulated up to 1.3× faster. Overall, our results show that the ATGA$^+$ approach with dynamically selected quantum boundaries can provide a fully utilized parallel simulation.

Chapter 7

Summary, Conclusions and Future Research

In this chapter, we first summarize our contributions and results, then discuss possible research directions.

7.1 Summary and Conclusions

In this dissertation, we have studied and addressed inefficiencies and tradeoffs in accuracy and speed of host-compiled simulation approaches. As the basis of our research, we have first identified and classified different sources of errors in convention host-compiled platform models. An analytical model of error behavior has thereby shown that, under certain conditions, errors can exceed the bounds set by the discrete timing model itself. We have accordingly proposed dynamic timing model management approaches to avoid such errors and improve the accuracy and speed tradeoff.

For the infrastructure of our research, we have presented a novel host-compiled multi-core platform simulator. At its core, the simulator consists of a configurable, abstract OS model, which emulates SMP task scheduling. Furthermore, the OS model is embedded in a flexible, high-level multi-core processor model that replicates a generic multi-core interrupt handling chain

and supports standard TLM interfaces to provide a full-system HW/SW simulation platform.

In order to improve the speed and accuracy, we have proposed an automatic timing granularity adjustment (ATGA) approach in which, when coupled with the OS model, the simulator itself automatically controls the timing granularity to provide an error-free task preemption model. In this approach, the OS model continuously monitors the state of the system, and accumulates or breaks statically annotated delays into a number of smaller steps as needed. Results show that using the ATGA timing model, a fast and accurate simulation is achieved regardless of discrete timing granularities in the original code.

In a multi-core configuration of the ATGA model, temporally decoupled execution of platform cores can cause memory accesses to be globally committed out-of-order to a shared cache model. As a solution, we have further introduced a multi-core, out-of-order cache hierarchy model, which incorporates a delayed reordering of aggregated requests to provide a fast and accurate host-compiled multi-core cache simulation. Our results show that the highest possible accuracy is obtained by using our reordering technique, while increased simulation performance benefits from the ATGA approach.

Finally, we have examined parallel simulation of host-compiled platform models via well-established PDES approaches. As our observations demonstrated, the parallel simulation benefit is highly limited by the inherent timing behavior of platform models. As a solution, we have introduced an approach

for synchronized timing models, which align platform threads to quantum boundaries in order to expose more parallelism to the underlying parallel simulator. We have realized a fully utilized parallel simulation where such approaches are incorporated into the OS and ATGA models.

7.2 Future Directions

In the following, we outline possible research directions in this domain to further improve accuracy and speed tradeoffs or to tackle simulation performance bottlenecks that still exist in complex multi-core platform modeling.

7.2.1 Platform Manager

In the ATGA approach, the OS model monitors the state of all tasks running on the system, and based on such information predicts the next possible scheduling point in order to advance the simulation time in larger granularities. In situations when the OS model is not able to predict the next scheduling point due to an external event, it switches to an expensive and fine-grained fallback mode until all events are captured. However, moving to fallback mode can degrade the simulation performance dramatically, especially for systems with high HW/SW interactions.

To tackle such a problem, the key concept of the ATGA approach can be applied to establish a complete *platform manager* (PM) model. In this case, the OS model is extended to monitor all resources and components in the system, including both application tasks running on top of the processor

model as well as hardware units connected to the system bus and interacting with the application software tasks. In this way, the platform manager is able to monitor the state of hardware units in a larger context, and thus eliminate unnecessary fallback modes by predicting event release times.

In order to implement a platform manager, all communication channels including bus and interrupt handling models, as well as inter-task communication primitives are instrumented to keep the track of all data dependencies between different components and tasks inside the platform model. Following the ATGA model, a platform manager can also accumulate hardware delays and only advance the time when an external interaction needed. In such a way, we speculate that it is possible to achieve fast full-system simulation while still providing accurate results.

7.2.2 Parallel Simulation

In this dissertation, we have shown that by careful timing model management and optimal temporal quantum alignment, available cores on host PCs can be fully utilized for efficient parallel simulation. However, there are several directions for improvements in this area:

Synchronized timing model supporting hardware interactions. In the ATGA$^+$ approach, platform cores are fully decoupled, and the simulator moves to the next simulation cycle by advancing delta cycles only without actually advancing the time. In this approach, the total time is defined by a core's local time plus the global simulated time. When a task communicates with

hardware, however, the simulated time is required to be synchronized with the total time of that core in order to provide timing-accurate hardware interactions. Consequently, platforms threads may move to different simulation cycles causing parallel simulation benefits to suffer.

Such a problem can be solved by introducing an extra timing synchronization level on top of our dynamic quantum alignment approach. In this approach, when a core requests a hardware interaction, the timing model globally distributes such synchronization points over platform threads in order to efficiently move them to the same simulation cycle. Using this approach, we expect to see the performance benefit of parallel simulation even in platforms with a high degree of HW/SW interactions.

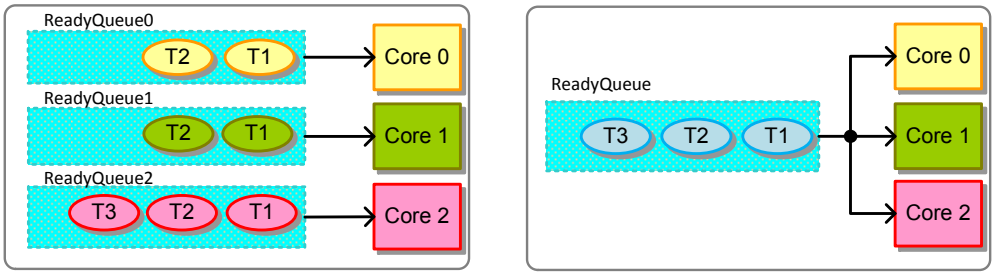
Parallel Cache Simulation. High simulation overhead of cache models has severely limited such models in host-compiled platform simulations. High-level cache models only emulate miss/hit behavior and do not need to be implemented as an SLDL process. In other words, a cache model can be executed in a separate thread next to the SLDL simulation kernel itself. In a host-compiled performance evaluation, cache penalties can be considered at the end of simulation or when an explicit synchronization is required. As such, with reduced inter-thread communications via lock-free channels, and with a balanced computation distribution, we anticipate to see improved cache simulation performance through such parallelization and separation of cache, functional and timing models.

Appendix

Appendix 1

Dynamic Task Scheduler Modeling

With recent trends in multi-core processor design, multi-core schedulers have become crucial components for the performance of real-time embedded systems. Figure 1.1 shows two common classes of multi-core scheduling schemes. The prevalent scheme has been a partitioned queue, in which each task is statically assigned to a fixed core and queue. An alternative is a global scheduling scheme, in which there is a single ready queue, and tasks are assigned to any available core according to a global priority policy. In Chapter 4, we evaluate the integration of common partitioned and global scheduling schemes using fixed priorities into the host-compiled OS model. In this appendix, we focus on two well-studied global scheduling policies with dynamic priorities (G-EDF and Pfair) to further examine the OS model on adopting new



(a) Partitioned scheme. (b) Global scheme.

Figure 1.1: Multi-core scheduling schemes.

schedulers and features. The integration of such dynamic schemes, however, requires slight changes in the structure of our scheduler and timing model.

For the rest of our discussions, we consider that an application consists of a set $\tau = \{T_1, T_2 \dots T_n\}$ of periodic tasks running on a multi-core processor. A periodic task T_i repeatedly releases jobs at its release times r_i. The j^{th} job of task T_i is denoted as $J_{i,j}$. A per-job execution cost and task period are denoted by e_i and P_i, respectively. We refer to the ratio of $wt_i = e_i/P_i$ as the weight of the task T_i. We further assume that processor time is advanced by discrete-time quanta, and all tasks' parameters are integer multiples of the quantum length.

1.1 G-EDF Scheduling

An earliest-deadline-first (EDF) scheduler, as its name implies, assigns a higher priority to a task with the earliest deadline. Accordingly, a *global EDF* (G-EDF) scheduler dispatches tasks from a global *Ready* queue in order of increasing deadline until either all cores are busy or no more task is ready [1]. As such, G-EDF automatically balances the load among the cores. In the following, we present details of a version of our host-compiled OS model that incorporates a G-EDF scheduler based on the algorithm implemented in a Linux extension called $LITMUS^{RT}$ [12].

Timing model management. In a G-EDF scheme, the task scheduler is called when a task finishes executing or when a new job of a periodic task becomes ready (i.e. at the tasks' next release time). To call the sched-

```
Function PredictNextPreemptionTime:
1    idleTask := getFirst(IdleQueue)
2    predictedDelay := idleTask.NextPeriodTime - CurrentTime()
3    for all coreID in active CPU list do
4     predictedDelay := Min(predictedDelay, CPU[coreID].currentTask.NextPeriodTime - CurrentTime())
5    endfor
6    return predictedDelay
```

Figure 1.2: G-EDF scheduling point prediction.

```
Function TimeWait (long long nsec, task runningTask):
1    runningTask.AccDelay + = nsec
2    predictedDelay := PredictNextPreemptionTime()
3    while runningTask.AccDelay > predictedDelay do
4      SLDL::wait(predictedDelay)
5      runningTask.AccDelay − = predictedDelay
6      GEDFScheduleTick()
7      Wait4Sched(runningTask)
8      predictedDelay := PredictNextPreemptionTime()
9    endwhile
```

Figure 1.3: G-EDF timing model.

uler at the right times, the algorithm shown in Figure 1.2 determines the next possible preemption point as the first release time among the first task in the *Idle* queue and all running tasks. Note that the *Idle* queue is sorted based on the tasks' next release times. As such, only the first task needs to be checked

```
Function GEDFScheduleTick():
 1    GEDF_Lock.Acquire()
 2    for all releasedTask do
 3      insertGEDFPriority(ReadyQueue, releasedTask)
 4    endfor
 5    if Check4Preemptions() then
 6      for all coreID in the preempted cores list do
 7        Dispatch(coreID);
 8      endfor
 9    endif
10    GEDF_Lock.Release()
```

Figure 1.4: G-EDF scheduler.

to determine the next scheduling point. We further overload the `TimeWait()` method from the OS API to replicate a G-EDF scheduling model (shown in Figure 1.3). In this model, when the accumulated delay has reached the predicted scheduling boundary (line 2-3), the simulated time is advanced and a G-EDF scheduler is called accordingly (line 6).

G-EDF Scheduler. Figure 1.4 shows the algorithm that implements the G-EDF scheduler. The shared queue in the scheduler is protected using a global lock in order to prevent concurrent accesses. When the scheduler is called, just recently released tasks are first inserted into the global *Ready* queue according to an EDF policy (line 3), in which tasks with earliest deadlines are located first, and tasks with the same deadline are then listed based

```
Function Check4Preemptions():
 1    preemptionNeeded := false
 2    lastCPU := GEDFLowestPriorityCPU()
 3    readyTask := peekFirst(ReadyQueue)
 4    while GEDFHigherPriority(readyTask , lastCPU.currentTask) do
 5      lastCPU.newTask = getFirst(ReadyQueue)
 6      lastCPU.preempted := preemptionNeeded := true
 7      GEDFUpdateCPUPriorityList()
 8      lastCPU = GEDFLowestPriorityCPU()
 9      readyTask := peekFirst(ReadyQueue)
10    endwhile
11    return preemptionNeeded;
```

Figure 1.5: G-EDF check for task preemptions.

on the increasing order of task IDs. Next, the `Check4Preemption()` function is invoked to examine all running tasks for possible preemptions by the released tasks (line 5). Finally, the actual preemption is implemented when the `Dispatch()` function is called for all preempted cores (lines 6-8).

An efficient implementation of the `Check4Preemption()` function is shown in Figure 1.5. The OS model internally lists cores in an increasing order of the priority of associated running tasks. Hence, the core at the top of the list is running the task with the least EDF priority. As long as the task on top of the *Ready* queue has a higher priority compare to the task running on the least priority core, the OS schedules a new task on the least priority core

```
Function Dispatch(int coreID):
1    if CPU[coreID].preempted then
2      insertGEDFPriority(ReadyQueue, CPU[coreID].currentTask)
3      CPU[coreID].currentTask := CPU[coreID].newTask
4      SendSched(CPU[coreID].currentTask)
5    endif
```

Figure 1.6: G-EDF Dispatch method.

and updates its position in the core list accordingly (lines 4-10).

Finally, the actual preemption is implemented in the `Dispatch()` function (Figure 1.6), in which the current task is moved to the *Ready* queue and the previously assigned task is notified for scheduling.

1.2 Pfair Scheduling

In a perfectly fair (ideal) schedule of periodic tasks, every task receives wt_i quanta over the interval [0, t). This implies that all deadlines are met if the total utilization is less than the number of available cores [87]. In Pfair scheduling, each task is divided into a sequence of quantum-size subtasks $T = \tau_1, \tau_2 \ldots \tau_e$. Each subtask τ_i has an associated release time $r(\tau_i)$ and deadline $d(\tau_i)$, defined as:

$$r(\tau_i) = \lfloor \frac{i-1}{wt_i} \rfloor, \qquad d(\tau_i) = \lceil \frac{i}{wt_i} \rceil \qquad 1 \leq i \leq e \tag{1.1}$$

The Pfair scheduling algorithm thereby schedules subtasks on an earliest-

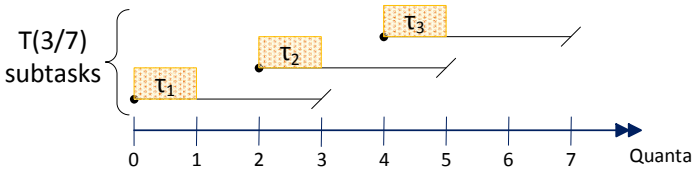


Figure 1.7: Pfair subtasks for task T(e=3,P=7).

deadline-first basis during their $[r(\tau_i), d(\tau_i))$ intervals [87]. An example of Pfair subtasks is given in Figure 1.7. Task T with weight $\frac{3}{7}$ is divided into three subtasks, and in each interval only one quantum-size subtask is scheduled even if the core is idle and no other task is ready. In the following, we present the integration of Pfair scheduling into our OS model based on the algorithm developed in $LITMUS^{RT}$ [12].

Timing model management. Figure 1.8 shows the pseudo code of the `TimeWait()` method, in which the PFair scheduler is called every time that

```
Function TimeWait (long long nsec, task runningTask):
1    runningTask.AccDelay + = nsec
2    while runningTask.AccDelay > SCHEDULING_QUANTUM do
3      SLDL::wait(SCHEDULING_QUANTUM)
4      runningTask.AccDelay − = SCHEDULING_QUANTUM
5      currentCoreID := GetSchedCoreID(runningTask)
6      PfairScheduleTick(currentCoreID)
7      Wait4Sched(runningTask)
8    endwhile
```

Figure 1.8: Pfair timing model.

```
Function PfairScheduleTick(int coreID)
1    Pfair_Lock.Acquire()
2    AdvanceSubtasks(CurrentTime())
3    PollReleases(CurrentTime())
4    ScheduleSubtasks()
5    for all coreID in the cores list do
6      Dispatch(coreID);
7    endfor
8    Pfair_Lock.Release()
```

Figure 1.9: Pfair scheduler.

the scheduling quantum boundary is reached (line 6). To replicate such behavior, the OS model accumulates task delays and only advances the simulated time by a fixed, quantum-length value (line 3).

Pfair Scheduler. The internals of Pfair scheduling are shown in Figure 1.9 and Figure 1.10. Again, since the scheduler is globally accessible by all cores, it is protected using a global lock. As shown in Figure 1.9, in the first step, all subtask are updated and moved to the *Idle* queue (lines 2-3). Next the released subtasks are scheduled based on an EDF strategy, details of which are shown in Figure 1.10. Finally, the `Dispatch()` method physically assigns the scheduled task to the current core, as shown in Figure 1.11.

```
Function ScheduleSubtasks()
1    for all coreID in CPU list do
2      readySuntask := peekFirst(ReadyQueue)
3      if PfairHigherPriority(readySubtask, CPU[coreID].currentTask do
4        CPU[coreID].newTask := getffirst(ReadyQueue)
5      else
6        CPU[coreID].newTask := Null
7      endif
8    endfor
```

Figure 1.10: Pfair subtasks scheduler.

```
Function Dispatch(int coreID):
1    CPU[coreID].currentTask := CPU[coreID].newTask
2    if CPU[coreID].currentTask ! = Null then
3      SendSchedule(CPU[coreID].currentTask)
4    endif
```

Figure 1.11: Pfair Dispatch() method.

1.3 Experiments and Results

To evaluate the accuracy of our models, we use randomly-generated periodic task sets. Task periods are uniformly distributed over [10, 100] ms, and task weights are distributed over [0.001, 0.4]. We run each task set for 10 s of simulated time. Experiments are executed on a simulated host-compiled dual-core platform. For accuracy measurements, task response times are com-

Table 1.1: Multi-core scheduling accuracy results.

Periodic Task Sets		τ_1	τ_2	τ_3	τ_4	τ_5
Number of Tasks		5	6	7	7	8
Avg. Task Weight ($wt_i = \frac{e_i}{P_i}$)		0.245	0.271	0.184	0.172	0.130
Total Utilization ($U = \sum wt_i$)		1.227	1.628	1.289	1.204	1.043
G-EDF	Avg. Error	0.34%	1.54%	0.59%	0.65%	0.72%
	Max. Error	9.28%	59.14%	17.18%	50.38%	81%
Pfair	Avg. Error	0.76%	3.66%	3.20%	0.29%	1.26%
	Max. Error	18.75%	24.76%	42.85%	4.10%	14.25%

pared to a reference execution on a dual-core 1.6 GHz Atom platform running $LITMUS^{RT}$ with a scheduling quantum of 1 ms.

Table 1.1 summarizes task set features and accuracy measurements for both G-EDF and Pfair scheduling policies. The average timing errors for G-EDF and Pfair schedulers are measured to be 0.8% and 1.8%, respectively. However, very large timing errors of up to 81% are observed for some iterations. Deeper investigations of related task scheduling traces show that the reference platform schedules the tasks differently across similar situations. This happens when more than two tasks have the same deadline, and the scheduler therefore determines the higher priority task based on some unobservable and hence not easily replicable other parameters, such as Linux process ID. Overall, given the fact that $LITMUS^{RT}$ is an extension of the Linux kernel, which is not in itself a fully deterministic and predictable real-time OS, high errors may be observed in our model when comparing to such a platform.

1.4 Summary

In this appendix, we presented the integration of two common, more complex multi-core scheduling algorithms into our host-compiled OS model. We can observe that the new scheduling policies can be easily integrated into our OS model by overloading the scheduler and timing model management functions. Furthermore, when comparing our models to scheduler behavior on a real platform, results show that our models are able to accurately track and predict actual system behavior, in as much as it can be predicted from given parameters.

Bibliography

[1] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2005.

[2] ARM Generic Interrupt Controller Architecture Specification, Co.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *Computer*, 35(2):59 –67, February 2002.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATEC)*, April 2005.

[5] G. Beltrame, L. Fossati, and D. Sciuto. ReSP: A nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28(12):1857 –1869, December 2009.

[6] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 41(2):169–182, September 2005.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell,

M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[8] G. Blake, R. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, November 2009.

[9] A. Bouchhima, I. Bacivarovx, 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)*, January 2005.

[10] A. Bouchhima, P. Gerin, and F. Petrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2009.

[11] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2003.

[12] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS RT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the International Real-Time Systems Symposium (RTSS)*, December 2006.

[13] A cache-memory and TLB calibration tool., 2013. `http://homepages.cwi.nl/~manegold/Calibrator/`.

[14] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, and H. Meyr. A high-level virtual platform for early MPSoC software development. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2009.

[15] S. Chakravarty, Z. Zhao, and A. Gerstlauer. Automated, retargetable back-annotation for host compiled performance and power modeling. In *Preceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, September 2013.

[16] W. Chen and R. Dömer. Optimized out-of-order parallel discrete event simulation using predictions. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2013.

[17] Y.-T. Chen, J. Cong, and G. Reinman. HC-Sim: A fast and exact L1 cache simulator with scratchpad memory co-simulation support. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2011.

[18] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2007.

[19] B. Chopard, P. Combes, and J. Zory. A conservative approach to systemc parallelization. In *Proceedings of the International Conference on*

Computational Science (ICCS), May 2006.

[20] R. Dömer, W. Chen, and X. Han. Parallel discrete event simulation of transaction level models. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2012.

[21] R. Dömer, W. Chen, X. Han, and A. Gerstlauer. Multi-core parallel simulation of system-level description languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2011.

[22] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski. System-on-chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *The EURASIP Journal on Embedded Systems (EURASIP JES)*, pages 1–13, 2008.

[23] P. Ezudheen, P. Chandran, J. Chandra, B. Simon, and D. Ravi. Parallelizing SystemC kernel for fast hardware simulation on SMP machines. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS)*, June 2009.

[24] R. M. Fujimoto. Parallel discrete event simulation. In *Proceedings of the Winter Simulation Conference (WSC)*, 1989.

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

[26] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, and A. Jerraya. Flexible and executable hardware/software interface modeling for multiprocessor SoC design using SystemC. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2007.

[27] A. Gerstlauer. Host-compiled simulation of multi-core platforms. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, June 2010.

[28] A. Gerstlauer, S. Chakravarty, M. Kathuria, and P. Razaghi. Abstract system-level models for early performance and power exploration. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2012.

[29] A. Gerstlauer, H. Yu, and D. Gajski. RTOS modeling for system level design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2003.

[30] F. Ghenassia. *Transaction level modeling with systemc: TLM concepts and applications for embedded systems*. Springer, 2005.

[31] M. Gligor, N. Fournel, and F. Pétrot. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2009.

[32] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization (WWC)*, December 2001.

[33] M. S. Haque, R. Ragel, A. Ambrose, S. Radhakrishnan, and S. Parameswaran. DIMSim: A rapid two-level cache simulation approach for deadline-based MPSoCs. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2012.

[34] R. Hassan, A. Harris, N. Topham, and A. Efthymiou. Synthetic trace-driven simulation of cache memory. In *Proceedings of International Conference on Advanced Information Networking and Applications Workshops (AINAW)*, May 2007.

[35] HCSim, 2014. `http://users.ece.utexas.edu/~gerstl/releases/`.

[36] Z. He, A. Mok, and C. Peng. Timed RTOS modeling for embedded system design. In *Proceedings of the Real Time and Embedded Technology and Applications Symposium (RTAS)*, March 2005.

[37] J. Heidemann, K. Mills, and S. Kumar. Expanding confidence in network simulations. *IEEE Network: The Magazine of Global Information Exchange*, 15(5):58–63, September 2001.

[38] C. Helmstetter, J. Cornet, B. Galilee, M. Moy, and P. Vivet. Fast and accurate TLM simulations using temporal decoupling for FIFO-based com-

munications. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2013.

[39] Y. Hwang, S. Abdi, and D. Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2008.

[40] Imperas Software Limited, Ltd. `http://www.imperas.com`.

[41] Performance profiler for serial and parallel performance analysis., 2013. `http://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[42] S. Iqbal, Y. Liang, and H. Grahn. ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems. *Computer Architecture Letters*, 9(2):45–48, February 2010.

[43] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2008.

[44] G. KAHN. The Semantics of a Simple Language for Parallel Programming. 1977.

[45] M. Krause, D. Englert, O. Bringmann, and W. Rosenstiel. Combination of instruction set simulation and abstract RTOS model execution for fast

and accurate target software evaluation. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2008.

[46] S. Lauzac, R. G. Melhem, and D. Moss. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 1998.

[47] R. Le Moigne, O. Pasquier, and J.-P. Calvez. A generic RTOS model for real-time systems simulation with SystemC. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, February 2004.

[48] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 17:1217–1229, 1998.

[49] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han. FaCSim: A fast and cycle-accurate architecture simulator for embedded systems. *ACM SIGPLAN Notices*, 43(7):89–100, June 2008.

[50] R. Lee, S. Abdi, D. Regehr, and F. Risacher. System level modeling of real-time embedded software. In *Proceedings of the International Conference on Computer Design (ICCD)*, October 2012.

[51] K.-L. Lin, C.-K. Lo, and R.-S. Tsay. Source-level timing annotation for fast and accurate TLM computation model generation. In *Proceedings of*

the Asia and South Pacific Design Automation Conference (ASPDAC), January 2010.

[52] K. Lu, D. Muller-Gritschneder, and U. Schlichtmann. Accurately timed transaction level models for virtual prototyping at high abstraction level. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2012.

[53] SWARM 0.44 documentation. `http://www.cl.cam.ac.uk/~mwd24/pdh/swarm.html`.

[54] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2), February 2002.

[55] A. Mello, I. Maia, A. Greiner, and F. Pecheux. Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2010.

[56] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen. Source-level timing annotation and simulation for a heterogeneous multiprocessor. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2008.

[57] B. Miramond, E. Huck, F. Verdier, M. E. A. Benkhelifa, B. Granado, M. Aichouch, J.-C. Prvotet, D. Chillet, S. Pillement, T. Lefebvre, and

Y. Oliva. OveRSoC : a framework for the exploration of RTOS for RSoC platforms. *International Journal on Reconfigurable Computing*, 2009(450607):1–18, December 2009.

[58] J. Moreira, F. Klein, A. Baldassin, P. Centoducatte, R. Azevedo, and S. Rigo. Using multiple abstraction levels to speedup an MPSoC virtual platform simulator. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, May 2011.

[59] L. Murillo, J. Eusse, J. Jovic, S. Yakoushkin, R. Leupers, and G. Ascheid. Synchronization for hybrid MPSoC full-system simulation. In *Proceedings of the Design Automation Conference (DAC)*, June 2012.

[60] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[61] D. Nguyen, A. Lenharth, and K. Pingali. Deterministic Galois: On-demand, portable and parameterless. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.

[62] D. Nicol and P. Heidelberger. Parallel execution for serial simulators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 6(3):210–242, July 1996.

[63] SystemC. `http://www.accellera.org`.

[64] Open Virtual Platforms, Co. `http://www.ovpworld.org`.

[65] M. Oyamada, F. Wagner, M. Bonaciu, W. Cesario, and A. Jerraya. Software Performance Estimation in MPSoC Design. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2007.

[66] A. Pedram, D. Craven, and A. Gerstlauer. Modeling Cache Effects at the Transaction Level. In *Proceedings of the International Embedded Systems Symposium (IESS)*, September 2009.

[67] I. Pessoa, A. Mello, A. Greiner, and F. Pecheux. Parallel TLM simulation of MPSoC on SMP workstations: Influence of communication locality. In *Proceedings of the International Conference on Microelectronics (ICM)*, December 2010.

[68] H. Posadas, J. Adamez, P. Sanchez, E. Villar, and F. Blasco. POSIX modeling in SystemC. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2006.

[69] H. Posadas, J. damez, E. Villar, F. Blasco, and F. Escuder. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. *Design Automation for Embedded Systems*, 10(4):209–227, December 2005.

[70] H. Posadas, L. Diaz, and E. Villar. Fast data-cache modeling for native co-simulation. In *Proceeding of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2011.

[71] P. Razaghi and A. Gerstlauer. Host-compiled multicore RTOS simulator for embedded real-time software development. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2011.

[72] P. Razaghi and A. Gerstlauer. Automatic timing granularity adjustment for host-compiled software simulation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2012.

[73] P. Razaghi and A. Gerstlauer. Predictive OS modeling for host-compiled simulation of periodic real-time task sets. *IEEE Embedded Systems Letters (ESL)*, 4(1):5–8, March 2012.

[74] P. Razaghi and A. Gerstlauer. Multi-core cache hierarchy modeling for host-compiled performance simulation. In *Proceedings of the Electronic System Level Synthesis Conference (ESLsyn)*, May 2013.

[75] P. Razaghi and A. Gerstlauer. Host-compiled multi-core system simulation for early real-time performance evaluation. *ACM Transactions on Embedded Computing Systems (TECS)*, To be appeared 2014.

[76] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. `http://sesc.sourceforge.net`.

[77] R. Salimi Khaligh and M. Radetzki. Modeling constructs and kernel for parallel simulation of accuracy adaptive TLMs. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2010.

[78] G. Schirner and R. Dömer. Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2008.

[79] G. Schirner and R. Dömer. Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(1):4:1–4:29, January 2009.

[80] G. Schirner, A. Gerstlauer, and R. Domer. Automatic generation of hardware dependent software for MPSoCs from abstract system specifications. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, March 2008.

[81] G. Schirner, A. Gerstlauer, and R. Dömer. Fast and accurate processor models for efficient MPSoC design. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 15(2):10:1–10:26, March 2010.

[82] J. Schnerr, O. Bringmann, and W. Rosenstiel. Cycle accurate binary translation for simulation acceleration in rapid prototyping of SoCs. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2005.

[83] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *Proceedings of the Design Automation Conference (DAC)*, June 2008.

[84] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann. parSC: Synchronous parallel systemc simulation on multi-core host architectures. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2010.

[85] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, March 2003.

[86] M. Shantharam, P. Raghavan, and M. Kandemir. Hybrid techniques for fast multicore simulation. In *Proceedings of International Euro-Par Conference on Parallel Processing (Euro-Par)*, August 2009.

[87] A. Srinivasan and J. H. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, September 2006.

[88] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate resource conflict simulation for performance analysis of multi-core systems. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2011.

[89] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, and W. Rosenstiel. Hybrid source-level simulation of data caches using abstract cache models. In *proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2012.

[90] M. Streubuhr, J. Gladigau, C. Haubelt, and J. Teich. Efficient approximately-timed performance modeling for architectural exploration of MPSoCs. In *Proceedings of the Forum on Specification Design Languages (FDL)*, September 2009.

[91] Z. Wang and J. Henkel. Accurate source-level simulation of embedded software with respect to compiler optimizations. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*, March 2012.

[92] Z. Wang and A. Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. In *Proceedings of the Design Automation Conference (DAC)*, July 2009.

[93] E. Weingartner, H. vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *Proceedings of the International Conference on Communications (ICC)*, June 2009.

[94] G. Zhou, J. Lu, C.-Y. Wan, M. D. Yarvis, and J. A. Stankovic. *Body Sensor Networks*. MIT Press, Cambridge, MA, 2008.

Vita

Parisa Razaghi is a PhD candidate at the University of Texas at Austin. She received the Bachelor and Master of Science degrees in Computer Engineering from the University of Tehran in 2001 and 2004, respectively. Prior to joining UT Austin in September 2009, she was an Assistant Researcher at the same university. Her research interests include system-level design, and high-level modeling and simulation.

Permanent address: parisa.r@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.