Software Engineering for Embedded Systems Group
Electrical Engineering and Computer Sciences
Technische Universität Berlin, Germany

Compiler and Architecture Design Group
Institute for Computing Systems Architecture
University of Edinburgh, United Kingdom

Design and Implementation of an Efficient Instruction Set Simulator for an Embedded Multi-Core Architecture

Thesis

Submitted in partial fulfillment
of the requirements for the degree of

Dipl.-Ing. der Technischen Informatik

Volker Seeker
Matr.-Nr. 305905

9. May 2011

Thesis advisors:

Prof. Dr. Sabine Glesner
Dr. Björn Franke
Dipl.-Inform. Dirk Tetzlaff

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

$Berlin, May2011$
Volker Seeker

Contents

List of Figures

List of Tables

Abstract

In recent years multi-core processors have become ubiquitous in many domains ranging from embedded systems through general-purpose computing to large-scale data centers. The design of future parallel computers requires fast and observable simulation of target architectures running realistic workloads. Aiming for this goal, a large number of parallel instruction set simulators (ISS) for parallel systems have been developed. With different simulation strategies all those approaches try to bridge the gap between detailed architectural observability and performance. However, up until now state-of-the-art simulation technology was not capable to provide the simulation speed required to efficiently support design space exploration and parallel software development.

This thesis develops the concepts required for efficient multi-core just-in-time (JIT) dynamic binary translation (DBT) and demonstrates its feasibility and high simulation performance through an implementation in the ARCSIM simulator platform. ARCSIM is capable of simulating large-scale multi-core configurations with hundreds or even thousands of cores. It does not rely on prior profiling, instrumentation or compilation and works for all multi-threaded binaries implementing the PTHREAD interface and targeting a state-of-the-art embedded multi-core platform implementing the ARCOMPACT instruction set architecture (ISA).

The simulator has been evaluated against two standard benchmark suites, EEMBC MULTIBENCH and SPLASH-2, and is able to achieve speed ups up to 25,307 million instructions per second (MIPS) on a 32-core `x86` host machine for as many as 2048 simulated cores while maintaining near-optimal scalability. The results demonstrate that ARCSIM is even able to outperform an FPGA architecture used as a physical reference system on a per core basis.

Parts of this thesis have been published as a paper at the IC-SAMOS conference 2011 in Greece:

O.Almer, I.Böhm, T.Edler von Koch, B.Franke, S.Kyle, V.Seeker, C.Thompson and N.Topham: Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation [2011] To appear in Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS'11), Samos, Greece, July 19-22, 2011

Überblick

In den vergangenen Jahren haben sich Mehrkernprozessoren in nahezu allen Gebieten der Computertechnik durchgesetzt. Man findet sie in eingebetteten Systemen, handelsüblichen Personalcomputern und auch groß angelegten Rechenzentren. Um zukünftige parallel arbeitende Systeme effizient designen zu könnnen, benötigt man schnelle und detaillierte Simulationstechnologie, die in der Lage ist realistische Anwendungsaufgaben auszuführen. Mit diesem Hintergrund wurden viele Befehlssatzsimulatoren entwickelt, die sowohl parallele Architekturen modellieren, als auch selbst während der Ausführung das volle Potenzial bereits vorhandener paralleler Strukturen nutzen können. Mit dem Ziel einen Mittelweg zu finden zwischen einer detaillierten Darstellung des Mikroprozessors und einer hohen Simulationsgeschwindigkeit, kommen die unterschiedlichsten Strategien zur Anwendung. Bis jetzt jedoch, ist es keinem System möglich genügend Geschwindigkeit zu liefern um eine effiziente Entwicklung von Mehrkernprozessoren und paralleler Software zu unterstützen.

Diese Diplomarbeit zielt darauf ab Konzepte zu entwickeln, die benötigt werden, um eine effiziente just-in-time dynamic binary translation Technologie für Mehrkernsysteme zu unterstützen. Die Durchführbarkeit des Vorhabens wird anhand der Implementation der genannten Technologie in der ARCSIM Simulator Platform demonstriert. ARCSIM ist in der Lage groß angelegte Mehrkernkonfigurationen mit hunderten oder gar tausenden von Kernen zu simulieren. Dazu sind weder vorangegangenes Profiling, Instrumentierung oder statische Kompilierung notwendig. Der Simulator kann sämtliche Binaries ausführen, die zur Nutzung mehrerer Threads das PTHREAD Interface implementieren und für eingebettete state-of-the-art Mehrkernplattformen kompilieren, die die ARCOMPACT Befehlssatzarchitektur abbilden.

Der Simulator wurde mittels zweier standard Benchmarksammlungen evaluiert, EEMBC MULTIBENCH und SPLASH-2, und erreichte dabei Beschleunigungen von bis zu 25,307 MIPS auf einem 32-Kern `x86` Rechner als Ausführungsplatform. Dabei war es möglich Konfigurationen mit bis zu 2048 Prozessorkernen zu simulieren und gleichzeitig geradezu optimale Skalierbarkeit zu erhalten. Die Ergebnisse demonstrieren, dass es ARCSIM sogar möglich ist eine FPGA Referenzarchitektur zu übertreffen.

Teile dieser Arbeit wurden als Paper auf der IC-SAMOS Konferenz 2011 in Griechenland veröffentlicht:

O.Almer, I.Böhm, T.Edler von Koch, B.Franke, S.Kyle, V.Seeker, C.Thompson and N.Topham: Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation [2011] To appear in Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS'11), Samos, Greece, July 19-22, 2011

Acknowledgements

First of all, I would like to thank my German thesis advisors Prof. Dr. Sabine Glesner and Dirk Tetzlaff for their strong support and valuable advice. Not only did they make it possible for me to spend four month in Edinburgh, where I had a great time working for my diploma thesis at the Institute for Computing Systems Architecture, they also managed to arouse my interest in embedded systems.

I would also like to thank my Scottish thesis advisor Dr. Björn Franke, who has supported me throughout my thesis with patience and knowledge whilst allowing me the room to work in my own way. His door was always open and even when I was back in Germany his support and advice stayed consistent. I really enjoyed working with him and learned more than I had ever expected.

Many thanks to the whole PASTA research group from the University of Edinburgh: Oscar Almer, Igor Böhm, Tobias Edler von Koch, Stephen Kyle, Chris Thompson and Nigel Topham. It was a great pleasure for me to work as part of this team, which always had a wonderful atmosphere and brilliant team spirit.

Finally, special thanks goes to my fiancé Luise Steltzer. She helped me a lot with understanding and care when I was working much and did not have time for other things.

Chapter 1

Introduction

Programmable processors are at the heart of many electronic systems and can be found in application domains ranging from embedded systems through general-purpose computing to large-scale data centers. Steve Leibson, an experienced design engineer, engineering manager and technology journalist, said at the Embedded Processor Forum Conference 2001 that more than 90% of all processors are used in the embedded sector. Also MICROSOFT founder and former chief software architect Bill Gates refers to that fact [1]. Hence, a lot of embedded program code is being developed apart from general-purpose software (including desktop applications and high-performance computing).

In an efficient software and hardware development process, the simulation of a processor is a vital step supporting design space exploration, debugging and testing. Short turn-around times between software and hardware developers are possible, if the costly and time consuming production - which can easily take several weeks or even months - of an expensive prototype can be avoided in order to test your system. Using simulation techniques time-to-market can be significantly reduced as software programmers can start working, while hardware development is still in the prototyping stage.

This is where instruction set simulation comes into play. The instruction set architecture (ISA) is the boundary between software and processor hardware. It is the last layer visible to the programmer or compiler writer before the actual electronic system. An instruction set simulator (ISS) substitutes the complete hardware layer beneath the ISA boundary and allows simulation of a target architecture on a host system[1] with different characteristics and possibly architecture. In contrast

[1]The system being simulated is called the *target* and the existing computer used to perform the simulation is called the *host*.

to real hardware, an ISS can provide detailed architectural and micro-architectural observability allowing observation of pipeline stages, caches and memory banks of the simulated target architecture as instructions are processed. This feature is crucial for debugging and testing.

In recent years a large number of ISS were developed, focusing on different simulation techniques to bridge the gap between detailed architectural observability with *cycle-accurate* simulation and performance with pure *functional* simulation. This problem is particularly challenging for upcoming multi-core architectures in embedded systems. Early approaches like RSIM [2] or SIMPLESCALAR [3] implementations are simulating multi-core targets in a sequential manner, not fully utilizing available multi-core host systems. Later simulation tools like PARALLEL EMBRA [4] or GRAPHITE [5] are able to achieve that by simulating multi-core systems on a multi-core host or even distributing the target architecture onto multiple machines. Despite this, simulation of multi-core targets is still either slow or does not scale well for architectures with a large number of cores.

This thesis introduces ARCSIM, a target-adaptable instruction set simulator with extensive support for the ARCOMPACT ISA [6]. It is a full-system simulator (see 3.2), implementing the processor, its memory sub-system including MMU, and sufficient interrupt-driven peripherals like a screen or terminal I/O. This simulator uses parallel just-in-time dynamic binary translation (JIT DBT) techniques to translate frequently executed target program regions, also called traces, in order to accelerate simulation. At the same time ARCSIM is capable of augmenting JIT generated code with a detailed cycle-accurate processor model, without prior profiling, instrumentation or compilation. Using this techniques ARCSIM is able to even exceed the native execution performance of speed-optimized silicon implementations of single-core target architectures [7]. However, due to its inherent complexity, up until now JIT DBT technology has not been available for multi-core simulation. To efficiently support hardware and software research with a simulator that is capable of simulating a system with hundreds or even thousands of cores, a rate of 1000 to 10,000 MIPS is necessary [8]. The thesis presents how the multi-core version of ARCSIM developed in this study scales up to 2048 target processors whilst exhibiting minimal and near constant overhead.

1.1 Contributions

The main contributions are twofold: The first part concerns software design aspects which were applied in order to provide a scalable multi-core instruction set simulation methodology by extending established single-core JIT DBT approaches to effectively exploit the available hardware parallelism of the simulation host. This

is done by creating a thread for each simulated core which enables the simulator to execute the target application in parallel, whereas a communication system together with a controlling main thread is responsible for synchronization tasks.

The second part focuses on various optimizations maintaining high performance and detailed observability utilizing JIT translation mechanics. The key idea is that each processor thread feeds work items for native code translation to a parallel JIT compilation task farm shared among all CPU threads. Combined with private first-level caches and a shared second level cache for recently translated and executed native code, detection and elimination of duplicate work items in the translation work queue and an efficient low-level implementation for atomic exchange operations, a highly scalable multi-core simulator was constructed that provides faster-than-FPGA simulation speeds and scales favorably up to 2048 cores.

The following list summarizes the main contributions:

- The methodology to extend a single-core simulator to a scalable multi-core ISS includes:
  - Splitting the single threaded main loop into system and processor threads responsible for multiple cores
  - Introducing an event communication system to control running processor threads
  - Synchronizing shared resources and making them available for multiple simultaneously running cores
- Optimizations of the JIT DBT translation mechanic for a parallel multi-core simulator include:
  - A centralized parallel task farm for translating frequently used target code traces
  - A multi-level cache hierarchy for JIT-compiled code
  - Detection and Elimination of duplicate work items in the translation work queue
  - An efficient low-level implementation for atomic exchange operations

The simulation methodology developed in this thesis has been evaluated against the industry-standard EEMBC MULTIBENCH and SPLASH-2 benchmark suites. The functional ISS models homogeneous multi-core configurations composed of ENCORE (see 3.3.2) cores, which implement the ARCOMPACT ISA. On a 32-core `x86` host machine simulation rates up to 25,307 MIPS for as many as 2048 target processors are demonstrated. Across all benchmarks ARCSIM's JIT DBT simulation approach achieves an average simulation performance of 11,797 MIPS (for 64 simulated cores) and outperforms an equivalent system implemented in FPGA.

1.2 Overview

The remainder of this work is organized as follows: Chapter 2 discusses research work related to this thesis. This is followed by a detailed background description in Chapter 3, where an overview is presented about the functionality of an instruction set architecture, general instruction set simulation approaches, the ARCSIM simulator project, common `pthread` programming techniques and the benchmarks used for this study. The methodology in Chapter 4 covers in detail the proposed parallel simulation technology. It is followed by a presentation of the results for the empirical evaluation in Chapter 5. Finally, Chapter 6 summarizes the thesis and gives an outlook to future work.

Chapter 2

Related Work

The instruction set simulation of single-core systems already has a wide variety of implementation approaches, of which the most important ones are described in detail in Section 3.2.2. Therefore, also a lot of related research work exists concerning instruction set simulation for multiple cores. To not exceed the focus of this thesis, this chapter only presents the most relevant approaches in software and FPGA-based multi-core simulation.

The related work section starts with an overview about studies concerning software based approaches which are capable of simulating or emulating multi-processor systems in a sequential manner. The second part covers parallel software based simulation of parallel architectures, and the last part introduces the most relevant and recent work to FPGA-based instruction set simulation.

2.1 Sequential Simulation of Parallel Systems

Early approaches to efficient simulation of multi-core systems date back to the 1990s. However, those simulators/emulators did not run multiple cores in parallel but executed them sequentially on the host machine. The following explains some of those works in detail:

RPPT [9] (Rice Parallel Processing Testbed) is a simulation system based on a technique called *execution-driven simulation*. The testbed was designed to investigate performance issues concerning the execution of concurrent programs on parallel computing systems and to evaluate the accuracy and efficiency of

this technique.

In execution-driven simulation, execution of the program and the simulation model for the architecture are interleaved. A preceding profiling of the target source calculates instruction counts and timings for each basic block. This information is then added to the source program in the form of a few instructions on each basic block boundary that increment the cycle count by the amount of the estimate for that block. During the simulation, the modified target application is then executed and statistics are updated automatically.

Multiple processors are simulated sequentially. The target program executes until an interaction point occurs, where the current process needs synchronization with a different one. The current process is then delayed and another processor continues with its work.

RSIM [2] is a simulator for shared-memory multi-core systems based on the RPPT, whereas its focus lies on modeling instruction-level parallelism (ILP). Processor architectures aggressively exploiting ILP have the potential to reduce memory read stalls by overlapping read latency with other operations. Studies with RSIM showed, that simulation of ILP is crucial to get proper simulation results for architectures using this technique.

SIMPLESCALAR [3] introduces a multi-core simulator built with the the SIMPLESCALAR tool set, which allows users to build interpretive simulators (see 3.2.2) for different architectures, also called SIMPLESCALAR modules. SYSTEMC is used to combine different architecture simulators in order to integrate heterogeneous processor architecture models with only slight modifications. Different SIMPLESCALAR modules are then executed in a round robin fashion, whereas the shared-memory is used as the inter-core communication media.

QEMU [10] is a full-system machine emulator relying on dynamic binary translation (see 3.2.2). It is capable of emulating multi-core targets, but uses a single thread to emulate all the virtual CPUs and hardware. However, an extension called COREEMU was implemented [11], which is able to run parallel systems in a parallel way. Each core uses a separate instance of the QEMU binary translation engine, with a thin library layer to handle the inter-core and device communication and synchronization.

Although those instruction set simulators together with SIMOS [12] and SIMICS [13] are capable of simulating/emulating multi-core architectures, they do not work in parallel[1]. Therefore, those systems force a single host core to perform the work of many target cores. It is hardly possible to achieve higher speed-ups that way in comparison to a simulator utilizing all cores of the host system as ARCSIM does.

[1] COREEMU handles multi-core architectures in a parallel way. But in contrast to ARCSIM, it emulates the target and therefore provides no precise architectural observability.

2.2 Software Based Parallel Simulation of Parallel Systems

A large number of parallel simulators for parallel target architectures have been developed over the last two decades: SIMFLEX [14], GEMS [15], COTSON [16], BIGSIM [17], FASTMP [18], SLACKSIM [19], PCASIM [20], Wisconsin Wind Tunnel (WWT) [21], Wisconsin Wind Tunnel II (WWT II) [22], and those described by Chidester and George [23], and Penry et al. [24]. Some of them also use dynamic binary translation techniques, simulate shared memory systems or are able to run `pthread` target applications without source code modifications. The following lists in detail examples of software based parallel multi-core simulators and discusses what they have in common with ARCSIM and where ARCSIM surpasses them or focuses on a different aspect.

SIMFLEX [14], GEMS [15] and COTSON [16] are all approaches using a preset simulation framework to simulate the functional behavior of multi-core architectures and add their own components to model the timing aspect. Whereas GEMS and SIMFLEX are based on the full-system simulator SIMICS, COTSON uses AMD's SIMNOW! simulator.

GEMS implements a multiprocessor memory system, which models caches and memory banks with corresponding controllers, and an interconnection network, which is able to provide detailed timings for inter-component communications. A processor timing model runs along with SIMICS' functional simulation and drives the functional execution of individual instructions by providing proper timings. However, the effort to get precise architectural observability results in low performance. SIMFLEX is a lot faster since it uses SMARTS' statistical sampling approach (see 3.2.2) as a timing model extension, but therefore does not observe the entire architectural state. COTSON is also relying on statistical sampling to provide timing information alongside with functional simulation by AMD's SIMNOW! simulator. It therefore suffers from the same trade-off between architectural observability and speed as SIMFLEX does.

FASTMP [18], BIGSIM [17], WWT [21] and WWT II [22] are parallel multi-core simulators, which in contrast to ARCSIM do not fully support a shared memory system or only with prior modifications of the target application.

FASTMP increases the simulation performance of multi-core architectures by using statistical methods. A single simulation run only collects detailed traffic information for a subset of the simulated cores, whereas the other remaining cores' behavior is estimated from that collected data set. This approach, however, assumes workloads which are homogeneous for all participating cores and supports only distributed memory target architectures. The same limitations affect BIGSIM, which is a parallel simulator that is capable of predicting

performance for large multi-core systems with tens of thousands of processors.

The Wisconsin Wind Tunnel (WWT) is one of the earliest parallel simulators. It runs parallel shared-memory target programs on a non shared-memory parallel host. WWT, like its successor WWT II, uses a direct execution mechanism where the majority of an application is executed on native hardware, with the simulator paying special attention only to those events that do not match the target architecture. The WWT runs on a THINKING MACHINE CM-5 and traps on each cache miss, which is simulated in a shared virtual memory. However, WWT is restricted to the CM-5 machine and requires target applications to use an explicit interface for shared memory. The Wisconsin Wind Tunnel II aims at creating a portable multi-core simulator. WWT II only simulates the target memory system and modifies the target source code to get remaining simulation statistics. Additional code is added to each basic block of the target program that updates the target execution time and target code modifications are necessary to explicitly allocate shared memory blocks. ARCSIM models a complete target system including peripheral devices, implements a transparent shared memory model and does not require any target code changes.

PARALLEL EMBRA [4] is part of the PARALLEL SIMOS complete machine simulator. Like ARCSIM, it is a parallel, functional simulator using a binary translation mechanic, which supports complete machine simulation of shared-memory multiprocessors. PARALLEL EMBRA takes an aggressive approach to parallel simulation, where nearly every module of the simulator can operate in parallel. The simulated target machine is divided into pieces which the simulator executes in parallel, at the granularity of the underlying host hardware. For example, a 16-processor simulation with 2 GB of memory could be divided across 4 real hardware nodes by assigning 4 simulated processors and 512 MB of simulated memory to each node. Despite this high level of parallelization, PARALLEL EMBRA relies on the underlying shared memory system for synchronization and runs without cycle synchronization across virtual processors. As a result it executes workloads non-deterministically but preserves overall correctness, since memory events cannot be reordered across virtual processors. While PARALLEL EMBRA shares its use of binary translation with ARCSIM it lacks its scalability and parallel JIT translation facility.

P-MAMBO [25] and MALSIM [26] are parallel multi-core simulators which result, like ARCSIM, from parallelizing sequential full-system software simulators. P-MAMBO is based on MAMBO, IBM's full-system simulator which models POWERPC systems and MALSIM extends the SIMPLESCALAR tool kit mentioned above. Whereas P-MAMBO aims to produce a fast functional simulator by extending a binary translation based emulation mode, MALSIM utilizes a

multi-core host with a funtional `pthread` simulation model to support multi-programmed and multi-threaded workloads. Published results, however, include a speed-up of up to 3.8 for a 4-way parallel simulation with P-MAMBO and MALSIM has only been evaluated for workloads comprising up to 16 threads. Despite some conceptual similarities with these works this thesis aims at larger multi-core configurations where scalability is a major concern.

A DBT based simulator [27] for the general purpose tiled processor RAW was developed in Cambridge. This work aims at exploiting parallelism in a tiled processor to accelerate the execution of a single-threaded `x86` application. It focuses on three mechanisms to reach that goal: Like ARCSIM, this simulator uses parallel dynamic binary translation but instead of hot spot optimization, a speculative algorithm is applied which translates a predicted program execution path in advance. The second mechanism exploits the underlying hardware to implement pipelining which increases the throughput of needed resources like memory or caches. Static and dynamic virtual architecture reconfiguration is the third technique. The emulated virtual architecture is configured statically for the executed binary and reconfigured at runtime depending on the current application phase. This means that, for example, in the applications start up phase more processor tiles are dedicated for translation and in the end more for the memory system. However, this work does not attempt to simulate a multi-core target platform.

ARMN [28] is a multiprocessor cycle-accurate simulator which can simulate a cluster of homogeneous ARM processor cores with support for various interconnect topologies, such as mesh, torus and star shape. Whilst this provides flexibility, the performance of ARMN is very low (approx. $10k$ instructions per second) and, thus, its suitability for both HW design space exploration and SW development is limited.

GRAPHITE [5] is a distributed parallel simulator for tiled multi-core architectures that combines direct execution, seamless multi-core and multi-machine distribution and lax synchronization. It also aims to simulate larger multi-core configurations and has been demonstrated to simulate target architectures containing up to 1024 cores on ten 8-core servers. GRAPHITE's multi-core simulation infrastructure is most relevant for this thesis.

With its threading infrastructure GRAPHITE is capable of accelerating simulations by distributing them across multiple commodity Linux machines. For a multi-threaded target application it maintains the illusion that all of the threads are running in a single process with a single shared address space. This allows the simulator to run off-the-shelf `pthread` applications with no source code modification. Application threads are executed under a dynamic binary instrumentor (currently PIN) which rewrites instructions to generate events at key points. These events cause traps into GRAPHITE's backend which contains the compute core, memory, and network modeling modules.

Instructions and events from the core, network and memory subsystem functional models are passed to analytical timing models that update individual local clocks in each core in order to maintain proper timings for the simulated architecture. However, to reduce the time wasted on synchronization, GRAPHITE does not strictly enforce the ordering of all events in the system as they would have in the simulated target.

In contrast, the ARCSIM simulator uses DBT to implement any ISA (currently ARCOMPACT), which can also be different from the target system's ISA. In addition, the primary design goal of this study's simulator has been highest simulation throughput as showcased by the parallel JIT task farm contained within ARCSIM. As a result speed-ups over native execution were achieved for many multi-core configurations, whereas GRAPHITE suffers up to $4007\times$ slowdown.

A simulator proposed by the HP LABS [29] simulates large shared-memory multi-core configurations on a single host. The basic idea is to use thread-level parallelism in the software system and translate it into core-level parallelism in the simulated world.

An existing full-system simulator is first augmented to identify and separate the instruction streams belonging to the different software threads. Then, the simulator dynamically maps each instruction flow to the corresponding core of the target multi-core architecture. The framework detects necessary thread synchronization spots and implements the corresponding semantics by properly delaying the application threads that must wait. Since the buffering necessary to synchronize separated thread instruction streams can easily consume large amounts of memory, an instruction compression technique and a scheduling feedback mechanism are applied. When the local buffer of a thread runs out of instructions its priority is raised. It is filled with new instructions and the waiting period of related threads is decreased.

This approach treats the functional simulator as a monolithic block, thus requires an intermediate step for de-interleaving instructions belonging to different application threads. ARCSIM does not require this costly preprocessing step, but its functional simulator explicitly maintains parallel threads for the CPUs of the target system.

2.3 FPGA-based Parallel Simulation of Parallel Systems

Section 3.2.2 will describe FPGA-based instruction set simulation in detail, but as a quick summary: one can generally distinguish between two approaches to FPGA-based multi-core simulation. The first approach utilizes FPGA technology for *rapid prototyping*, but still relies on a detailed implementation of the target platform,

whereas the second approach seeks to speed up performance modeling through the combined implementation of a functional simulator and a timing model on the FPGA fabric. In the following, examples for this latter approach are discussed.

RAMP GOLD [30,31] is a state-of-the-art FPGA-based "many-core simulator" supporting up to 64 cores. The structure of the simulator is partitioned into a functional and a timing model, whereas both parts are synthesized onto the FPGA board. In functional-only mode, RAMP GOLD achieves a full simulator throughput of up to 100 MIPS when the number of target cores can cover the functional pipeline depth of 16. For fewer target cores (and non-synthetic workloads), the fraction of peak performance achieved is proportionally lower. In comparison to the 25,307 MIPS peak performance of ARCSIM's software-only simulation approach (based on an ISA of comparable complexity and similar functional-only simulation) the performance of the FPGA architecture simulation is more than disappointing. Other approaches with RAMP [32] had similar results.

FAST [33,34] and PROTOFLEX [35] are also FPGA accelerated simulators which aim at partitioning simulator functionalities. Like RAMP GOLD, FAST partitions timing and functional mechanics into two different models. However, only the timing model is running on an FPGA platform whereas the functional model is implemented in software. PROTOFLEX is not a specific instance of simulation infrastructure but a set of practical approaches for developing FPGA-accelerated simulators. An actual instance is the BLUESPARC simulator that incorporates the two key concepts proposed by PROTOFLEX. It uses *hybrid simulation* like FAST, where parts of the architecture are hardware and others software, but also goes on step further and uses *transplanting*. Transplanting is a technique where only frequent behavior of a component is synthesized for the FPGA and remaining functions are rerouted to a software algorithm. For example, an FPGA CPU component would only implement the subset of the most frequently encountered instructions. The remaining ones are executed by a software model. Since both simulators also achieve only speed-ups similar to the FPGA approaches mentioned above or even less, they can not compete with ARCSIM in terms of multi-core simulation performance.

Chapter 3

Background

In Chapter 3, the first Section 3.1 will give an overview of how a processor interacts with executed software using an instruction set. Section 3.2 explains how instruction set benefits are used to build different types of processor simulators that help developers designing hard- and software. Afterwards, Section 3.3 introduces the ARCSIM instruction set simulator used in this study and Section 3.4 and 3.5 give an overview about common multi-core programming techniques as well as executed benchmarks.

3.1 Instruction Set Architecture

Since the central part of an embedded system is its processor, how is it possible for an embedded application to access the processor's functionality? How does software tell the CPU what to do? The boundary between software and processor hardware is the *instruction set architecture*. It is the last layer visible to the programmer or compiler writer before the actual electronic system. This layer provides a set of instructions, which can be interpreted by the CPU hardware and therefore directly address the underlying micro-architectural pipeline. A black bar in the middle of Figure 3.1 indicates the ISA, whereas the upper blue part holds different software layers and the lower red part defines the hardware layer.

The lower red layers beneath the ISA are called the micro-architecture of the processor. It contains the Circuit Design Layer at the bottom, which consists of electrical wires, transistors, and resistors combined into flip-flops. Those components are arranged and composed to blocks of specific electronic circuitry like adders, multiplexers, counters, registers, or ALUs. Together they form the Digital Design

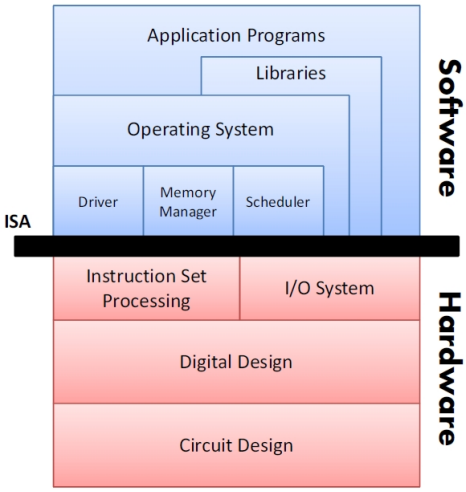


Figure 3.1: Instruction set architecture as boundary between hard- and software.

Layer, which defines the processor's underlying logical structure. Touching the ISA boundary is a layer for Instruction Set Processing and IO-devices. All incoming instructions are decoded and sequenced to commands in some kind of corresponding Register Transfer Layer (RTL), which directly addresses the physical micro-architecture of the processor or attached devices.

The ISA as a programming interface hides all micro-architectural information about the processor from the software developer. Since developing applications on Instruction Set level is complicated and inefficient, this level is hidden again by high level languages like C or C++, libraries and functionalities provided by the operating system, as seen in the top half of Figure 3.1. However, compiler writers, for example, need to use the ISA directly. This information hiding mechanism simplifies the software development process and makes micro-architecture interchangeable.

Hence, a micro-architecture is a specific implementation of an ISA, which is then referred to as the processor's architecture. Various ways of implementing an instruction set give different trade-offs between cost, performance, power consumption, size, etc. Processors can differ in their micro-architecture but still have the same architecture. A good example would be INTEL and AMD. Both processor types implement the `x86` architecture but have totally different underlying micro-architecture.

3.1.1 ISA Classification

The most basic differentiation between ISAs is the type of internal storage used for instruction operands in a processor. Typically the following three major types are used [36]:

Stack Architecture All instruction operands are implicitly at the top one or two positions of a stack located in the memory. Special instructions like `PUSH A` or `POP B` are used to load operands to the stack and store them back into memory. This architecture benefits from short instructions and few options, which simplifies compiler writing. Though a major disadvantage is that a lot of memory accesses are required to get operands to the stack. The stack therefore becomes a huge bottleneck.

Accumulator Architecture One of the instruction operands is implicitly the accumulator. The accumulator is always used to store the calculation result. This architecture is easier to implement than the stack architecture and has also small instructions, however it still needs a lot of memory accesses.

General Purpose Register (GPR) Architecture Instruction operands are stored either in memory or in registers. A register can be used for anything like holding operands or temporary values. GPR is the most commonly used architecture these days because of two major advantages. At first, register access is much faster than memory access and using registers to hold variables reduces memory traffic additionally. The second big advantage is the possibility to efficiently use registers for compiler work. However, this flexibility and big amount of options makes compiler design quite complicated and results in long instructions.

Table 3.1 shows an example for the mentioned architecture types defining a simple ADD operation: **A + B = C**

Stack	Accumulator	GPR (2-op)	GPR (3-op)
`PUSH A`	`LOAD A`	`LOAD R1,A`	`ADD A,B,C`
`PUSH B`	`ADD B`	`ADD R1,B`	
`ADD`	`STORE C`	`STORE R1,C`	
`POP C`			

Table 3.1: ADD operation displayed for different operand storage architectures.

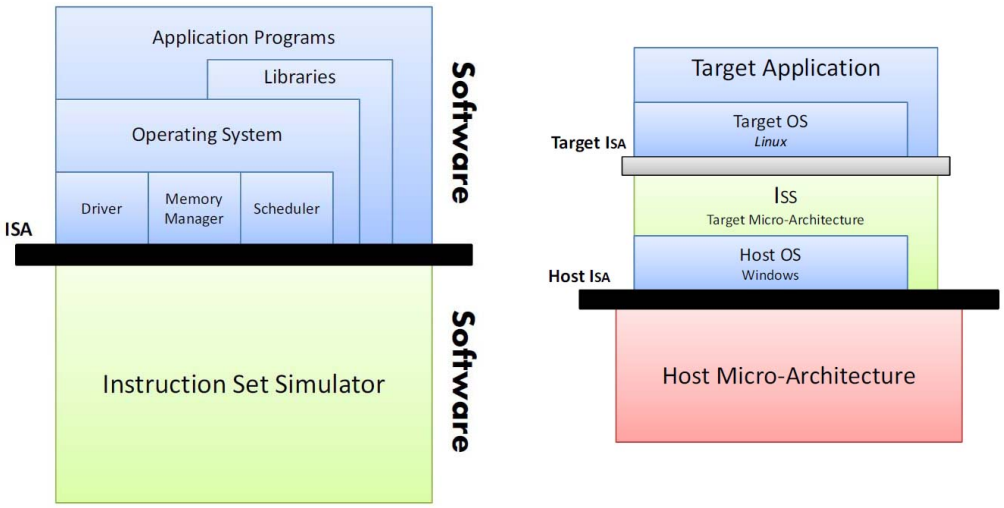
As said before, most modern CPUs like PENTIUM, MIPS, SPARC, ALPHA or POWERPC implement the GPR type. This architecture can be further classified into number of operands per ALU instruction, which varies between two and three, and

number of operands referring directly to memory locations, which goes from zero up to three references. The more operands an ALU instructions has, the longer instructions get and the more memory needs to be used. Many memory references in a single instruction reduce the total instruction count in a program but result in more memory access during program execution. Therefore, architectures with three possible memory references per instructions are not common today.

Available GPR ISAs can be separated into two main groups: *Complex instruction set computing* (CISC) and *Reduced instruction set computing* (RISC). CISC aims for complex but powerful instructions, which can vary in size and operand count. This is supposed to produce compact and fast code. However, decoding CISC instructions on the processor needs a lot of work and is mostly slow. It is done by micro-code running on the processor. A CISC machine does not have many registers, since register access is mostly encoded into the instructions.

The RISC strategy aims for a reduced instruction set with a standard length and simple operations. This results in longer code, but faster decoding times, since RISC instruction decoding is mostly hard wired. Simple instructions make this architecture flexible enough to use a large amount of registers, which reduces slow memory access. The most significant advantage of RISC is an efficient usage of the processor's instruction pipeline. Every instruction needs multiple stages to be executed. They are basically instruction fetch from memory, instruction decode, instruction execute and writing back the results. A pipeline can handle multiple instructions at the same time, where an instruction can start with its first stage while other instructions are still in a different one. Using RISC instructions with a standard length enables efficient and short pipelines, whereas executing CISC instructions with various sizes efficiently in a pipeline is a lot more complex. In addition, a short pipeline is only an advantage if instruction execution can be done in a few cycles. This is easy for simple RISC instructions but not always possible for powerful but complex CISC ones.

A third group is *very long instruction word* architecture (VLIW). This strategy takes advantage of instruction level parallelism. The compiler analyzes the program for instructions that can be executed in parallel. These are then grouped and executed at the same time. The group size depends on available execution components. However, if program operations have interdependencies, it can happen that a single instruction needs to be executed by itself, which reduces efficiency.



(a) Target simulation... **(b)** ...on a host system

Figure 3.2: ISS as simulation system for a target application on a different host system.

3.2 Instruction Set Simulator

Simulation is a method used in scientific and engineering areas that is based on the idea of building a model of a system in order to perform experiments on this model. Provided that the model is reasonably similar to the system being modeled, the results from the simulation experiments can be used to predict the behavior of the real system.

The previous section described how the ISA hides the system hardware from the programmer. This provides the possibility of interchanging executing hardware with other hardware or even against software. As long as instructions used by the running application are interpreted correctly and proper results are delivered, no difference will be recognized. This is where instruction set simulation comes into play. A pure ISS only refers to the simulation of a micro-processor. In order to take full advantage of all simulation benefits, additional hardware simulation is necessary. A *full-system simulator* encompasses all parts of a computer system. Including the processor core, its peripheral devices, memories, and network connections [37]. This enables the simulation of realistic work loads which fully utilize a computer system. In the following text ISS will refer to a full-system simulator.

Figure 3.2a shows how an ISS substitutes the complete hardware layer beneath the ISA boundary. In order to run the system displayed in Figure 3.2a, however, additional host hardware executing simulator and target application is required.

Figure 3.2b depicts relations between target application, ISS and host system. *Target* always refers to the system being simulated, whereas *Host* indicates the hardware the simulation is running on. Consider the following example for using hardware simulation:

Anti-lock braking system The target could be an embedded system, for example, the controller of an anti-lock braking system in a car. It consists of a microprocessor, wheel speed sensors and hydraulic valves to access the brakes. The target application (blue part at the top of Figure 3.2b) in our case is monitoring the rotational speed of each wheel and actuates the valves, if it detects a wheel rotating significantly slower than the others.

For developing corresponding control software it would be very complicated to drive around in a car all time in order to get proper test signals. Even a test board with connected peripherals could be to complex in the early states of development. Hence, processor and IO-devices are simulated by an ISS (green part in the middle of Figure 3.2b). The ISS is running on a common computer terminal in the lab and executes the target application. The computer terminal is called the host system (red part at the bottom of Figure 3.2b).

This short example shows how it is possible to create a virtual simulation environment. In the case of micro-processors, using an ISS enables developers to run an application written for a specific target ISA on a host system supporting a totally different one.

Working with a software simulation instead of a real hardware system has a variety of advantages [37]. Section 3.2.1 will point out these advantages for the example of an ISS. There are different types of ISS, which provide models of the target micro-architecture of varying degrees of detail and abstraction. But as mentioned before, no problems will arise as long as the target application does not recognize a difference. Section 3.2.2 classifies those types in more detail.

3.2.1 ISS Advantages

ISS provide detailed *architectural observability*. Developers can observe machine state, registers and memory at any time and collect statistical information such as cycle and instruction counts or cache accesses. This observability enables early exploration of the architecture design space, which means design errors can be found and corrected quickly without expensive and time consuming overhead of producing a hardware prototype. If the design is worked out and a prototype has been produced, Hardware/Software co-simulation can be used to verify it, using the simulator as a software reference model. This is done by running hardware

(a) S60 Simulator **(b)** IPhone Simulator

Figure 3.3: An example for popular simulators are simulation environments for mobile phone development like Nokias S60 simulator or Apples IPhone simulator.

and software in lock-step mode and evaluating the machine state of both systems against each other. Not only does the hardware development of the processor benefit from virtual prototypes, but software development is also simplified.

Software teams can start early with developing compilers or applications for the new system. If a design issue is found, *faster turn-around times* between hard- and software development can be achieved thanks to quick adaptation of the simulated reference model. Because software development can start while hardware is still in design phase, *time-to-market* speeds up significantly by reducing the overall development time for new micro-processors. High *configurability* and *controllability* help when debugging a system in development. Any machine configuration can be used, unconstrained by available physical hardware, and the execution of the simulation can be controlled arbitrarily, disturbed, stopped and started. It is also possible to *inject faults* on purpose in order to observe the system's behavior concerning error detection and correction methods like interrupts and exceptions.

Once the hardware design is stable and no longer in prototype stage, the simulation model really becomes virtual hardware. Virtual systems have a much higher *availability* than real hardware boards, which simplifies the test and maintenance process. Creating a new processor is now just a matter of copying the setup. There

is no need to procure hardware boards, which sometimes is not even possible anymore when it comes to older computer systems. Two examples of simulators being used in the development process can be seen in Figure 3.3.

As this section shows, ISS have a large number of benefits, which make them popular tools among both embedded hardware and software developers. However, there are also disadvantages. The most significant one is the trade-off between simulation speed and architectural observability. Simulating micro-architecture of a processor on a very low level like register-transfer-level (RTL), for example, is time consuming and sometimes too slow to be used efficiently. Typically those cycle-accurate simulators operate at 20 K - 100 K instructions per second. On the other hand a faster functional simulation, which operates at 200 - 400 million instructions per second [38], does not always provide enough details for proper debugging. The following section depicts different strategies to optimize this trade-off.

3.2.2 ISS Classification

To achieve the best trade-off between speed and architectural observability different simulation strategies have been developed. Though today's simulators focus on a specific strategy, often multiple simulation modes are supported. Instruction set simulation can basically be classified into three different types:

- Interpretive simulation
- Compiled simulation
- FPGA based simulation

Depending on the level of detail being simulated, an ISS can have a timing model that simulates the micro-architectural structures affecting timing and resource arbitration. Hence, a second way of classification would be the following:

- *Cycle-accurate*: simulation with detailed timing model
- *Instruction-accurate*: pure functional simulation without timing model
- *Cycle-approximate*: hybrid functional and timing simulation

Interpretive Simulation Interpretive simulation is a very basic and straight-forward simulation strategy, which is either working close to the actual micro-architecture or on a fast functional level. It simulates single stages corresponding to the instruction set processing layer of the processor, such as instruction *fetch*, instruction *decode* and instruction *execute*. All instructions of the target program along the execution path are fetched from the simulator's memory, decoded by using the opcode and interpreted by a corresponding routine. This routine executes the instruction's functionality and updates a data structure representing the state of the target processor. Registers, memory and the context of IO-devices are updated as instructions commit, thereby maintaining a precise view of the target system. Faithfully processing instruction by instruction makes this strategy quite flexible considering code modifications at runtime. They can result from either self-modifying code or manual changes done at debugging breakpoints.

Even though interpretive simulation is relatively straight-forward to implement, provides good observability and is flexible, it is rather slow. Most interpretive simulators are either written in a common high level language like C. Hence, interpreting a single target instruction results in approximately 10 to 100 host instructions [39]. Performing fetch, decode and execute for all theses additional instructions results in a significant slow-down over native execution. That means the virtual system is much slower than a real hardware simulation would be. Nevertheless, building software simulations is still faster than building hardware boards (see Section 3.2.1).

An example for interpretive simulation is the SIMPLESCALAR [40, 41] simulator. Implemented in 1992 as a first version at the University of Wisconsin, it is now available as version 3.0, providing a flexible infrastructure for building interpretive simulators for different target systems. The SIMPLESCALAR licensing model permits users to extend the SIMPLESCALAR tools. Hence, multiple modifications to the tool set have been developed, for example, an extension called DYNAMIC SIMPLESCALAR (DSS) [42]. DSS simulates Java programs running on a JVM and supports, among other new features, a dynamic just-in-time compilation strategy as described in the next section.

Compiled Simulation The basic idea of compiled simulation is to reduce interpretive runtime overhead by removing statically executed fetch and decode stages [44]. Each target machine instruction is translated to a series of host machine instructions, which represent corresponding simulation methods. These methods manipulate the data structure representing the processor's state, and are therefore the equivalent to the execution stage. Instead of executing fetch and decode for the same code multiple times, translation work is done once; Either *statically* during start-up phase of the simulator or *dynamically* at runtime for repeatedly executed

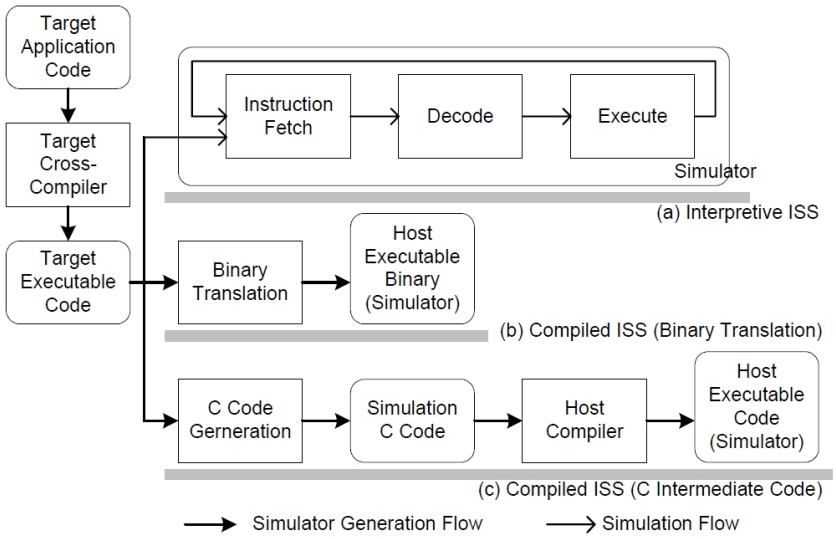


Figure 3.4: Figure 1 (a) shows the behavior of the interpretive ISS. (b) and (c) show the compilation process of the compiled ISS with binary translation and intermediate code. [43]

code portions like loops.

In addition to reducing runtime overhead, redundant code can be eliminated through host compiler optimizations. There are two compilation schemes [45]:

- The target binary is translated directly by replacing target instructions with corresponding host instructions.
- Intermediate code is used by going through high-level language generation and compilation stages.

Figure 3.4 (a) shows an interpretive process, (b) binary translation and (c) uses intermediate code. Binary translation is fast but the translator is more complex and less portable between hosts. Using intermediate code in a high-level language like C is more flexible concerning the host machine. It also benefits from using C compiler optimizations. However, due to longer translation times, losses in simulation speed are possible.

Static Compilation Once a target program translation is done, its simulation is much faster than the interpretive strategy. But there are limitations:

- The static compiled method has a considerable start-up cost due to the translation of the target program. If a large program is used for simulation only once before it needs to be compiled again, interpretive simulation is probably the better choice.
- The static compilation method is less flexible than interpretive simulation. Since the static compiled method assumes that the complete program code is known before the simulation starts, it cannot support dynamic code that is not predictable prior to runtime. For example, external memory code, self-modifying code and dynamic program code provided by operating systems, external devices or dynamically loaded libraries cannot be addressed by a static compiled ISS.

An example of a static compiled ISS is OBSIM introduced in [46] and OBSIM2 for multi-processor simulation introduced in [43]. Those papers also provide a good overview of compiled ISS. OBSIM aims to reduce static compilation drawbacks by providing more flexibility and decreasing start-up cost. Key to this improvement is the usage of target object files in Executable and Linkable Format (ELF) as simulator input instead of binaries.

Dynamic Compilation Dynamic compilation combines interpretive and compiled simulation techniques, maintaining both flexibility and high simulation speeds [38]. This strategy uses statistical methods to detect frequently executed parts of the target programs execution path (also called *traces*). In the beginning all instructions are executed interpretively. If the execution count for a trace reaches a specific threshold, it is called "hot" and translated at runtime to host machine code. Hence, future executions of that code fragment will not be done interpretively anymore but will use the compiled version instead. This is much faster and the simulation speed increases. Compiling hot traces at runtime is typically done by a just-in-time compiler and therefore better known as *just-in-time dynamic binary translation* (JIT DBT). The actual translation scheme, however, is mostly one of the two schemes mentioned above.

JIT DBT does not have a big start-up cost like static compilation, however necessary compilation work is done at runtime. This additional runtime complexity can be somewhat balanced, if it appears only for hot traces and not the whole program. If done properly, this simulation method can also handle dynamic code. Two different situations are possible, if dynamic code appears at runtime:

- The modified code portion has not yet been translated. Nothing happens, the interpretive mode faithfully reads instruction by instruction and is not affected by the modifications.
- The code portion without the modifications has already been translated to host code. If the simulator is able to detect the modified code, the old translation is discarded. The modified code is now interpreted again until it gets hot enough for a second translation.

  An example technique to detect modified code is using the hardware page protection mechanism. All memory pages with the program code are set to read only [47]. If code is being modified, a segmentation fault error would arise. This exception can be caught and an existing translation for the corresponding page can be discarded directly. Another possibility is using Write and Execute flags, which can only be set apart from each other. If a page is being modified, its Write flag is set and the Execute flag erased. Every time the simulator reaches a new page, the Write flag is checked. If it is set, a corresponding translation is discarded and the simulation runs in interpretive mode for that page.

Apart from ARCSIM (see 3.3) another DBT simulator is EMBRA [48]. EMBRA is part of the SIMOS simulation environment and simulates MIPS R3000/R40000 binary code on a Silicon Graphics IRIX machine. Dynamic code is a problem for EMBRA. Once a code portion is translated, it is not interpreted again, therefore changes done later in the translated part will not be recognized. As described in Section 2.2 a parallel version of EMBRA was developed, which is capable of simulating multi-core targets.

FPGA-based Simulation The fastest and most accurate method to test the new hardware design of a micro-chip is using an actual hardware board. The implementation of a software simulator takes a long time to become bug free and can still always be somewhat different from the actual hardware implementation. However, building a hardware prototype is expensive and can take up to several months. Hence, this evaluation technique is mainly used if the hardware design is almost finished, and even then only rarely.

A solution to benefit from fast and accurate hardware execution, yet maintaining an acceptable amount of development effort, lies in using field-programmable gate arrays (FPGA). FPGAs contain programmable logic components called "logic blocks" and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together". In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or complete blocks of memory. Developers can design hardware using a hardware description language (HDL) like VHDL or VERILOG and

load the hardware description to an FPGA. Designing hardware is mostly still more time consuming and complex than software development, but if done properly, simulation speed-ups and accurate modeling of the target micro-architecture can be worth the effort.

In general one can distinguish between two approaches to FPGA-based simulation:

- The first approach essentially utilizes FPGA technology for rapid prototyping, but still relies on a detailed implementation of the target platform [49, 50]. These systems are fast and accurate but still transparent. On the other hand the development effort is high and a final architecture is difficult to modify since a complete hardware description is necessary.

- The second approach seeks to speed up performance modeling through a combined implementation of a functional simulator and a timing model on the FPGA fabric. This can also mean that some parts of the simulator are built in software, and some on FPGAs [8,51]. A partitioned simulation approach is fast, accurate and requires less development effort than designing everything with an HDL, if some parts are implemented in software. Incorrect partitioning, however, could result in lower performance than a pure software simulator. For example, communication between hard- and software parts can become a bottleneck.

Timing Model A *cycle-accurate* ISS operates on the RTL and provides the best architectural observability including precise cycle counts for executed instructions. In order to provide statistical information at that level of detail, a precise model of the time spent in different micro-architectural stages is necessary. Modeling all register-to-register transitions in a cycle-by-cycle fashion is very computationally-intensive since every instruction is moved through every pipeline stage. Therefore, some simulators drop a precise *timing model* and use only a fast *functional model* instead, which operates on instruction set level and abstracts away all hardware implementation details. Cycle-accurate simulators typically operate at 20K - 100K instructions per second, compared with pure functional dynamic compiled simulators, which operate typically at 200 - 400 million instructions per second [38].

Cycle-accurate simulators which partition their hardware implementation into a timing and a functional model are, for example, the FPGA-based simulator FAST [51] or the interpretive simulator SIMPLESCALAR [41]. FAST raises simulation performance by moving the timing model to an FPGA and leaves the functional model as a software implementation.

Another possibility to speed up simulation and maintain a sufficient precise view of the micro-architectural state is statistical *cycle-approximate* simulation. This technique is based on the idea of using previously collected cycle-accurate profiling information to estimate program behavior. This is much faster than running the whole program in cycle-accurate mode and provides a sufficient approximation. Examples for cycle-approximate simulations are the following:

SIMPOINT is a set of analytical tools, that uses basic block vectors and clustering techniques to estimate program behavior [52]. The frequency of basic blocks being executed is measured and used to build basic block vectors (BBV). BBVs indicate specific code intervals, which represent program behavior at this point. Calculating distances between BBVs is then done to find similarities in program behavior. Now similar BBVs are summarized into clusters and one BBV of each cluster is chosen as representative. Simulating only the representatives (called SIMPOINTS) speeds-up simulation time and provides representational statistics for the whole program.

SMARTS stands for Sampling Micro-Architecture Simulation [53]. It is a framework used to build cycle-approximate simulators. An implementation of the SMARTS framework is the SMARTSIM simulator. In a first step this simulator systematically selects a subset of instructions from a benchmark's execution path. These samples are then simulated in cycle-accurate whereas the rest of the benchmark is executed in fast functional mode. Collected information during sample simulation are used to estimate the performance of the whole program. Each sample has a prior warm-up phase where the simulator already runs in cycle-accurate mode to build up a proper micro-architectural state. The length of the warm-up phase depends on the length of the micro-architecture's history and must be chosen correctly for proper cycle-accurate results.

Regression analysis and machine learning is the key strategy for ARCSIM's cycle-approximate mode [54, 55]. A set of training data is profiled cycle-accurately to construct and then update a regression model. This model is used for a fast functional simulation to predict the performance of previously unseen code. Training data is either a set of independent training programs or parts of the actual program being simulated. Detailed information about ARCSIM's cycle-approximate mode can be found in Section 3.3.3.

3.3 ARCSIM Simulator

The next section presents the ARCSIM Instruction Set Simulator [56]. It was developed at the Institute for Computing Systems Architecture [57], Edinburgh University, by the PASTA research project group. ARCSIM is a target-adaptable instruction set simulator with extensive support for the ARCOMPACT ISA [6]. It is a full-system simulator (see 3.2), implementing the processor, its memory sub-system including MMU, and sufficient interrupt-driven peripherals like a screen or terminal I/O, to simulate the boot-up and interactive operation of a complete Linux-based system. Multiple simulation modes are supported, whereas the main focus lies in fast dynamic binary translation using a just-in-time compiler and cycle-approximate simulation, based on machine learning algorithms.

Since ARCSIM is a full-system simulator, it supports simulation of external devices. They can be accessed using an API for *Memory Mapped IO* and run in separate threads from the main simulation thread. Among them is, for example, a screen for graphical output and a UART for terminal I/O.

3.3.1 PASTA Project

PASTA stands for **P**rocessor **A**utomated **S**ynthesis by i**T**erative **A**nalysis [58]. The project began in 2006, funded by a research grant from EPSRC[1]. About 10 researchers are involved and work on different topics in the area of architecture and compiler synthesis.

The PASTA group's research focus lies in automated generation of customizable embedded processors. By using different tools supporting design-space exploration, information about micro-processor characteristics are analyzed and accurately predicted for different architectural models. Processor characteristics essential for embedded systems are, for example, speed and power consumption. Also of interest is the effect ISA choice has on execution time, as well as how certain compiler optimizations change the effectiveness of the overall number of instructions executed. Information about those characteristics enable designers to select the most efficient processor designs for fabrication.

As shown in Section 3.2.1, ISS strongly support design space exploration. Hence, ARCSIM is a vital tool in PASTA's research area.

[1] EPSRC is the main UK government agency for funding research and training in engineering and the physical sciences.

3.3.2 EnCore Micro-Processor Family

As a result of research activities at the Institute for Computing Systems Architecture, the ENCORE micro-processor family was created [59, 60]. ENCORE is a configurable and extensible state-of-the-art embedded processor implementing the ARCOMPACT™ ISA [6]. Its micro-architecture is based on a 5-stage interlocked pipeline with forwarding logic, supporting zero overhead loops (ZOL), freely intermixable 16- and 32-bit instruction encodings, static and dynamic branch prediction, branch delay slots, and predicated instructions. In this study the used configuration has 8K 2-way set associative instruction and data caches with a pseudo-random block replacement policy.

The processor is highly configurable. Pipeline depth, cache sizes, associativity and block replacement policies, as well as byte order (i.e. big endian, little endian), bus widths, register file size and instruction set specific options such as instruction set extensions (ISEs) can be customized. It is also fully synthesisable onto an FPGA and three fully working application-specific instruction-set processors (ASIP) have been taped-out recently. Two were made with the code-name CALTON, one fabricated in a generic 130nm CMOS process and the other one with 90nm. CALTON does not support instruction set extensions, has a 5-stage-pipeline and 8 K instruction and data caches. The third chip with the name CASTLE was fabricated in a generic 90nm CMOS process, supports instruction set extensions for AAC audio decoding, has a 5-stage-pipeline as well and 32 K caches.

ARCSIM models the ENCORE processor and serves as a golden standard model for further research and development work. Prior to this study a multi-core version of ENCORE was already designed and able to run within an FPGA platform (see Section 3.3.5). However, ARCSIM was only capable of simulating the single-core version.

3.3.3 ARCSIM Simulation Modes

Five different simulation modes are supported by the ARCSIM ISS:

- Cycle-accurate simulation
- Fast cycle-approximate simulation
- High-Speed JIT DBT simulation
- Hardware/Software co-simulation

- Register tracking simulation

Each mode implements a different simulation strategy (see 3.2.2), which makes the simulator a useful tool for various tasks. The simulator can be executed *cycle-by-cycle*, which is rather slow but very accurate, at high-speed based on a functional *instruction-by-instruction* mode, or in a lock-step mode with common hardware tools. The following text gives an overview of the single simulation modes, whereas the High-Speed simulation is described in more detail as this study's ARCSIM multi-core extension is implemented for this mode. Later multi-core extensions for the remaining modes are possible, but would have exceeded the scope of this work.

Cycle-Accurate Simulation The target program is executed cycle-by-cycle on the RTL level with a precise timing model. This enables the simulator to collect detailed micro-architectural statistics like cycles per instruction (CPI) or cache latencies. However, this simulation mode is very slow and therefore not sufficient for all upcoming software and hardware design tasks.

Fast Cycle-Approximate Simulation Cycle-approximate simulation is based on cycle-accurate mode but provides a significant speed-up. The key strategy in this mode is regression analysis. This method tries to find a regression model that shows the relationship between single values within a set of statistical data. Provided a sufficient model was found, it can be used to directly calculate specific data values depending on a set of given input data. The choice of the regression model's type (linear, polynomial, etc.) is an essential factor for the quality of estimated statistics and depends on the mechanism that generated the data.

An early version [54] of this strategy was implemented for ARCSIM as follows: In a first *training stage* a set of training programs are profiled in both slow cycle-accurate and fast functional mode. Collected counter statistics like cycle and instruction count form single data points. They are processed by a regression solver to calculate regression coefficients and build up a corresponding regression model. In a later *deployment stage* previously unseen programs are now simulated in fast functional mode. The regression model is used to predict cycle-accurate information for available functional statistics.

A refined version [55] uses machine learning algorithms to include the training stage into the actual simulation. During a single simulation run, the simulator is changing between cycle-accurate and instruction-accurate mode, depending on the current stage. In the training/learning stage, the target program is simulated cycle-

accurate and the regression model is updated. As soon as prediction of upcoming values is sufficient enough, the simulator changes into fast instruction-accurate mode and uses the regression model to estimate low-level statistics. If the quality of predicted data falls below a customized threshold, simulation switches back to training stage, possibly repeating the last time slice, and updates the model again.

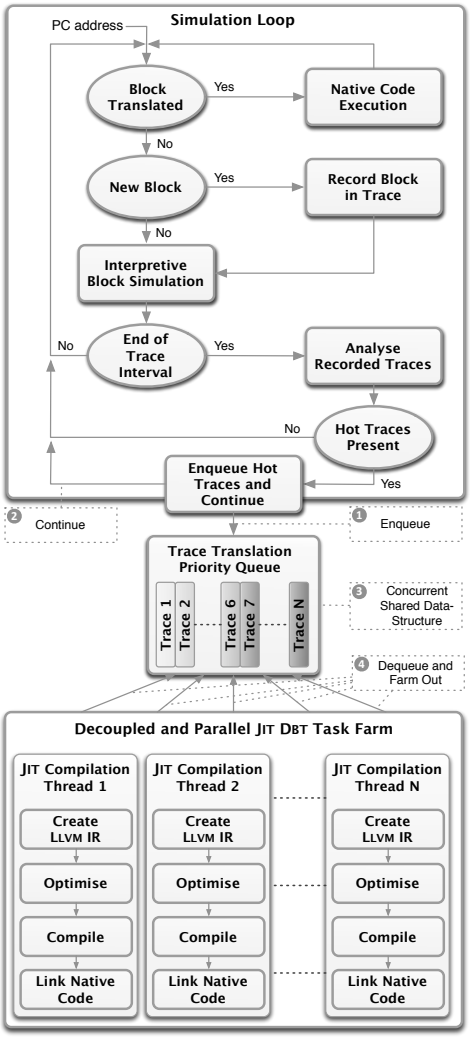


Figure 3.5: JIT dynmaic binary translation flow. [61]

High-Speed Simulation The High-Speed mode is based on DBT using a Just-In-Time compiler to gain significant speed-ups. In fact speed-ups were possible which even exceed the native execution performance of speed-optimized silicon implementations of the ENCORE processor [7]. As the evaluation chapter will show later, the same has been achieved again for ARCSIM's multi-core extension done in this study.

As mentioned in Section 3.2.2 most DBT simulators use a hybrid simulation approach to balance out additional translation complexity at runtime. ARCSIM translates only frequently used code portions to native code, while the rest is still interpreted. Obviously the more code that is executed natively, the faster the simulation becomes. At the same time the simulator is flexible enough to support dynamic code changes like self-modifying code and precise enough to maintain detailed micro-architectural observability [7].

Figure 3.5 gives an overview of ARCSIM's JIT DBT algorithm for a single-core system. Simulation time is partitioned into trace-intervals. During a trace interval instructions are either interpreted or, if a translation is present, executed natively. After each interval, statistics recorded for executed traces are analyzed. If a trace is considered hot, it is dispatched to a concurrent trace translation priority queue (see ③ in Figure 3.5).

A trace-interval starts with fetching the next PC address from memory. Depending on which block entry the address points to, the following events are possible (see flow-chart in Figure 3.5):

- The address hits an entry in the Translation Cache (TC), which means the following block has been translated recently and a pointer to a corresponding Translated Function (TF) in native code is present. In that case, the TF is executed.
- The address is not registered in the TC but can be found in the Translation Map (TM), which contains an entry for every translated trace. In that case, the TC is updated with the entry found in the TM and the corresponding function is executed.
- A block is discovered that has not yet been translated and is therefore neither found in the TC nor in the TM. In that case the block is interpreted and corresponding execution statistics are updated.
- The block belongs to an already translated trace but has been changed by self-modifying code for example. In that case, the translation is discarded and the block is interpreted.

If the end of a trace-interval is reached, collected profiling information about interpreted basic blocks is analyzed. With a simple heuristic depending on *recency* and *frequency* of interpretation, the simulator decides if a trace is hot. If so, the trace is enqueued in a shared priority queue (see ③, ④ in Figure 3.5), where a Jit translation worker can access it. In a single-threaded execution model, the interpreter would pause until translation is done. ARCSIM's simulation loop can continue directly after dispatching translation items (see ② in Figure 3.5), while the decoupled

JIT DBT task farm translates hot traces concurrently. Running several JIT translation workers asynchronously to the simulation loop in separate threads helps to hide the compilation latency - especially if the JIT compiler can run on a separate core. In addition, translations for newly discovered hot traces are available earlier because multiple hot traces can be translated at the same time. The lower part of Figure 3.5 shows the decoupled and parallel JIT DBT task farm and the compilation process.

Translation to native code is based on the LLVM [62] compiler infrastructure and partitioned into multiple steps. At first, target instructions are mapped onto corresponding C-code statements. Intermediate C-code is actually not necessary but provides better debuggability. These functions are then processed by the LLVM compiler, which translates them into an Intermediate Representation (IR) and applies standard LLVM optimization passes. In the last step shared libraries in `x86` host-code are created and loaded by a dynamic linker. To get the most out of LLVM optimizations, the granularity of translation units can be adjusted to very large ones, which comprise complete CFGs or memory pages instead of a single basic block [63]. Hence, more space for code optimizations is provided.

At any time during JIT DBT simulation, a detailed view of the ENCORE micro-architecture is maintained. This is possible by partitioning the simulation into a functional and a simple, yet powerful, software timing model. Typically cycle-accurate ISS simulate every single step of the hardware pipeline cycle-by-cycle and simply count executed cycles to get corresponding statistics. The essential idea of ARCSIM's simulation mode is simulating instruction-by-instruction and updating the micro-architectural state afterwards. Instead of modeling every detail, the timing model adds up array entries with corresponding latency values for each pipeline stage, operand availabilities and so forth. Using this timing model, the High-Speed mode provides precise micro-architectural information and is on average still faster than a speed-optimized FPGA implementation of the ENCORE processor [7]. If precise information is not required, the timing model can be deactivated. A pure functional JIT DBT simulation is on average 15 times faster than a simulation using the cycle-accurate model.

Hardware/Software Co-Simulation The cooperative simulation runs the simulator in lock-step with a hardware simulation environment (i.e. MODELSIM). An extra tool called COSIM is used to control the hardware-software co-design process. It uses the Verilog Programming Language Interface (VERILOG PLI) to interact with the internal representation of the design and simulation environment. After each executed instruction the results of both simulators are checked against each other. With ARCSIM as a golden master reference model the ENCORE VERILOG chip design can be verified.

Register Tracking Simulation This profiling simulation mode works like the modes mentioned above and adds additional statistics about the simulation run. Register accesses like reads and writes are tracked and enable the collection of statistics like dynamic instruction frequencies, detailed per-register access statistics, per-instruction latency distributions, detailed cache statistics, executed delay slot instructions, as well as various branch prediction statistics.

3.3.4 ARCSIM's Processor Model Caches

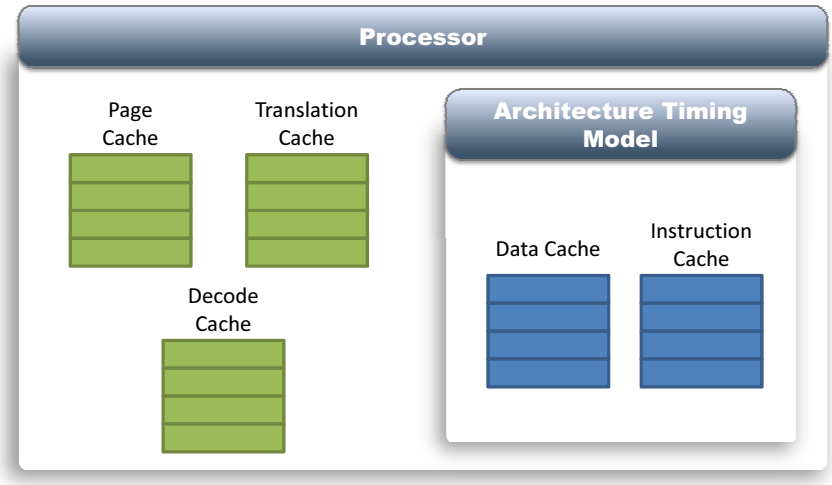


Figure 3.6: ARCSIM's different Processor model caches.

The Processor model used by the ARCSIM simulator has multiple different caches. This section gives a quick overview of the different types and their purpose. As you can see in Figure 3.6, the caches can be separated in two different groups. One group is part of the timing model to correctly simulate target architecture timings. The other group is used to speed up simulation performance.

Data Cache This cache is used to speed up access to frequently needed data addresses in the main memory.

Instruction Cache This cache stores a couple of instructions up front to speed up the fetch-stage during program execution.

Translation Cache This cache holds references to frequently used translations of target to native code which are needed for High-Speed simulation.

Page Cache Recently used memory pages are stored here for fast access. The page cache is also helpful in avoiding repeated translations of target memory addresses to host memory addresses, which is quite costly.

Decode Cache Since the decoding of instructions is expensive, this cache is used to provide access to already decoded instructions, if they are needed again later on.

In addition to the private processor caches comes a second level translation cache, which is accessed by the JIT DBT translation workers. It also speeds up simulation performance for ARCSIM's High-Speed mode (see Section 4.2 for details).

3.3.5 Target System Architecture

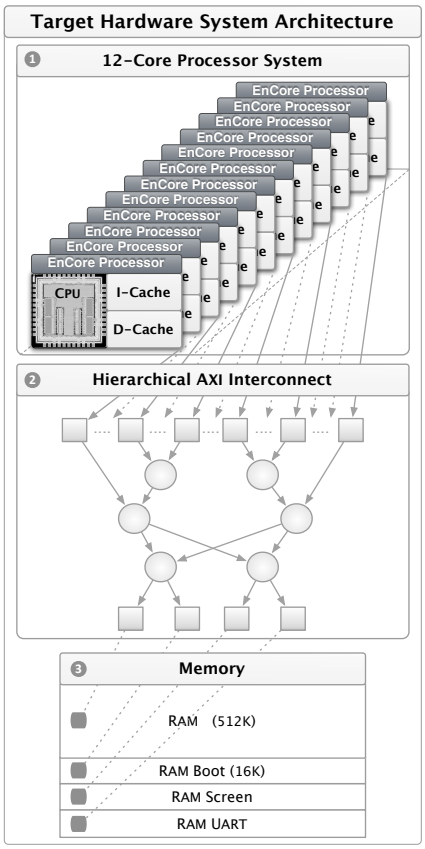


Figure 3.7: Multi-core hardware architecture of the target system.

As a physical reference system this study uses a 12-core implementation of the multi-core system, synthesised for a Xilinx X6VLX240T FPGA. The system ar-

chitecture is shown in Figure 3.7. The twelve processor cores (① in Figure 3.7) are connected through a 32-bit hierarchical, switched, non-buffered AXI interconnect fabric (② in Figure 3.7) to RAM and I/O devices (③ in Figure 3.7). An ASIP implementation of the same ENCORE processor, implemented in a generic 90 nm technology node, is currently running in the PASTA research group's laboratory at frequencies up to 600 MHz. The processor cores can attain a 50 MHz core clock using this FPGA fabric, while the interconnect is clocked asynchronously to the cores at 75 MHz.

I/O devices connected to the system are a display driver with a 40 Kbytes memory mapped character buffer, and a UART. Memory in the system is limited to 512 Kbyte RAM plus a 16 Kbyte bootup PRAM. JTAG accessible utility functions and event counters were inserted to be able to record data from the cores. Recorded data for each core includes total clock cycles when not halted, total committed instructions, total I/O operations, and total clock cycles spent on I/O operations. From these counters the MIPS of each core are calculated at 50 MHz (FPGA) and 600 MHz (ASIP), respectively.

3.3.6 `libmetal`: A Bare-Metal Library

Typically binaries being executed on a computer system use the underlying operating system's functionality like file or device access instead of implementing it by themselves. Even though ARCSIM is able to boot up and run a complete Linux OS for a single-core simulation, it is sometimes necessary to execute binaries *bare-metal*. Bare-metal means no underlying OS is present at runtime. In order to avoid implementing basic functionality for each binary, a bare-metal library called `libmetal` has been created.

Since `libmetal` is only a library, it does not fulfill all operating system tasks. But it provides enough functionality to run target applications without extensive modifications to their program code. Key components are among others a keyboard interrupt handler, a keyboard API, a screen output API and an implementation of the *printf* family.

Section 4.3 describes how this library has been further extended for multi-core target binaries running on multi-core ARCSIM.

3.4 POSIX Threads

As this study is about extending an application for single-core programs to an application supporting multi-core ones, this section will give an overview of multi-threaded programming techniques using `pthreads`. This standard programming interface for multi-threaded applications has been implemented for the `libmetal` library mentioned in Section 3.3.6 and allows multi-core ARCSIM to run multi-threaded target binaries using the `pthread` interface without any modification of the source code.

`Pthread` stands for POSIX threads, where POSIX (Portable Operating System Interface) indicates the family of IEEE operating system interface standards in which `pthread` is defined. This standardized C language threads programming interface provides the functionality to create independent streams of instructions, which can be scheduled by the operating system to run in parallel or interleaved. `Pthreads` were originally created for UNIX based systems and are now a popular tool for writing multi-threaded programs [64, 65].

This section will describe three major classes of routines in the `pthread` API, which are thread management, mutex variables and condition variables.

3.4.1 Thread Management

The `pthread` interface is not necessary to create parallel applications on a UNIX system. You could also use the *fork* command which copies the currently running process and creates a child process that is almost identical to its parent. The new process gets a different Process Identification (PID) and the fork call returns different values to the parent and the child. Using this information a programmer can create different execution paths through the application for both processes and the operating system can interleave or even parallelize them for multiple cores.

However, creating and managing a process comes with a lot of runtime overhead for the operating system. Calling a light-weight *pthread_create* needs much less work, since no new process needs to be created. Threads exist within their parent process and can therefore use corresponding resources. Only bare essential resources are duplicated such as the stack pointer, registers, scheduling properties, a set of pending and blocked signals and thread specific data. This enables a thread, even if it is running inside a parent process, to be scheduled individually. In addition, inter-thread communication is more efficient and in many cases easier to use than inter-process communication.

Figure 3.8 shows an example of creating and terminating threads using the `pthread` interface which can be accessed by the header included in line 1. Every thread apart from the main thread prints a string with its thread id. The main tread uses a *pthread_create* call in line 19 to create five new threads. This call expects a unique identifier for the new thread, an object to set thread attributes, the C routine that the thread will execute once it is created and a single argument that may be passed to the start routine. The routine used here is *PrintHello* which is defined in line 6. Upon exiting the routine *pthread_exit* is called to shut down the thread and free corresponding resources. It is also possible to exit a thread from a different thread by calling *pthread_cancel* or shut it down by terminating the parent process.

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define NUM_THREADS     5
5
6  void *PrintHello(void *threadid) {
7   long tid;
8   tid = (long)threadid;
9   printf("Hello World! It's me, thread #%ld!\n", tid);
10  pthread_exit(NULL);
11 }
12
13 int main(int argc, char *argv[]) {
14  pthread_t threads[NUM_THREADS];
15  int rc;
16  long t;
17  for(t=0;t<NUM_THREADS;t++) {
18   printf("In main: creating thread %ld\n", t);
19   rc = pthread_create(&threads[t], NULL,
20                       PrintHello, (void *)t);
21   if (rc) {
22    printf("ERROR creating thread: %d\n", rc);
23    exit(-1);
24   }
25  }
26  pthread_exit(NULL);
27 }
```

Figure 3.8: `Pthread` creation and termination

Of course, many more functions are provided to manage threads with the `pthread` interface, like joining threads or modifying the thread environment or runtime.

3.4.2 Mutex Variables

An important task which goes hand in hand with using multiple threads is providing thread safeness. That means, if more than one thread accesses a shared resource such as a global object or global memory, then access to this resource needs to be synchronized. If not, race conditions or synchronization errors can occur. A race condition appears as a result of unpredictable program behavior in parallel applications. Since thread scheduling is not deterministic, a programmer can not rely on specific timings in a parallel program execution if it is not synchronized explicitly. If synchronization is missing, one thread could always execute a specific code portion before the other ones, until it arbitrarily happens the other way around and execution results change or even an error occurs. An error could, for example, arise, if one thread deletes an object from a list, while a different thread is already modifying it.

```
1  #include <pthread.h>
2  pthread_mutex_t count_mutex;
3  long count;
4
5  void increment_count() {
6   pthread_mutex_lock(&count_mutex);
7   count = count + 1;
8   pthread_mutex_unlock(&count_mutex);
9  }
10
11 long get_count() {
12  long c;
13  pthread_mutex_lock(&count_mutex);
14  c = count;
15  pthread_mutex_unlock(&count_mutex);
16  return (c);
17 }
```

Figure 3.9: Synchronizing threads with mutex varibales

A very basic synchronization mechanism provided by the `pthread` interface is the use of *mutex varibales*. Mutex is an abbreviation for mutual exclusion. It is an object that protects shared resources or critical program parts. Figure 3.9 shows how a mutex varibale is used in order to synchronize threads. The *count* variable is being protected by a mutex. A thread can increment the count variable by using the *increment_count* method or read it with a *get_count* call. Every direct access to count has a prior call of *pthread_mutex_lock* with the corresponding mutex object *count_mutex*. This call will reserve the mutex for the calling thread and block every

other thread until the mutex is freed again by a *pthread_mutex_unlock* call. A possible implementation for a mutex could, for example, use a list to gather blocked threads, send them to sleep as long as they are blocked and wake them up again when the mutex is free. Of course, if multiple threads are waiting for a mutex to be released, only one is allowed to go on. The others are sent to sleep again until it is their turn to access the protected section.

3.4.3 Condition Variables

Consider a Producer and Consumer application, where a thread takes calculation results from a different thread and processes them, if they are available. A possible implementation could be a Boolean variable that is set by the producing thread, if data is available. The Consumer runs in a while-loop constantly monitoring the variable and bails out as soon as it turns true. This implementation is called the *busy-loop* concept. It works but is not very efficient, since the waiting thread uses a lot of CPU resources polling for the Producer's notification.

A much more efficient technique is the use of *condition variables*. They are used to communicate between threads sharing a single mutex and based upon programmer specific conditions. If a thread waits on a condition variable, it goes to sleep and therefore does not need any resources. If the Producer finishes a data package, it wakes up the waiting Consumer thread and continues with its work.

Figure 3.10 shows example code implementing this scenario. The routines Producer and Consumer in line 6 and 17 are entry points for corresponding threads. The Producer thread is constantly producing work items as indicated in line 8. As soon as an item is ready, it locks the access to a global work queue, inserts the item and signals a possibly waiting Consumer thread that new items are available. Afterwards, it releases the queue again and processes the next work item. The Consumer on the other side waits at the condition variable in line 21 after locking the work queue access. As soon as it goes to sleep at the condition variable, the work_mutex gets released again. If a signal arrives, the Consumer wakes up, immediately blocks the mutex and leaves the if-clause to consume the available item in line 23 and 24. As soon as consumption is finished, it continues with the main loop and goes to sleep again. In case of multiple Consumers, the Producer would use a broadcast call instead of a signal in line 11, since a broadcast wakes up all waiting threads instead of only one. This, however, would need a second modification in line 20 where the *if* needs to be replaced by a *while*. Assume several Consumers are waiting at the condition variable. As soon as the broadcast arrives, all of them are woken up in a random order and get the mutex one after the other. The first one consumes the work item whereas the other ones stay in the while-loop and go to sleep again.

```
1  #include <pthread.h>
2  pthread_mutex_t work_mutex;
3  pthread_cond_t work_queue_cv;
4  queue<WorkItem> work_queue;
5
6  void* Producer(void* arg) {
7    for(; /* ever */;) {
8     WorkItem item = produce();
9     pthread_mutex_lock(&work_mutex);
10    queue.push(item);
11    pthread_cond_signal(&work_queue_cv);
12    pthread_mutex_unlock(&work_mutex);
13   }
14   pthread_exit(NULL);
15 }
16
17 void* Consumer(void* arg) {
18   for(; /* ever */;) {
19    pthread_mutex_lock(&work_mutex);
20    if(work_queue.empty()) {
21      pthread_cond_wait(&work_queue_cv, &work_mutex);
22    }
23    consume(work_queue.front());
24    work_queue.pop();
25    pthread_mutex_unlock(&work_mutex);
26   }
27   pthread_exit(NULL);
28 }
```

Figure 3.10: Thread communication with condition variables

3.4.4 `Pthreads` in ARCSIM

`Pthreads` are used for the ARCSIM simulator in two different ways. One way is directly for the simulator implementation, where multiple JIT translation threads are created (see 3.3.3), external devices are running in parallel and every processor gets its own simulation thread apart from the main thread (see 4.1). Since the `pthread` interface was originally designed for the C language, multiple wrapper classes have been implemented for ARCSIM to support object orientation and simplify development. A new class can now inherit from a *Thread* structure, which provides a *run* function to start the thread. Synchronization is done by special *Mutex* and *ConditionVariable* classes, which hide instantiation complexity and can

easily be used by calling, for example, `mutex.acquire()` or `mutex.release()`.

As a second way, the `pthread` interface is used in the `libmetal` library (see 3.3.6). Here a customized `pthread` interface implementation is provided for target programs. Hence, they can benefit from the `pthread` interface to run multi-threaded programs on a simulated multi-core target system. Section 4.3 explains the implementation in detail.

3.5 Benchmarks

The multi-core extensions of the ARCSIM simulator implemented in this study were tested against two official benchmark suites: EEMBC's MULTIBENCH 1.0 and SPLASH-2, as well as other applications. The following section provides a brief overview of the design and functionality of the benchmarks.

3.5.1 MULTIBENCH 1.0

The Embedded Microprocessor Benchmark Consortium (EEMBC) has produced benchmarks for embedded processors since 1997. In 2008 a new benchmark suite called MULTIBENCH 1.0 was released [66, 67]. This suite consists of compute-intensive tasks commonly performed by networking equipment, office equipment (especially printers) and consumer-electronics products.

A MULTIBENCH *task*, also called *workload*, can include one or more *kernels*, also called *work items*. MULTIBENCH 1.0 comes with 36 predefined workloads, which are combinations of different work items. A work item is simply an algorithm or routine that performs a common process found in real-world embedded software. Besides using predefined workloads it is also possible to create customized ones from a total of 14 work items. Work items can be combined in various ways as mentioned above to reflect typical application workloads of embedded systems. Table 3.2 lists all work items and gives a short functional summary for each of them.

A single MULTIBENCH workload can be designed to focus on a specific level of thread parallelism depending on the combination of work items. It can focus on thread-level parallelism among disparate work items, among two instances of the same work item, operating on different data, or on data-level parallelism within a single work item. Threading is implemented with an API that closely resembles the `pthread` API. In order to run the benchmark suite on a system support-

Name	Description
`md5`	networking routine that calculates the Message Digest 5 checksums for a given data set
`ippktcheck`	networking routine that checks the header of an IP packet
`rgbcmyk`	color-conversion routine that transforms an image from RGB color space to the CMYK color space
`rotate`	image-rotation routine that rotates an image 90° clockwise
`ipres`	networking routine that reassembles a given data set of IP packets
`mpeg2`	imaging routine that runs an MPEG-2 decoder application
`rgbyiq03`	color-conversion routine that transforms an image from RGB color space to the YIQ color space
`rgbhpg03`	imaging routine that receives a dark or blurry gray-scale image and sharpens it with a high-pass filter or smoothens it with a low pass filter
`cjpeg`	imaging routine that compresses an image to the JPEG format
`djpeg`	imaging routine that decompresses a JPEG image to the original RGB format
`tcp`	networking routine that processes data for TCP transfers
`mp3player`	audio processing benchmark that decodes an MP3 audio file
`huffde`	algorithmic benchmark kernel executes Huffman decoding on a variety of data sets
`x264`	video processing routine that encodes a video stream in H.264 format

Table 3.2: Work items provided by EEMBC's MULTIBENCH 1.0 benchmark suite.

ing `pthreads`, EEMBC's API calls can be mapped on the corresponding `pthread` functions.

3.5.2 SPLASH-2

The SPLASH-2 benchmark suite [68] is an extension of the original **S**tandford **P**aralle**L** **A**pplications for **SH**ared memory suite (SPLASH), a suite of parallel programs written for cache-coherent shared address space machines. Since the original version was not designed to work with modern memory system characteristics or a large number of cores, it has been extended and improved to a second version.

The SPLASH-2 suite is a collection of parallel programs providing a workload for the quantitative evaluation of multi-core systems. Each program is designed to partition its work using threads. In contrast to MULTIBENCH, SPLASH-2 sets the focus on scientific workloads. All together SPLASH-2 consists of twelve applications where eight of them are complete ones and the remaining four are application kernels. These programs cover a number of common complex calculations in areas such as linear algebra, complex fluid dynamics and graphics rendering. Table 3.3 lists all applications and gives a short functional summary for each of them.

Name	Description
`Barnes`	simulates the interaction of a system of bodies in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method
`Cholesky`	factors a sparse matrix into the product of a lower triangular matrix and its transpose
`FFT`	is a complex one-dimensional version of the "Six-Step" Fast Fourier Transformation algorithm described in [69]
`FMM`	works like `Barnes` but simulates interactions in two dimensions using a different hierarchical N-body method called the adaptive Fast Multipole Method
`LU`	factors a dense matrix into the product of a lower triangular and an upper triangular matrix
`Ocean`	simulates large-scale ocean movements based on eddy and boundary currents
`Radiosity`	computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method
`Radix`	implements an integer radix sort algorithm based on the method described in [70]
`Raytrace`	renders a three-dimensional scene using raytracing
`Volrend`	renders a three-dimensional volume using a ray casting technique
`Water-Nsq`	evaluates forces and potentials that occur over time in a system of water molecules using a $O(n^2)$ algorithm
`Water-Sp`	works as `Water-Nsq` but uses an $O(n)$ algorithm instead

Table 3.3: SPLASH-2 application overview.

The benchmarks `Ocean` and `LU` come in two different versions. One version uses data structures that simplify the program implementation but prevent blocks of data from being allocated contiguously. The second version in contrast allows blocks of data to be allocated contiguously and entirely in the local memory of the corresponding processor by using a more complex implementation concept for the corresponding data structures.

Like EEMBC's MULTIBENCH suite, SPLASH-2 can be built for a system supporting `pthreads`. A predefined set of macros can be modified to map onto corresponding `pthread` library calls.

3.5.3 Other Benchmarks

Besides official benchmark suites, ARCSIM was tested against a couple of other benchmark programs. These include simple applications to debug multi-core functionalities, as well as bigger and more sophisticated ones to get simulations results for more realistic workloads.

A simple test application to verify if all cores are running properly, uses a floating picture and ARCSIM's screen device. The screen is partitioned among the simulated cores, each of which draw some section of the image. Each core then pans around the image, bouncing off the border should they come to it. It can now easily be observed if all cores are running properly. If the screen partition assigned to a specific core stays black or the picture stops moving, something must be wrong in the core's update cycle. As JIT workers translate more and more target traces to native code, one can see how the movement speed of the pictures increases the longer the program runs.

A more sophisticated application, which fully utilizes a target multi-core architecture, is a parallelized fractal drawing algorithm executed across twelve cores. The fractal drawing algorithm partitions its workload into multiple parts, which are each processed by different threads. It is perfectly qualified to test multi-core architecture because of its embarrassingly parallel nature. An embarrassingly parallel workload is one for which little or no effort is required to separate the algorithmic problem into a number of parallel tasks. Hence, it is possible to assign a row of the screen to each simulated core, or even a single pixel. This easy parallelization and a low memory footprint help to avoid scalability issues, when the fractal application is used for larger core combinations.

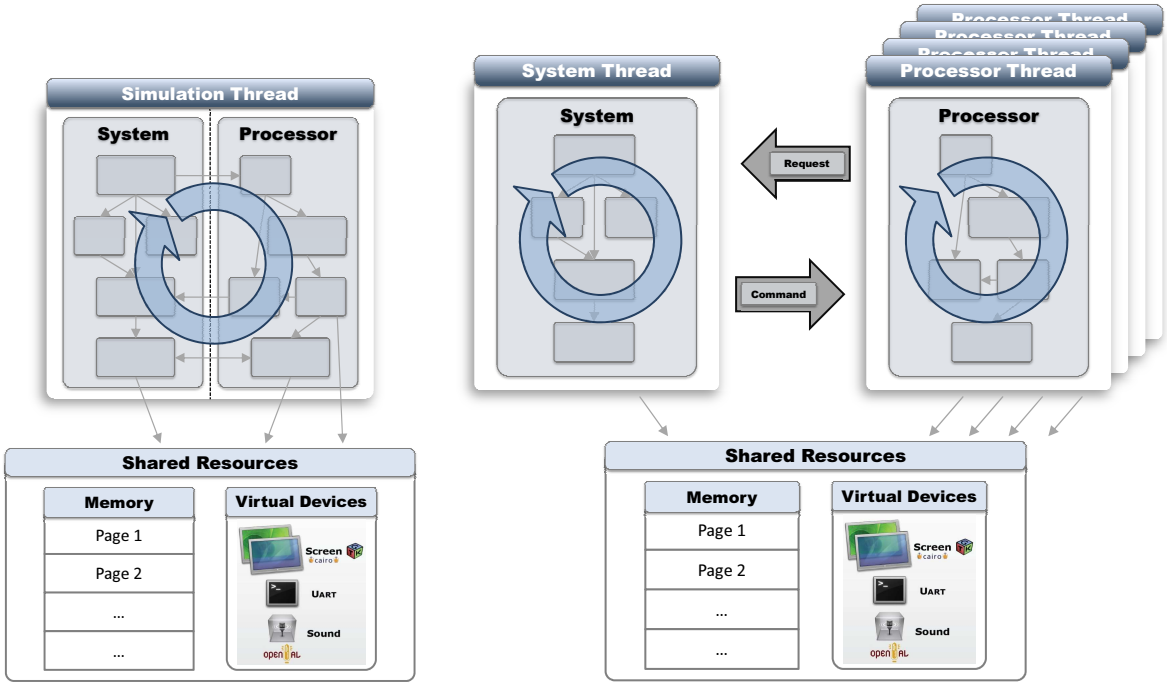
Chapter 4

Methodology

The following chapter describes how the single-core ARCSIM simulator, introduced in Section 3.3, has been extended to a multi-core simulation environment. This work splits up into two different main tasks. It is not only necessary to implement new system functionalities for running multi-core simulations. The extended simulator also needs to maintain high simulation performance even for several, simultaneously running cores. Therefore, various optimizations have been applied in this study to create a fast and scalable multi-core ISS. These include a carefully designed software architecture, which can handle complex synchronization tasks of parallel applications. A mechanism has been implemented to share translations between cores to benefit data-parallel target applications, where different cores execute the same code on different data. For a direct support of multi-threaded target programs a light-weight multi-threading library compatible to POSIX threads is provided, including an efficient mapping of atomic exchange operations to benefit synchronization. With this library multi-threaded target programs can be executed directly by the simulator without the need to run an OS or apply any changes to their program code.

4.1 Replicating the Single-Core Simulator

The main goal of this study is to extend single-core ARCSIM to a multi-core simulator. On a software architecture level this implies replicating the processor structure, which is responsible for modeling the target system by executing target applications. In ARCSIM this structure is called the *Processor* class. It is designed as a worker controlled by a supervising structure called the *System* class. The System is the first object created on program start-up and the last one active on shut-down.



(a) Single-Core Software Architecture **(b)** Multi-Core Software Architecture

Figure 4.1: Tasks being performed to replicate the CPU. The main loop has been separated into different threads, a communication system has been established and access to shared resources has been synchronized.

The System class is responsible for reading the target hardware configuration and simulation parameters to build up a corresponding simulation environment. This includes instantiating objects for a global memory model, IO-devices, the Processor, etc. After a simulation model has been fully initialized, the simulator's main loop is started. It runs sequentially in a single thread, where the control flow changes repeatedly between controlling System and working Processor (see 4.1a).

To begin using multiple Processors it is necessary to identify System and Processor specific tasks, separate them and split up the main loop. This allows us to create two independent main loops for System and Processor and run them in different threads. Once the main loop is separated and the two components are running alongside each other, a communication mechanism needs to be implemented and access to shared resources must be synchronized. Now multiple Processor instances can be spawned and executed concurrently (see Figure 4.1b). In the following paragraphs this process will be described in detail.

4.1.1 Sequential Main Loop

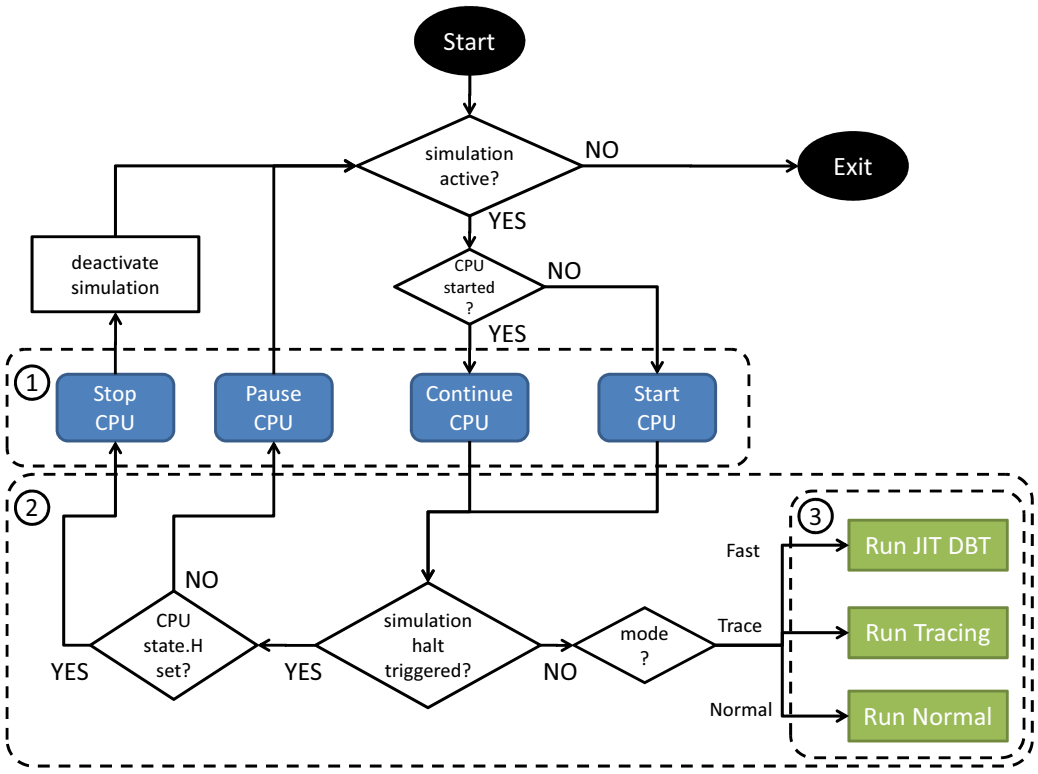


Figure 4.2: Sequential System-Processor main loop.

Figure 4.2 shows a simplified flow chart of the sequential main simulation loop. While simulation is active, the Processor (or CPU) is updated. A simulation halt can be triggered for two reasons. The first one is caused during Processor update, where an execution step encountered an exception or the end of a target program is reached. In that case the CPU's *halt* flag (state.H) is set, which is part of the target's architectural model. The second reason has an external source, where the user can interrupt the simulation, or another program calls an API function which terminates the program. The latter could happen during hardware-software co-simulation, for example, as described in Section 3.3.3.

The System controls the Processor's simulation tasks through specific function calls, which stop, start, continue or pause the CPU (see ① in Figure 4.2). Depending on their intended functionality, these functions reset or initialize state variables, log current statistics or control the Processors timing model by starting or stopping internal timers.

The lower part in Figure 4.2 marked by a ② shows which tasks are actually related to the CPU and could be extracted into a Processor loop. Here, depending on the currently active simulation mode a corresponding Processor update function is called. Number ③ marks those function calls, which execute different modes of operation of the ARCSIM simulator as described in Section 3.3.3. Since this flowchart is somewhat simplified, not all modes are displayed here. The three modes shown in the diagram are the ones that have been extended in this study. JIT DBT stands for the High-Speed mode, *Tracing* emits additional execution statistics and debug information and *Normal* is a basic interpretive simulation. After executing the update function, the halt flag is checked for errors or the end of an application. If the flag is set, the simulation terminates. However, the decision, if a simulation halt was triggered, needs to take care of external and Processor internal sources.

Spawning and starting CPU objects and handling upcoming Processor requests and external events are administrative tasks related to the System. In contrast to the Processor, it is not active all the time. It spends most of its time waiting, to be ready when it is needed. The next section will introduce the separated main loops in detail.

4.1.2 Decoupled Simulation Loops

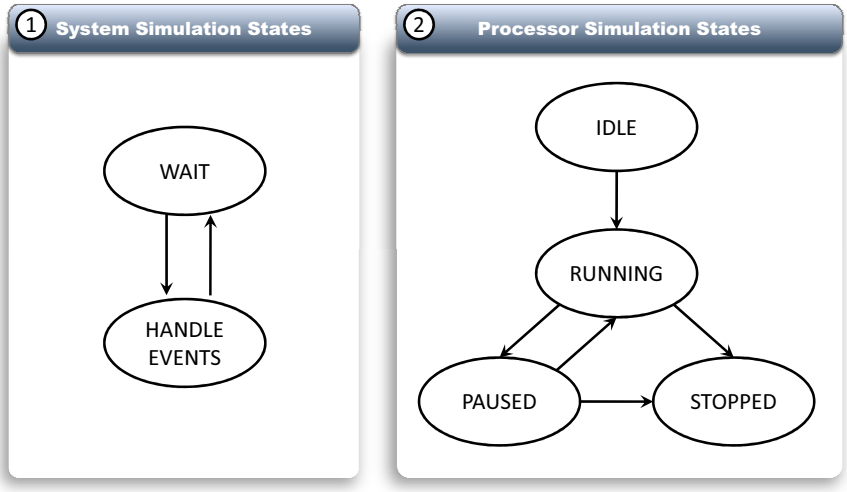


Figure 4.3: Simulation states for System ① and Processor ②.

Since the regions for System and Processor in the sequential main loop have been identified, it is now possible to create separate loops. A mechanic needs to be provided to maintain the original architecture pattern where the System is the controller and the Processors are the workers. For two separate loops working in par-

allel, this problem was solved by using state machines defining specific *simulation states*. That way the System can check what is currently happening in the Processor and direct the CPU's program flow by changing its simulation states.

The Processor's state machine consists of four different states (see ② in Figure 4.3):

Idle This state is the Processor's initialization state. It is set only when the object is being instantiated.

Running In this state, the Processor runs through its main loop and executes the target program.

Paused The Processor sleeps and does nothing. If it gets woken up again, it resumes operation.

Stopped Simulation has been completed and the Processor leaves its main loop.

These states enable the System component to control all running CPUs by changing their states and therefore directing them to act according to the System's commands. It also helps to avoid errors when the System wants to interfere with the Processor's update loop, for example, to shut it down or pause it.

Debug information useful to the programmer is provided. For example, to test the simulator's functionality, a programmer could use a self-made target application. This application switches between assigning calculation work, pausing, resuming or stopping available processors. By logging the current processor state and state changes in addition to executed instructions, it is possible to observe exactly how the simulated processors behave. Odd behavior like processors never reaching the STOPPED state or never going to sleep even if a sleep-instruction was processed, can be found more easily this way.

The System does not have an actual implemented state machine for its main loop, since it only needs two different states (see ② in Figure 4.3). One is handling upcoming events and the other is waiting for them. Section 4.1.3 lists possible simulation events and explains how to handle them in detail. It also shows how System and Processor interact with each other, since changing variables directly in a parallel environment can result in race conditions or deadlocks.

Parallel Simulation Loops If no problems like exceptions or external disturbances occur during the simulation of an application, the System has almost nothing to do. After creating the simulation environment and spawning and starting the Processors, it goes to sleep. As long as it is sleeping, it does not use any host CPU

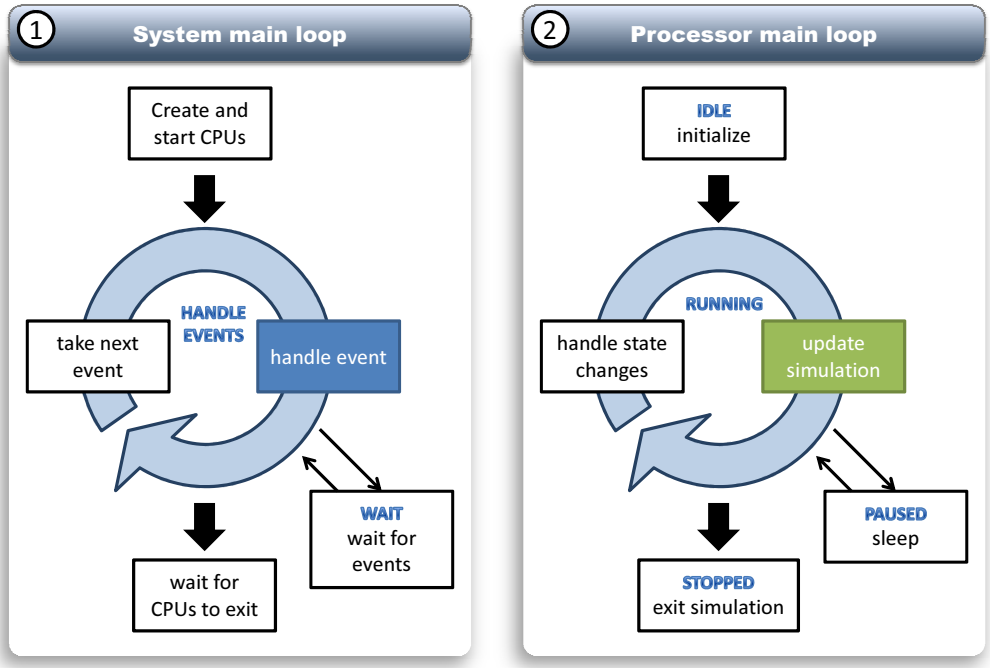


Figure 4.4: Parallel main loops for System ① and Processor ②.

resources which benefits simulation performance. As soon as simulation events arise, it wakes up, handles all pending events and goes to sleep again. If the end of a simulation is reached, it shuts down all CPUs, cleans up the memory and exits the application (see ① in Figure 4.4). The blue box labeled "handle event" is the equivalent of the blue boxes in the sequential main loop from Figure 4.2. Here the System changes Processor states indirectly by telling the CPU, which state to enter next.

The Processor's main task during its simulation loop is executing the target application, which is indicated by the green box labeled "update simulation" in Figure 4.4 ②. This box is equivalent to the green boxes in Figure 4.2. In each cycle it also checks for new state change notifications from the System and handles them, if necessary. It can break out of the loop and sleep, if it gets PAUSED, or leave the main loop by exiting the simulation.

To efficiently utilize multi-core host hardware, ARCSIM runs System and Processors in separate threads. The System runs in the application's main thread whereas the Processor classes are derived from the `pthread` wrapper class. A class deriving from the `pthread` wrapper needs to implement a run method. This method can then be called by the System in order to start the thread and therefore the Processor's main loop. Upon leaving the main loop, the CPU thread joins the main

thread and all resources are released (see Section 3.4 for details about `pthreads` and the implemented wrapper).

4.1.3 System-Processor-Communication

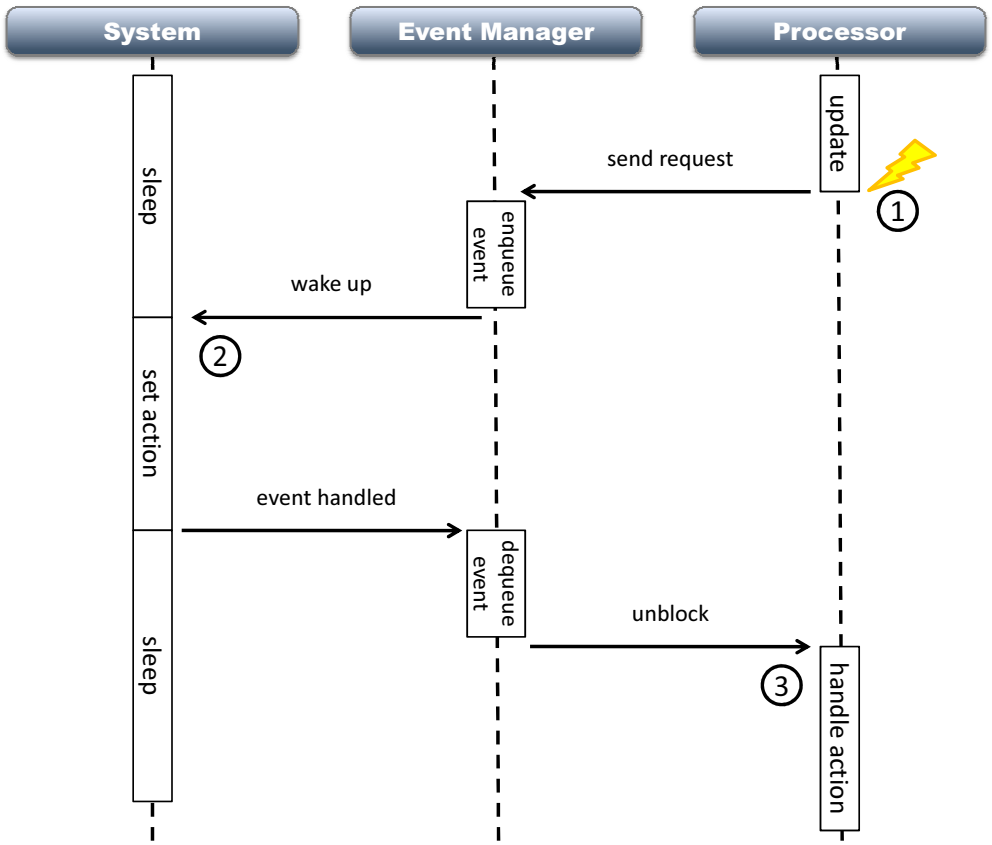


Figure 4.5: Sequence diagram showing how Processor events are handled by the System using the Event Manager.

System and Processor communicate with each other by using a structure called the *Event Manager*. This interface structure provides a synchronized method to send, handle and notify requests. Figure 4.5 shows an example sequence for communication between the Processor and the System.

Initially the System sleeps and the Processor executes a target application. At ① an event arises during the Processor's update cycle. For example, this could be a sudden exception in the target program flow. As the Processor recognizes the exception, it stops working and sends a *request* using the Event Manager. This intermediate structure creates a *simulation event item* for the request, puts it into an

internal *event queue* and wakes up the System. The System gets the wake-up call at ②, retrieves the pending event from the Manager and handles it as necessary. In our case it creates a *state change action item*, which tells the Processor to stop. This item is then placed in another queue inside the Processor's state machine structure (explained later) and the event is marked as handled. Now the System's work is done and it goes back to sleep. It is possible that multiple events are pending in the event queue. In that case, the System stays awake and processes all events. In our example the handled event is removed from the Manager's queue and the Processor, waiting for its request to be handled, gets *unblocked*. At ③ the Processor continues with its main loop. As it recognizes the action item in its state machine queue, it sets its own simulation state from RUNNING to STOPPED.

Using this intermediate structure for communication has several advantages in comparison to multiple communicating components accessing each other directly:

- Communication is done asynchronously. The component with the upcoming request does not need to wait until a receiver is ready to listen. However, in ARCSIM's case communication is blocking and the Processor waits for the request to be handled. This behavior is necessary since the System is the controlling component and decides what to do next for upcoming requests, before the CPU continues.
- Senders do not need to know the receiver of a message, as long as it is handled properly. It is also possible to register multiple receivers of which one is handling the events and the other one is monitoring them, for example.
- It is easy to add, remove or exchange participants at runtime without interfering with other components. The Processor class in our example could easily be exchanged for a user interrupt handler. The user closes the application window, which invokes an interrupt handler. This handler creates an event to stop all Processors and the System, and hands it over to the Event Manager. This is actually done when shutting down ARCSIM at runtime by closing the Screen device window.
- Creation of new simulation events is done easily by adding a new type and implementing corresponding handlers in all participating structures.
- Communication between multiple partners can easily be synchronized, where synchronization complexity is hidden from the calling components. The Event Manger provides methods to send or receive requests and synchronizes the methods responsible for event queue access internally.

The following sections give a detailed overview on Simulation Events and State Change Action types and how they are used by the Event Manager for communication matters.

Simulation Events Two different types of simulation events are supported. One is a *Processor Event*, which is caused by a Processor. The other is an *External* or *System Event* caused by the user or an external application. Table 4.1 shows all possible event reasons:

Event type	Reason	Description
SIM_EVENT_CPU	EVENT_CPU_STOP	If an issue arises during the Processor's update cycle, this event reason is used.
	EVENT_CPU_PAUSE	If the Processor encounters a sleep instruction, this event reason is used.
SIM_EVENT_SYSTEM	EVENT_SYS_STOP_ALL	If the user interrupts simulation and the whole application needs to shut-down, this event reason is used.

Table 4.1: System and Processor simulation event reasons.

State Change Action Parallelism and high simulation speed can cause multiple state changes for a single CPU to quickly follow each other. Because of external System Events, a state change can arise asynchronously anywhere in the Processor's update cycle. In order to still guarantee proper handling of upcoming changes, the CPU's state is only changed directly by the CPU itself. However, this is done only on behalf of the supervising System structure. The System uses state change actions as commands in order to tell a CPU which state to use next. This ensures proper handling of the problems mentioned above. Multiple commands are processed at a safe spot in the Processor's main loop, in the right order and without losing one.

Incoming state change actions are stored in the Processor's state machine structure by using a synchronized queue for concurrent access. At the beginning of each loop cycle, the Processor scans this queue for new state changes. If an action is present, the corresponding state is set and the main loop starts again. Possible state change actions are listed in Table 4.2.

Action	Description
CPU_STATE_CHANGE_RUN	Switch the Processor state to RUNNING mode.
CPU_STATE_CHANGE_PAUSE	Switch the Processor state to PAUSED mode.
CPU_STATE_CHANGE_STOP	Switch the Processor state to STOPPED mode.

Table 4.2: Processor state change actions.

Of course, a lot more events and actions are possible. For example, single-core ARCSIM has an interactive mode, where target program execution can be interrupted at any time to monitor the current state or change simulation modes. Possible actions for the interactive mode could be "change to fast mode", "execute the next N instructions" or "print architecture state". This work, however, exceeds the scope of this study.

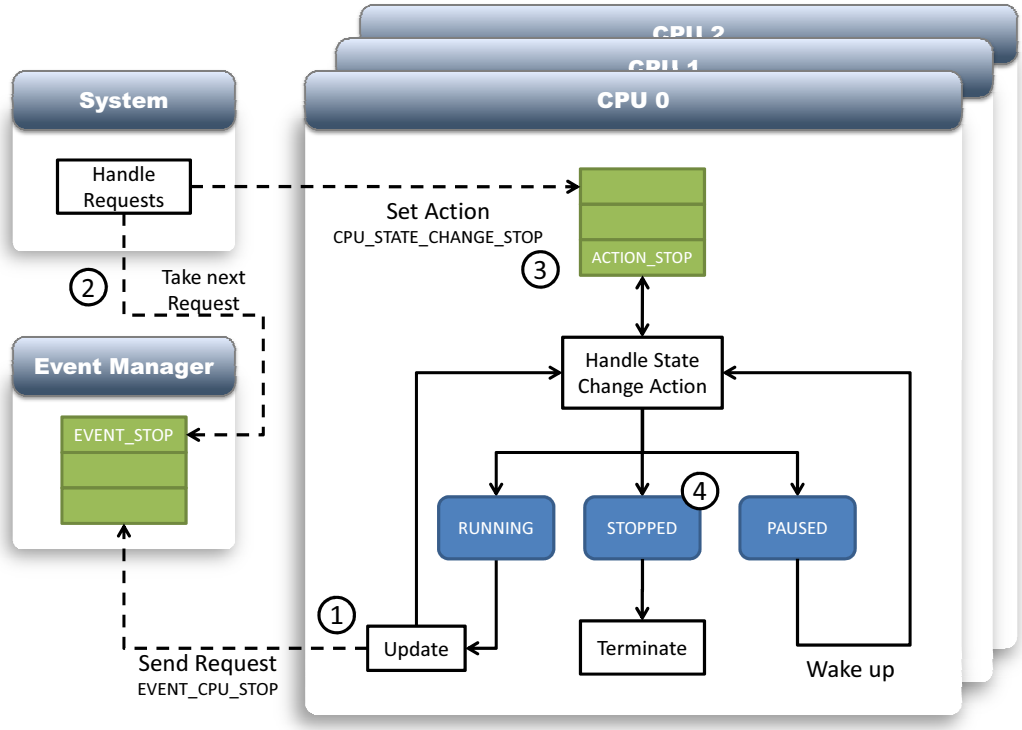


Figure 4.6: Interaction between Processor, Event Manager and System to handle state changes.

Figure 4.6 shows how Processors, Event Manager and System interact by using event- and action-items to control CPU states. In contrast to Figure 4.5 it does not focus on the the exact control flow sequence, but the underlying data structures. Initially all Processors are updating a target application, when CPU 0 encounters the end of the application at ①. Immediately the CPU sends a STOP-Request using the Event Manager interface. The request is placed in a queue inside the Manager class, where the System class can access it, as seen at ②. Recognizing that CPU 0 is finished, the System creates a state change action item with the STOP-command and puts it inside the command queue, addressed from CPU 0, as can be seen at ③. Eventually CPU 0 is unblocked by the Event Manager, as depicted in Figure 4.5, and checks its command queue. Since the STOP-command has been registered, the Processor changes its state to STOPPED in ④ and terminates.

As said above, this mechanism ensures that no event causing a state change gets lost or reordered. Even if, for example, the CPU's control flow is at ① (because it needs to be paused) and at the same time an external event causes a state change to STOPPED because the user closed the Screen application. This would result in two action items being placed inside the command queue, which are then executed one after the other; first a CPU_STATE_CHANGE_PAUSE, and, after going to sleep and waking up again, a CPU_STATE_CHANGE_STOP.

Simulation Event Manager Figure 4.7 provides a detailed view inside the Simulation Event Manager. The left block depicts a *Sender*, which is, in our case, a Processor or a user interrupt handler, and the right block displays a *Receiver*, which would be the System. The middle part shows provided methods allowing access to indicated functionality. Any structure using the Manager's methods to send events is called a Sender, whereas a Receiver can be any structure listening for event notifications and handling them, if they are present. The Manager uses a queue to store events. This queue is synchronized with a mutex and a condition variable (see Section 3.4 for details).

Sender To send a request, the methods *send_cpu_event* or *send_system_event* are used. Those methods automatically select the correct event type and need an event reason as an argument. *send_cpu_event* additionally needs the id of the sending CPU. Directly after entering the method, the event queue is locked with a mutex (QM). This assures only one caller can access the queue at a time. An event item is created with the given arguments and inserted in the event queue. Afterwards, a broadcast is executed for the queue condition variable (QCV) to wake up all waiting threads and thereby notify them about the pending work. As long as the event is not handled, the Sender will be blocked at the QCV with the corresponding QM. While waiting at the QCV, the QM, which protects the event queue access, is released. If not, a Receiver would not be able to retrieve events for handling. When all events are processed, the Sender gets a notification, wakes up from waiting at the QCV, releases the queue and exits the method.

Receiver A Receiver uses two methods to mark the beginning and end of a handling phase. These are *start_event_handling* and *end_event_handling*. Internally these methods lock and release the QM. If another component is currently modifying the event queue by reading, removing or adding events, the Receiver has to wait at the start method until the Sender is done. Of course, this also works the other way around. After locking the QM, the Receiver invokes the *wait_for_events* method and blocks until new events arrive. Internally, the manager checks for events inside the queue and sends the Receiver to sleep at the QCV, if the queue is empty. When the broadcast signal from the Sender

reaches the QCV, it will wake up the Receiver. Assuming events are pending, it calls the *get_next_event* method, which retrieves an event item from the queue and returns it. If the necessary work has been done, the *event_handled* method is called for the corresponding item. This call will set a *handled-flag* in the event item and wake up all Senders waiting at the QCV. To ensure only the Sender responsible for the request continues with its work, it can check the handled-flag of its own item. If the Sender was woken up by mistake, it goes to sleep again. This can happen if two or more senders are waiting for their request to be handled. Since a broadcast is used to wake up waiting Senders, all senders are woken up, even if only one of the pending requests has been handled so far. When all events are processed, *end_event_handling* is called and the QM is released. ARCSIM's System will loop back to *start_event_handling* and wait for new tasks to handle.

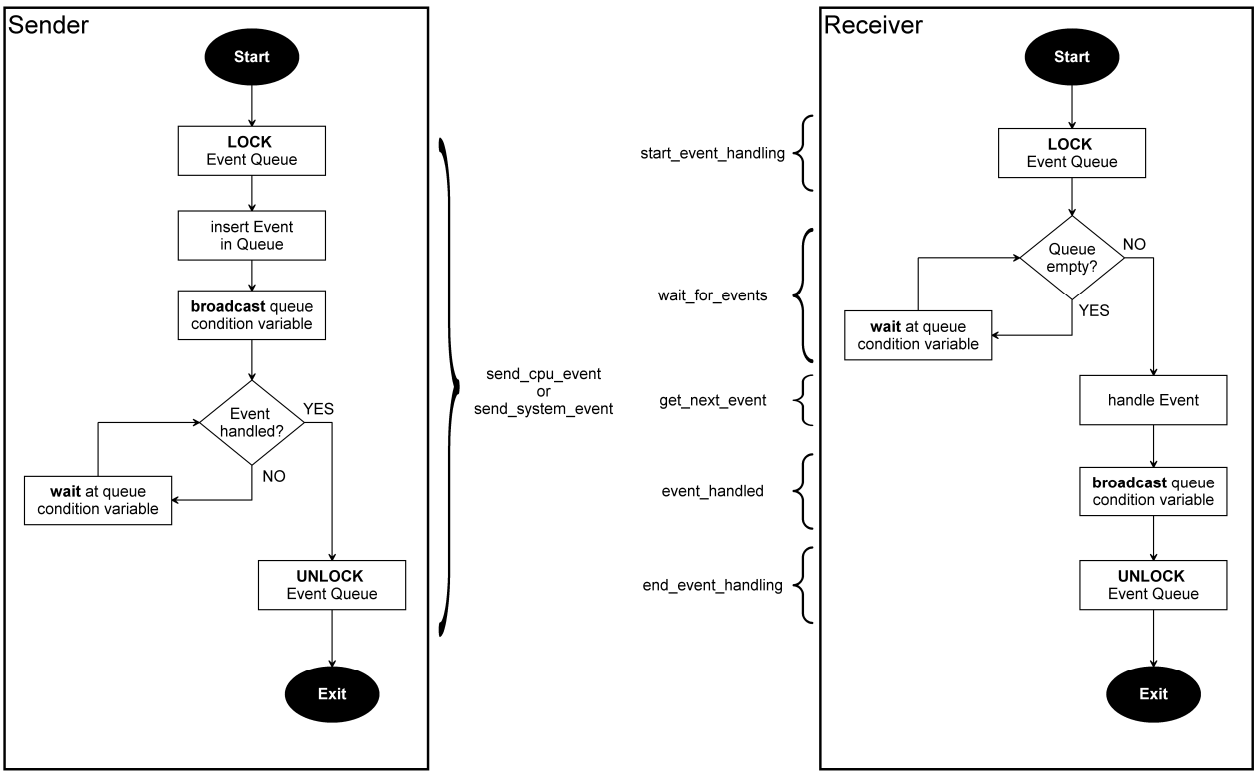


Figure 4.7: Detailed view inside the Simulation Event Manager

Figure A.1 and A.2 in Appendix A show an example implementation for Sender and Receiver.

4.1.4 Detailed Parallel Simulation Loop of System and Processor

After explaining single components of the decoupled main loops, the following shall now provide an overview of the complete picture. Figure 4.8 and 4.9 display the detailed main loop of System and Processor.

The Processor's Main Loop consists of two parts. Box ① and ② on the right side show the state handling whereas the logic in box ③ on the left is responsible for state changes. Blue boxes indicate a direct state change and the green box represents a target program execution cycle. The control flow starts with an initialization part on top, where the first state is set to RUNNING, and enters the loop. The first operation on entering the loop is deciding if a state change action is present or not. If there is one, control flow goes to the left. In box ③ the state change action is evaluated and executed. Every change has a function like *pause_cpu* or *continue_cpu*, which stops or restarts timers or logs current statistics. If the state is set to STOPPED, the main loop exits. Every other state change returns to the loop entrance. If no change is pending, the control flow continues on the right side. Box ① indicates the RUNNING mode, where the target program is executed, and box ② encompasses the PAUSED state, where the CPU goes to sleep by waiting at a sleep condition variable (SCV).

At first, in RUNNING state, the simulator checks the last *step result*, which is set for each target program execution step. Depending on the last executed instructions the Processor continues or sends a PAUSE or STOP request using the Event Manager. Handling the sleep mode is protected by a sleep mutex (SM) and an additional state change action check-up. This assures that the Processor does not miss any wake-up call. Without this protection it is possible that a wake-up call with its corresponding state change is set, while the Processor's control flow has already passed the decision in the middle but not yet reached the SCV. A deadlock would be the result. Now the System has to acquire the SM before setting a state change. If the Processor has already reached the SCV, the SM is free. If the CPU is in between, the System blocks until the CPU goes to sleep.

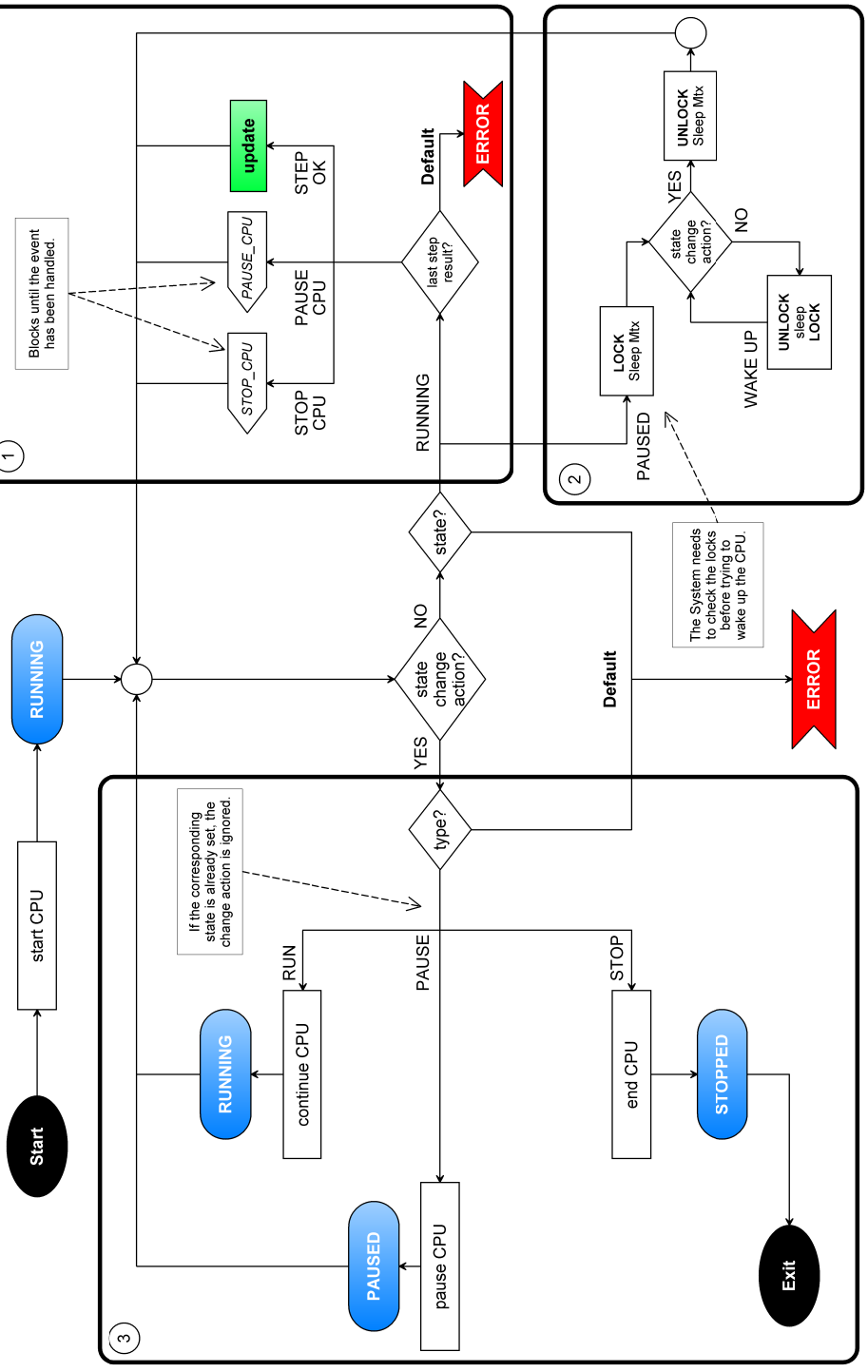


Figure 4.8: Detailed diagram of the parallel Processor simulation loop.

The Systems's Main Loop can also be split up into two different parts. The upper part in box ① indicates sleeping at the event queue, while box ② encompasses event handling depending on the given event reason. Access to the Event Manager interface is formatted in italic letters.

As before, it all starts with an initialization phase, where all Processor threads are spawned and started. Upon entering the main loop in box ①, *start_event_handling* is called, which locks the event queue. If no event is pending, the System goes to sleep with *wait_for_events*. It awakes upon incoming events and handles them by calling *get_next_event*.

Box ② shows the control flow for different events. Depending on the given reason, a state change action is set in the corresponding CPU's state machine. A SYS_STOP_ALL event is an external event and invokes a STOP action for all CPUs. An action *Set* always checks, if the corresponding CPU is in the PAUSED state and wakes it up, if necessary. Afterwards, the Event Manager is notified by marking the event as handled. If all Processors have been told to stop, the main loop is left and the ARCSIM terminates after all CPUs have reached their STOPPED state.

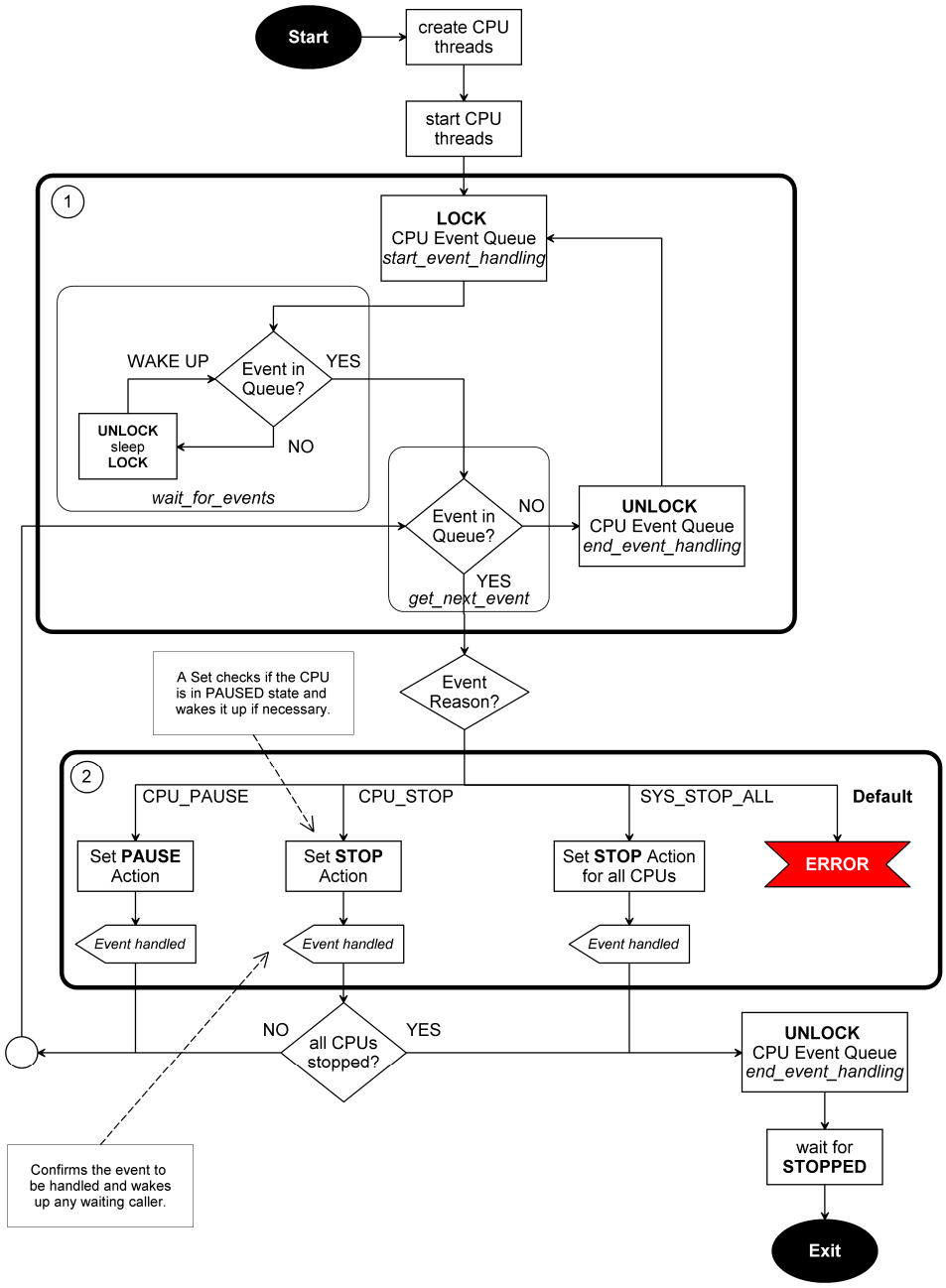


Figure 4.9: Detailed diagram of the parallel System simulation loop.

4.1.5 Synchronizing Shared Resources

Since multiple Processor instances are now working concurrently, all of them need access to shared resources. In order to avoid errors and race conditions, this access needs to be synchronized.

ARCSIM's architecture model contains a global memory, which among other things holds the target program. It is accessed on a per page basis, where every memory page has a size of 8 KB. Each Processor instance has a private page cache, where recently used memory pages are stored for fast access. The page cache is also helpful in avoiding repeated translations of target memory addresses to host memory addresses, which is quite costly (see Section 3.3.4 for details about ARCSIM's Processor caches). If a page can not be found in the private page cache, it is requested from the global memory by a call to *get_host_page*. This method returns a reference to a region of allocated memory. If it is not yet allocated, this will be done at the first access. The given reference is then stored in the page cache, ready to be accessed. In order to support multiple calls, the method *get_host_page* has been synchronized with a *Scoped Lock*.

This synchronization method is designed analogously to JAVA synchronized methods and is based on a mutex. It works by creating a local Scoped Lock object at the method's entrance. The constructor expects a mutex, which is acquired immediately. If a different thread is already using the method, this mutex will be blocked. Upon method exit, the local lock object's destructor is called automatically, which releases the mutex. (see Figure A.3 in Appendix A for a Scoped Lock example)

As mentioned in Section 3.3, ARCSIM supports simulation of external devices. These devices use Memory Mapped IO, which means regions of the architecture's memory are reserved for device specific data. If the simulation loop or a Processor wants to access a device, it writes to or reads from the corresponding memory region. If a target program is, for example, using graphical output, corresponding data is written to memory and displayed by a screen device running in a separate thread. Since memory access is already synchronized, access to external devices does not need additional work. Of course, external devices need to be adapted for multiple cores. The screen device, for example, is able to show the output of multiple cores.

A rather basic but nevertheless important synchronization is used to display ARCSIM's trace output. Depending on the configured debug level, more or less tracing information is printed to the terminal. By using a synchronized string stream, all threads can log concurrently, and a small header indicating the currently printing CPU helps to identify messages.

4.2 Optimizing the Multi-Core Simulator

The previous section showed how the ARCSIM simulator was extended to enable the simulation of multiple cores by separating the main loop, implementing an efficient and flexible communication system and synchronizing shared resources. The following section will give a detailed overview of the additional optimizations that have been performed for ARCSIM's High-Speed mode in order to achieve high multi-core simulation performance.

4.2.1 Centralized JIT DBT Compilation

Section 3.3.3 explains the functionality of single-core ARCSIM's High-Speed mode. The Processor's main loop interprets a target application and collects statistics on executed code traces. If a code trace is executed frequently, it is recognized as "hot" and dispatched to a parallel JIT DBT task farm, where a translation worker thread translates the hot trace to native code. As soon as a translation is available for the currently interpreted trace, ARCSIM switches to native mode and executes the native code directly, which results in a significant speed-up.

When the simulator executes a multi-threaded target binary, application threads are assigned to the cores of the target architecture model. It can now happen that the same code trace is interpreted by different Processor instances, especially for data parallel applications. Consider an algorithm which transforms a matrix. To accelerate the calculation, each row of the matrix is processed by a different thread. It is now possible that the same calculation is applied multiple times to each row but by different cores. If every Processor has its own JIT DBT translation workers and translates hot traces locally, lots of redundant translation work would be performed since the code trace corresponding to the matrix row calculation is translated for each core separately. Therefore, an optimization is to share translated code traces between different CPUs by using a single centralized JIT DBT Task farm and multi-level translation caches.

In addition to the private translation cache per core, another cache is added to the architecture which is only accessible by translation worker threads. This so called second level cache holds recently translated code traces of *all* participating cores in contrast to the private caches. It helps to avoid redundant translation work and is used to share translations between cores. To eliminate duplicate work items in the translation process, every item is added to a hash table. Every core can now register interest in a specific translation in this hash table if a corresponding work item is already in progress. These two mechanics are explained in detail in Section 4.2.2 and 4.2.3.

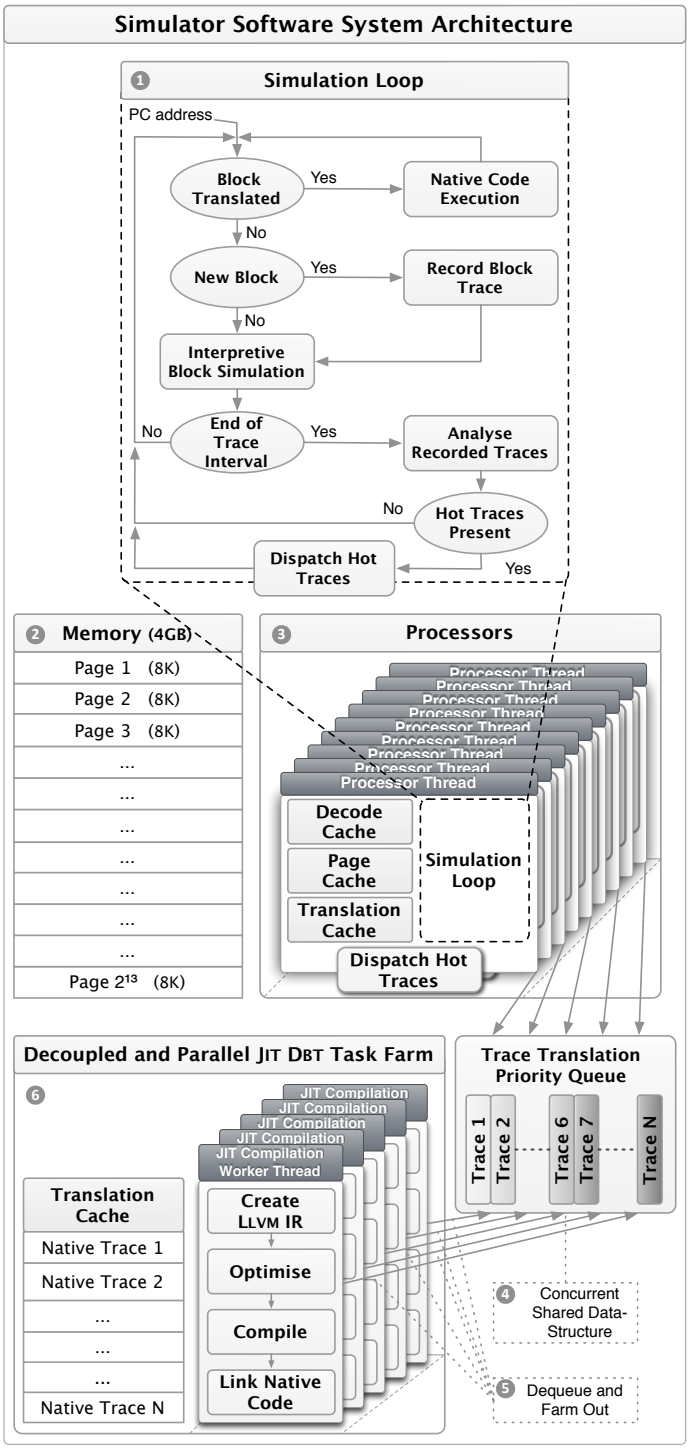


Figure 4.10: Software architecture of the multi-core simulation capable ISS using parallel trace-based JIT DBT.

Figure 4.10 shows the software architecture of the multi-core simulator and how single components interact during High-Speed simulation. The functionality of a single interpreter instance is represented by the flow chart in the upper part labeled ①. The interpreter's functionality does not change for multi-core simulations, and therefore it continues to work as described in Section 3.3.3. Encountered code regions are either interpreted or executed natively, if a translation is available. At the end of each trace-interval, hot translations are dispatched to a Trace Translation Priority Queue ④.

This queue, however, is now a shared data-structure, which is accessible by all concurrently running interpreter instances inside the different Processor threads ③. To make sure the most important traces are translated first, a priority queue is used. Importance of a translation work item is determined by a combination of execution frequency and time since the last execution of the corresponding code trace.

A single centralized JIT DBT Task farm labeled with ⑥ in Figure 4.10 translates hot traces. Corresponding translation workers run in separate threads, parallel to the Processor simulation loops and the System thread. One major benefit of this architecture is that compilation latency is hidden by performing it in parallel with the main simulation loops of the CPUs.

As soon as a translation work item has been dispatched to the priority queue, it is dequeued by the next available translation thread ⑤. During native code generation, the same translation steps are performed as described in Section 3.3.3. Target instructions are first mapped to C-code functions, transformed into the LLVM intermediate representation, optimized by standard LLVM compiler passes and eventually compiled and linked to native code.

Like the single-core High-speed mode, the multi-core version uses adjustable translation units and a light-weight translation scheme. This scheme only records basic block entry points as nodes and pairs of source and target entry points as edges to construct a CFG.

4.2.2 Multi-Level Translation Cache

Since there are now centralized translation workers, it is possible to share translations between cores and avoid repeated translation of identical traces. This goal is achieved by using a *multi-level translation cache*.

Translated code traces can be identified by the address of the basic block, which is the trace's entry point. Each Processor has a private translation cache, where recently executed and translated code traces are stored for quick access. If the in-

terpreter of a core encounters a new basic block, it looks up the corresponding address in the private translation cache. If an entry matches the basic block's physical address, a function pointer is returned which can be called to execute the corresponding translation in native code (see ① in Figure 4.10).

The second level cache is a data structure that is only accessible by the JIT translation workers (see ⑥ in Figure 4.10). Every time a worker finishes translation of a work unit, it registers the native trace in the second level cache. Afterwards, it is also registered inside the private first level cache of the processor which requested the translation. As soon as another CPU dispatches an identical work item, the responsible JIT worker checks the second level cache for already existing translations. If a matching entry is present, the corresponding translation is registered directly for the requesting CPU. That way translations between different cores can be shared and redundant compilation work is avoided.

Depending on the chosen granularity of translation units, determining if a translation is in the second level cache may be more complicated than only checking if the physical addresses match. If the simulator uses page-based translation units, different cores can take different paths through the CFG, even if they start at the same entry point. The corresponding native code would be different between those two traces. To solve that problem, a *fingerprint* is created for the trace and additionally used to identify it. The next section gives more details about that mechanism.

Figure 4.11 shows the algorithm using the second level cache in pseudo code notation. The upper half (line 1 to 14) is executed by the Processor class, whereas the lower part (line 16 to 30) belongs to the TranslationWorker class. After fetching the next trace in line 2, the Processor checks the private translation cache, to see if a translation is present for the current trace. If there is one, it is retrieved from the cache in line 6 and executed in line 12; if not, a work item is created in line 8 and added to the translation work queue.

The translation worker waits for upcoming tasks in line 17, wakes up as soon as a new item is present and fetches it from the translation work queue in line 19. Before the translation process starts, the worker checks the second level cache in line 22 if a translation for the given trace is registered already. If there is one, it is retrieved from the cache in line 23; if not, the item is translated in line 25 and registered as a new item in the second level cache in line 26. Finally, the corresponding translation is registered for the requesting core in line 29.

```
1  CodeTrace trace = get_next_trace();
2  Translation trans = Null;
3  if (translationCache.has_translation(trace.id)) {
4    trans = translationCache.get_translation(trace.id);
5  } else {
6    WorkItem item = create_work_item(trace);
7    add_item_to_queue(item);
8  }
9
10 if(trans = Null) execute_translation(trans);
11 else interpret_trace(trace);
12 ------------------------------------------------
13 wait_for_work();
14
15 WorkItem item = fetch_item_from_queue();
16 Translation trans = Null;
17
18 if (2ndLevelCache.has_translation(item.id)) {
19   trans = 2ndLevelCache.get_translation(item.id);
20 } else {
21   trans = translate_item(item);
22   2ndLevelCache.add_translation(trans);
23 }
24
25 item.core.add_translation(trans);
```

Figure 4.11: Algorithm implementing second level cache. The upper half is part of a Processor and the lower half belongs to a Translation Worker.

Figure 4.12 shows an example of the usage of the second level translation cache. The initial situation in picture ① is as follows: Two traces A and B have already been translated. Therefore, an entry for each of them exists in the second level cache. The translation task for A was dispatched by CPU 0 and the task for B by CPU 1. As a result, the private caches of those two cores have a corresponding entry as well.

In the first picture, CPU 0 encounters the same trace B that has already been translated for CPU 1. Since there is no entry in CPU 0's private cache, a task B is dispatched and enqueued. In picture ② Worker 0 is free and dequeues the new task. Before translating trace B, Worker 0 checks the second level cache for already existing translations ③. Since B was translated already for CPU 1, the worker discards the task and the private cache of CPU 0 is updated with the translation for B from the second level cache ④.

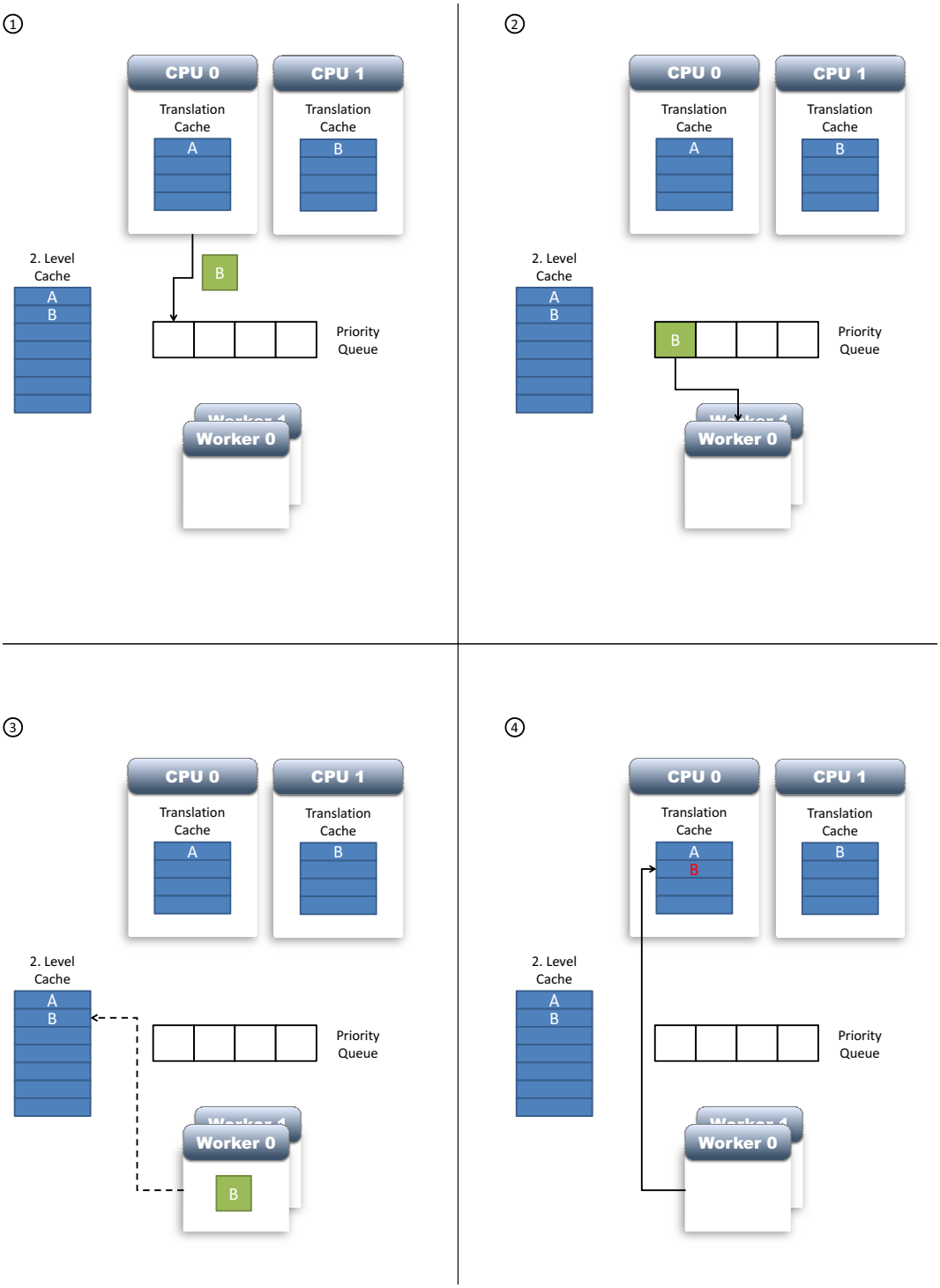


Figure 4.12: Using a second level cache to avoid repeated translations.

4.2.3 Detection and Elimination of Duplicate Work Items

Using a multi-level translation cache avoids repeated translation of work items, which have already been translated. But what happens if two or more cores add translation tasks directly after each other? Assume CPU 0 dispatches a trace A, which has not yet been translated for any core. Almost at the same time CPU 1 dispatches an identical trace, while A from CPU 0 is still in translation process. In that case no translation for the task from CPU 1 is present in the second level cache and it is added to the queue as well. In this case, the same translation would be performed twice. To detect and eliminate those duplicates, the trace fingerprint mentioned in the previous section comes into play.

A trace fingerprint is a unique[1] identifier generated by a hash function, which processes the physical addresses of all basic blocks along the trace. This mechanic assures that, even with page-based translation unit sizes, different paths through a memory page can be identified. This fingerprint is used to determine whether two tasks are the same trace or not.

A hash table stores lists of all tasks which have a particular key as their fingerprint. When a translation task is being dispatched by a core, the hash table is checked, whether an entry for the corresponding fingerprint already exists. If an entry is present, it means an identical task is already in the process of being translated. Therefore, the new task is added to the hash table entry's list and not enqueued in the priority queue. As soon as the identical task has been translated, all CPUs listed for the fingerprint in the hash table are updated.

This technique, in addition to the shared caching of translations described in Section 4.2.2, has the dual effect of reducing the waiting period between the dispatch of a task and the receipt of its translation for many cores. It also reduces the amount of similar tasks in the work queue, resulting in a greater percentage of the simulated code being translated earlier.

Figure 4.13 shows the algorithm from Figure 4.11 extended with the mechanic to detect and eliminate duplicate work items. If the Processor does not find a translation for the current trace in the private cache, it creates a work item as indicated by line 7. Before the item is added to the work queue, an additional check has been added. The hash table holding duplicate items is checked in line 8 for an existing entry matching the fingerprint of the current trace. If an entry is registered, the Processor adds itself to the entries list in line 10 and discards the work item in line 11; if no entry is present, a new one is created in line 13, with the fingerprint of the current trace and the Processor as the first entry in the corresponding list. Only in

[1] Since the fingerprint is created by a hash-function using physical block addresses, it is not guaranteed to be unique, but the probability is high.

```
1  CodeTrace trace = get_next_trace();
2  Translation trans = Null;
3  if (translationCache.has_translation(trace.id)) {
4    trans = translationCache.get_translation(trace.id);
5  } else {
6    WorkItem item = create_work_item(trace);
7    if(duplicateTable.has_entry(item.id)){
8      DupEntry entry = duplicateTable.get_entry(item.id);
9      entry.add_core(core);
10     discard_work_item(item);f
11   } else {
12     duplicateTable.add_entry(item, core);
13     add_item_to_queue(item);
14   }
15 }
16
17 if(trans = Null) execute_translation(trans);
18 else interpret_trace(trace);
19 ------------------------------------------
20 wait_for_work();
21
22 WorkItem item = fetch_item_from_queue();
23 Translation trans = Null;
24
25 if (2ndLevelCache.has_translation(item.id)) {
26   trans = 2ndLevelCache.get_translation(item.id);
27 } else {
28   trans = translate_item(item);
29   2ndLevelCache.add_translation(trans);
30 }
31
32 DupEntry entry = duplicateTable.get_entry(item.id);
33 foreach(Processor core in entry.cores){
34   core.add_translation(trans);
35 }
36 duplicateTable.remove_entry(item);
```

Figure 4.13: Algorithm for second level cache and detection of duplicate work items. The upper half is part of a Processor and the lower half belongs to a Translation Worker.

that case is the work item dispatched to the queue, as seen in line 14.

The translation worker did not require much modification. The additional functionality can be seen beginning in line 35. As soon as a translation is present, either retrieved from the shared second level cache or freshly created, the worker fetches an entry from the hash table matching to the fingerprint of the translated trace. In line 36 and 37 the private caches of all cores listed in the hash table's entry are updated with the translation. Afterwards, the entry is removed from the duplicate hash table, as the translation task is completed.

Figure 4.14 shows an example of the usage of the fingerprint to avoid identical work items. The initial situation in picture ① is the same as at the end of Figure 4.12. Additionally a hash table has been added on the right side, which has a fingerprint as the key and CPUs as corresponding task entries.

In ① CPU 0 dispatches a task C to be translated. Since there is no entry for the fingerprint of C in the hash table, a corresponding one is created and the task from CPU 0 is appended to the list. In picture ②, CPU 1 dispatches an identical task C. Before the task is enqueued, the hash table is checked for already existing fingerprints matching the trace of C. Since there is one present, the task from CPU 1 is discarded and CPU 1 is added to the list. The next free worker fetches task C from the queue and translates it, since there is no entry in the second level cache ③. In the last picture ④, the translation is done and the second level cache is updated. After checking the hash table, the private caches of CPU 0 and CPU 1 are updated as well.

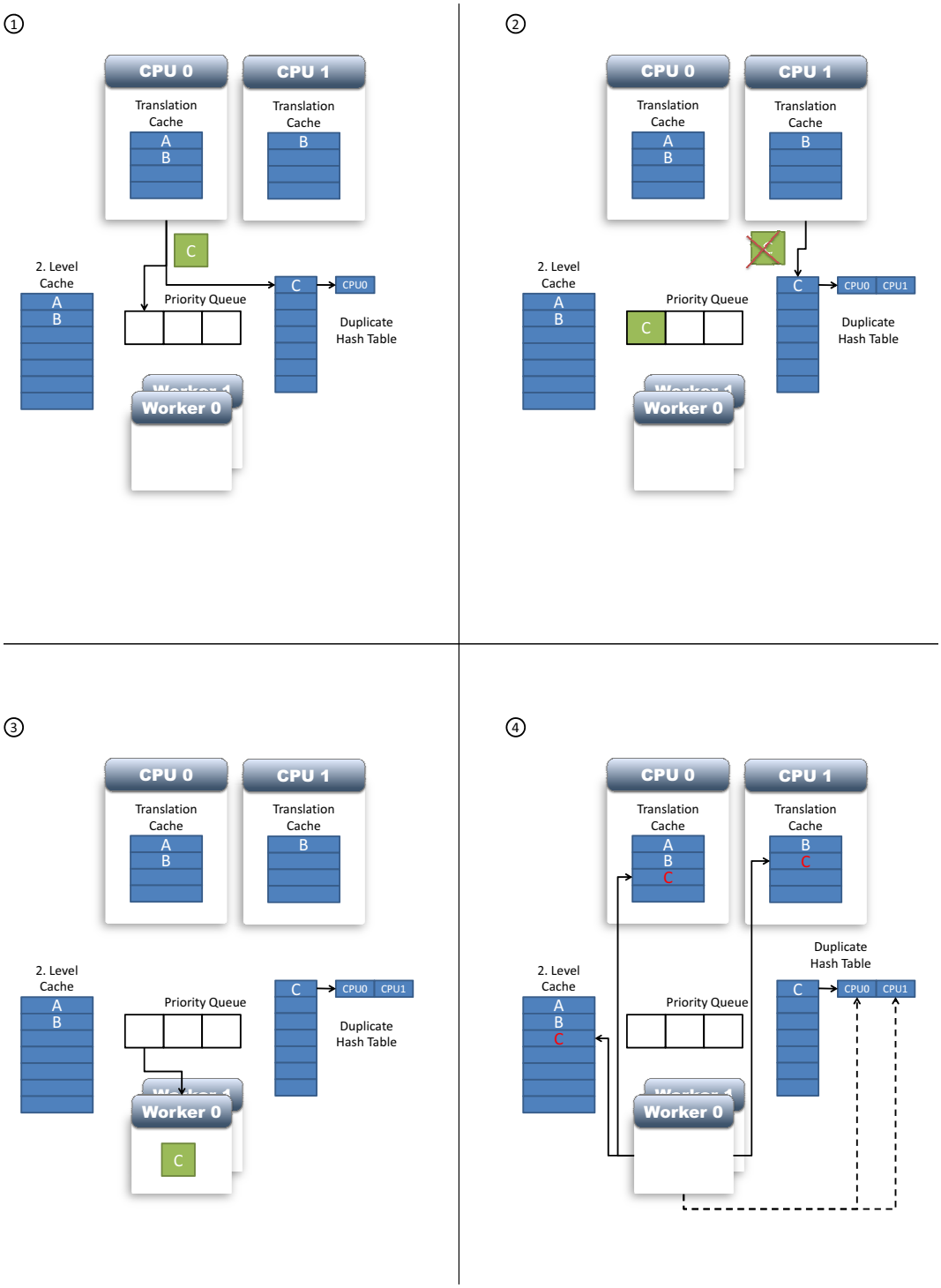


Figure 4.14: Using a trace fingerprint to avoid identical work items.

4.3 A `Pthread`-Model for `libmetal`

In order to run multi-core binaries, the bare-metal library `libmetal`, as introduced in Section 3.3.6, has been extended. Aside from a basic multi-core mechanic to access the current core ID and the overall number of cores, an implementation of the POSIX thread standard interface has been created (see 3.4 for more details about `pthreads`). The interface allows standard multi-core benchmarks like SPLASH-2 to be built and executed without changes to their program code.

4.3.1 `libmetal` for Multiple Cores

To allow a target program access to the core ID and the overall number of cores, an additional auxiliary register has been added to ARCSIM's Processor model. The 16 bit register holds the core ID of the corresponding CPU in the first 8 bits and the overall number of cores in the remaining ones. Two functions, *get_cpu_num()* and *get_num_procs()*, are provided by the `libmetal` library and can be called from a target program.
The implemented `pthread` interface fully supports thread creation and joining, as well as the usage of `pthread` mutexes, condition variables and semaphores. However, the implementation is simplified in terms of number of threads allowed per core. Since the implementation of a full preemptive scheduler would have exceeded the scope of this study, only one thread per core is allowed. This approach is permitted by the `pthread` specification and therefore POSIX compliant. If a new thread is created, it runs on its own simulated core. If no core is available, thread creation fails. Mutexes are implemented using spin locks based around the ARCOMPACT atomic exchange instruction explained in the next section.

4.3.2 Atomic Exchange Operation

The Atomic Exchange instructions in the ARCOMPACT instruction set are utilized for explicit multi-core synchronization. They are intended to atomically exchange a value in register with a value in memory. This is used during multi-core synchronization to avoid context switches, for example, with a lock, while a core is in a critical section of the target program. In ARCSIM, this is implemented using a global simulator lock for all atomic exchange instructions, to accurately model the hardware effects of the instruction. This means that the underlying `x86` hardware synchronization instructions are used to implement the atomic exchange instruction; in practice, it maps to a single `x86` instruction and can therefore be executed very quickly.

Chapter 5

Empirical Evaluation

The parallel JIT DBT multi-core simulator has been evaluated on over 20 benchmarks form EEMBC's MULTIBENCH and Standford's SPLASH-2 benchmark suites (see Section 3.5 for details about the suites). This chapter describes the experimental approach and presents the results for speed and scalability of the extended ARCSIM simulator.

5.1 Experimental Setup and Methodology

The simulator has been evaluated against all 14 embedded application kernels provided by the MULTIBENCH 1.0 benchmark suite. Each application kernel is executed as a separate workload with a varying number of JIT translation worker threads. Some of the kernels are not multi-threaded, so in these cases a separate instance of the kernel is executed on each of the simulated cores.

The SPLASH-2 benchmark suite comprises twelve benchmarks, however, only ten were used for the evaluation. It was not possible to build versions of `fmm` and `water-nsquared` which could run on 64 cores, so these are excluded from the results. In the case where contiguous and non-contiguous versions of the same benchmark were available, the contiguous version was built.

The results are reported in terms of MIPS achieved by the simulator. Each simulated core can calculate its own MIPS rate, where the number of instructions the core has executed is divided by the length of time from when the core itself starts to execute instructions, and when the core is halted. These individual MIPS rates are summed to provide the total simulation rate.

Consider the following calculation for a number of n simulated cores, where R_{total} is the overall MIPS rate of a simulation run and r_x is the MIPS rate of core x:

$$R_{total} = r_0 + r_1 + ... + r_{n-1} + r_n$$

With t_{end} and t_{start} as start and halt time of core x and i_{total} as executed instructions by core x, the MIPS rate for a single core is calculated as follows:

$$r_x = \frac{t_{end} - t_{start}}{i_{total}}$$

Figure 5.1 shows a calculation example for four simulated cores. All cores start separately from each other at t_{x_start}, run for various amounts of time and end at t_{x_end}. After the simulation each core has a specific amount of executed instructions indicated by i_x in the circles below. These, together with the corresponding core execution times, lead to a core MIPS rate of r_x and their accumulation results finally in a total simulation MIPS rate of R_{total}.

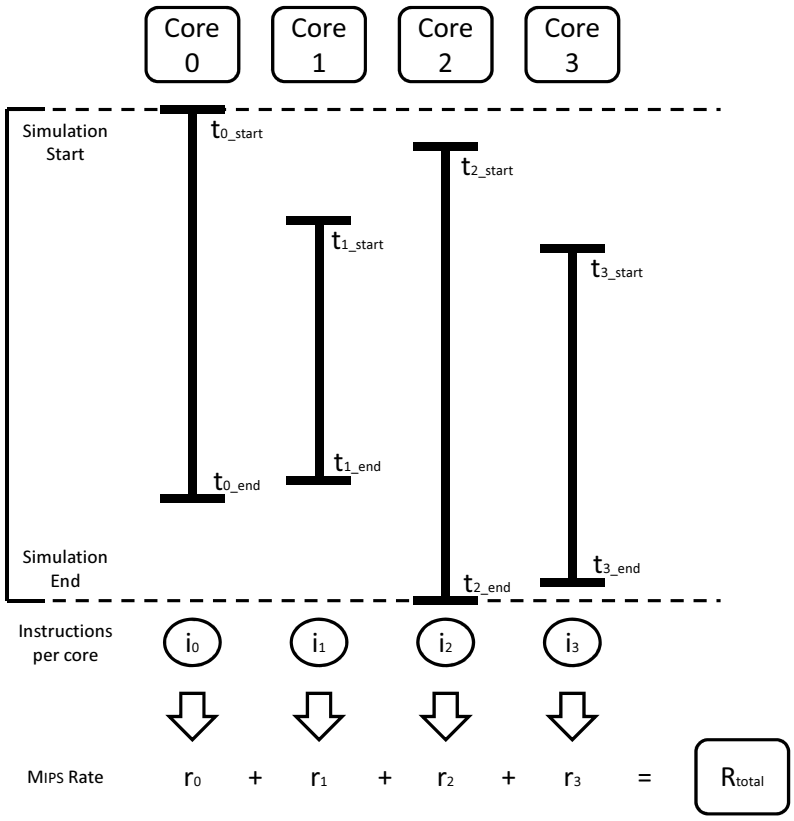


Figure 5.1: Calculation example of the total simulation MIPS rate for four simulated cores.

Three different test sets are presented in this study to evaluate ARCSIM's behavior concerning the following specific properties:

MIPS rate It is demonstrated for all 24 benchmarks, how their MIPS rate changes as the number of cores used for simulation increases from 1 to 64.

Speed-up The speed-ups for each benchmark and core configuration over the single-core configuration are shown, in terms of elapsed time between the point where all cores are ordered to start executing and when all cores have halted. It is presented in form of instruction throughput relative to single-core execution.

$$Throughput = \frac{\text{MIPS}_{multi-core}}{\text{MIPS}_{single-core}}$$

Scalability Three benchmarks were chosen from each of the two suites, and it is shown how their MIPS rate changes as simulated cores are scaled up to 2048 in the simulator.

Each benchmark and core configuration was run 5 times, with the arithmetic mean taken from these runs to present the results.

The system used to run the simulator was an `x86` host with 4 Intel Xeon L7555 1.87 GHz (8-core) processors with hyper-threading disabled, resulting in 32 host cores being made available to the openSUSE 11.3 Linux operating system. The system also had 64GB of RAM available, and all experiments were run under conditions of low system load.

5.2 Bare-Metal POSIX Multi-Threading Support

As mentioned in Section 3.3.6 and 4.3, the light-weight `libmetal` library was used, which provides the essentials of operating system functionality to run the benchmarks on bare-metal hardware and within the simulator. This library provides features such as startup code, I/O device management, memory management primitives, and in particular basic multi-threading support in the form of a `pthreads` API.

The restrictions for `libmetal's` thread implementation is permitted by the `pthreads` specification but differs from the approach taken, for instance, by UNISIM [71] where threading support is emulated by the simulator. This requires applications to be linked against a `pthreads` emulation library which re-routes

`pthreads` API calls to trigger simulator intervention such as suspending or waking up of simulated cores. This study's approach, in contrast, produces binaries that can be run both on real hardware and in the simulation environment without modification.

5.3 Summary of Key Results

The results shown in Figures 5.2 and 5.3 demonstrate that the initial target of "1,000 to 10,000 MIPS" [8] was easily attained, with seven of the benchmarks exceeding 20,000 MIPS when simulating a 32-core target. This is better than the performance of a theoretical 600 MHz 32-core ASIP. For the SPLASH-2 `fft` benchmark a maximum overall simulation rate of 25,307 MIPS is achieved for a 64-core simulation target, whilst on average 11,797 MIPS are provided for the same simulation target. For large-scale configurations of up to 2048 cores the results shown in Figure 5.4 demonstrate the ability of the ARCSIM simulator to scale with the number of processors and to sustain its simulation rate beyond the point at which the number of simulated cores exceeds those of the host system.

5.4 Simulation Speed

All of the MULTIBENCH baseline three-core[1] simulations exceed 1,000 MIPS, with `rgbhpg03` reaching 3,100 MIPS. Due to their higher complexity the single-core performance of the SPLASH-2 benchmarks ranges between 100 to 225 MIPS. On the other hand, they exhibit far greater scalability (see Section 5.5).

Simulating 64 target cores, simulation rates in excess of 20,000 MIPS are achieved for `fft` and `volrend` from SPLASH-2, and for `md5`, `rgbcmyk`, `mpeg2`, `rgbyiq03`, and `rgbhpg03` from MULTIBENCH. Only 5 out of 24 applications fail to deliver more than 3,200 MIPS (equivalent to 100 MIPS simulation rate per host core) while the average performance across all benchmarks for this configuration is close to 12,000 MIPS.

Not all benchmarks maintain this simulation rate as the number of cores increases, showing that simulation performance is application-specific. For instance, the MULTIBENCH networking benchmarks (`ippktcheck`, `ipres`, `tcp`) show little, if

[1] The MULTIBENCH test harness infrastructure runs in separate threads and therefore already needs two cores. This is the reason why core configurations for the MULTIBENCH tests are not always a power of two.

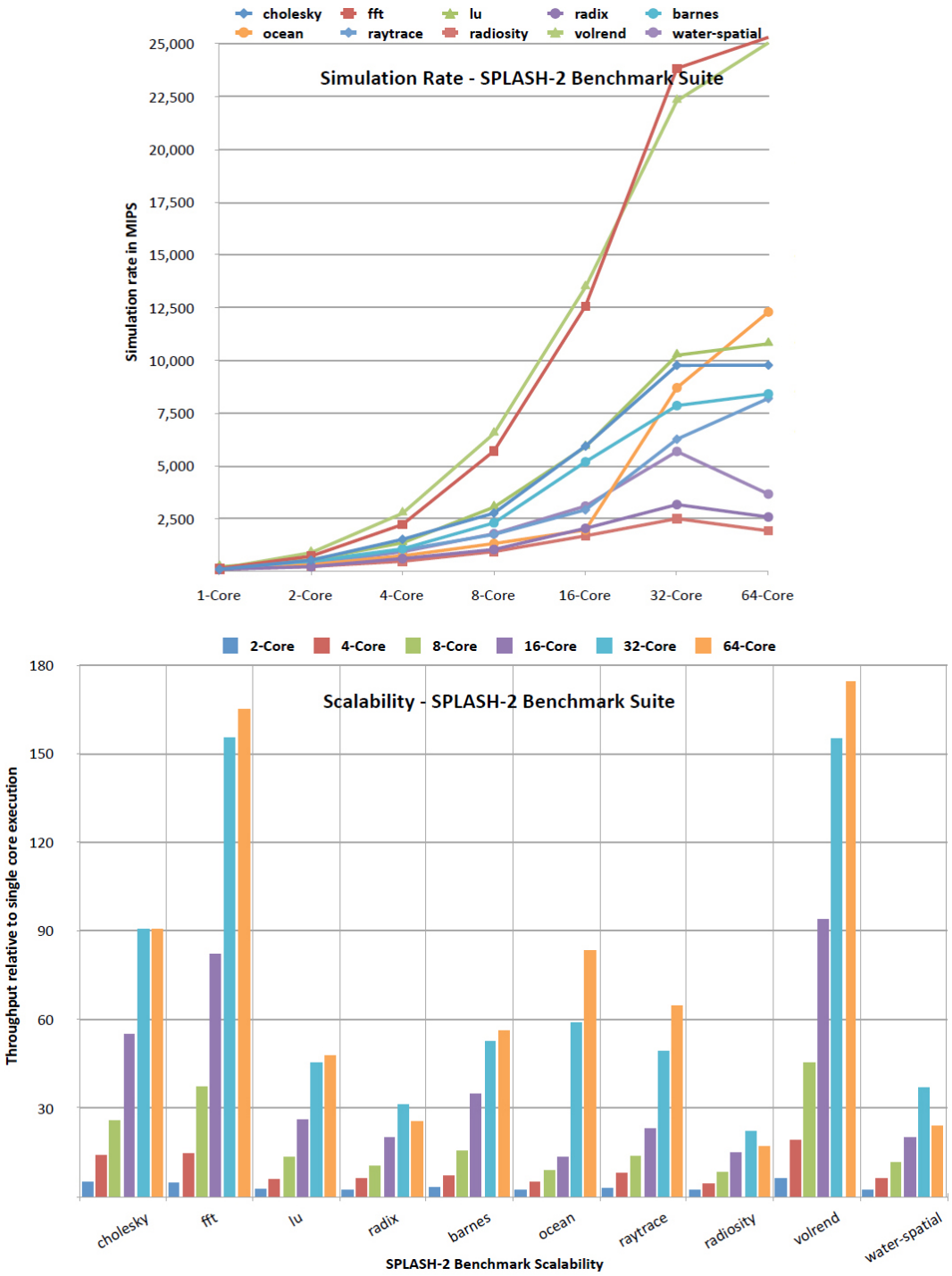


Figure 5.2: Simulation rate in MIPS (top chart) and throughput relative to single-core execution (bottom chart) using the SPLASH-2 benchmark suite for varying multi-core configurations of the ARCSIM ISS.

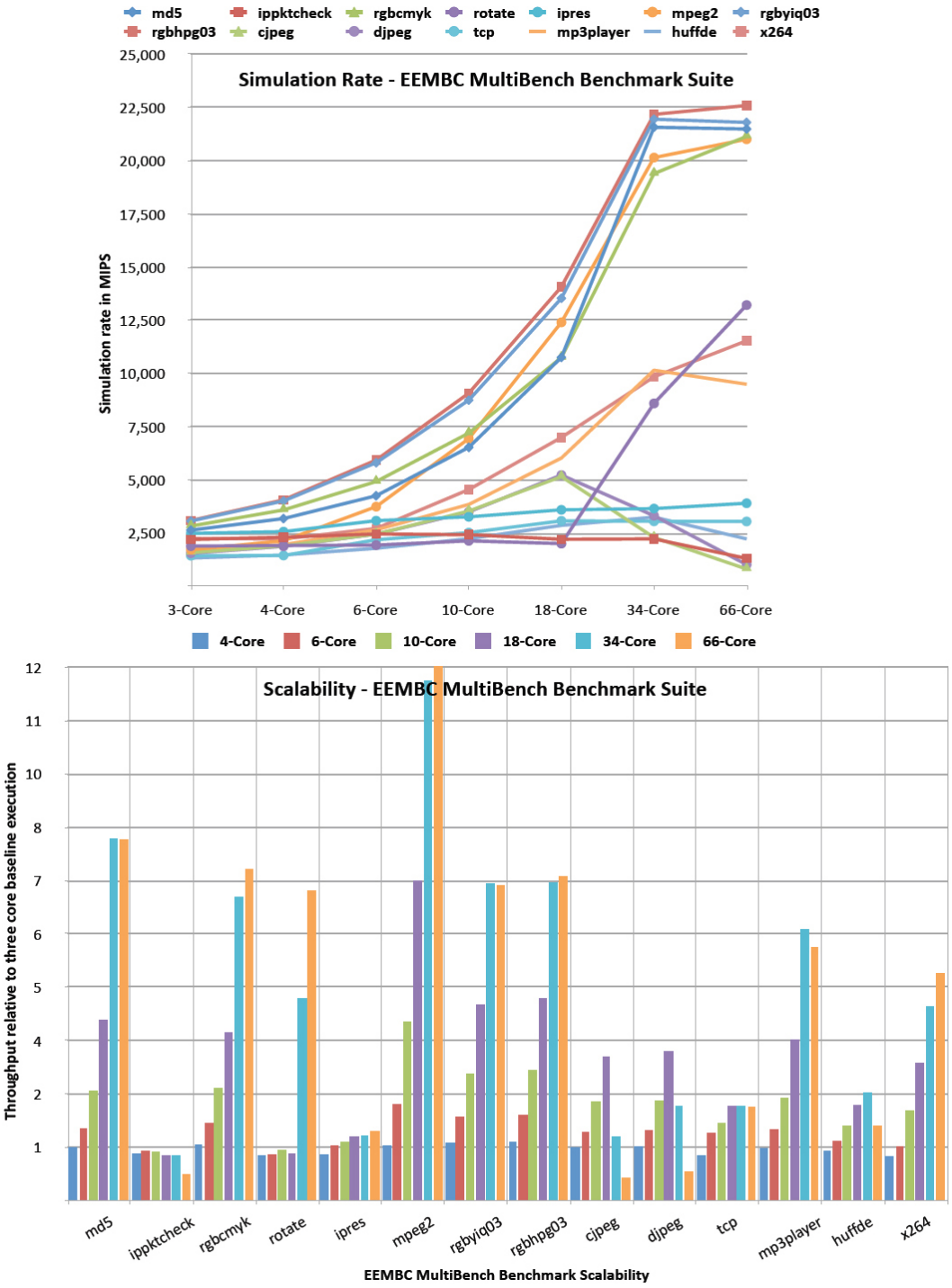


Figure 5.3: Simulation rate in MIPS (top chart) and throughput relative to *three*-core execution (bottom chart) using the EEMBC MULTIBENCH benchmark suite for varying multi-core configurations of the ARCSIM ISS. Note that the minimum MULTIBENCH core configuration is three due to the test harness infrastructure.

any, improvement over the baseline for higher numbers of simulated cores. The profile of the instructions executed by these benchmarks indicates a very high rate of memory accesses and memory-dependent branches which quickly saturate the available memory bandwidth of the host system. These findings are in line with the data sheets provided by EEMBC [66].

5.5 Scalability

It is important to evaluate how the simulator scales beyond the number of host processor cores for simulating tomorrow's many-core systems on today's commodity hardware. Most benchmarks demonstrate that the simulator scales well up to the number of physical cores on the host. Beyond this point occasionally modest further improvements can be seen (e.g. `cholesky`, `lu`, and `md5`) as shown in Figure 5.4.

For the same seven benchmarks that deliver the highest simulation rates, linear scalability can be observed as the number of target cores is increased up to 32. Other benchmarks such as `ocean`, `lu`, `cholesky`, `barnes`, `raytrace`, and `x264` do not achieve such high aggregate simulation rates, but still scale favorably.

For six representative benchmarks (three from each benchmark suite) this study shows scalability up to 2048 simulated target cores in Figure 5.4. Chart ① in Figure 5.4 shows the best result, with `cholesky` continuing to scale from 9,767 MIPS for 32 cores, to 17,549 MIPS for 2048 cores, with the performance always increasing. Chart ④ in Figure 5.4 shows a similar result for `md5`.

In Figure 5.2 super-linear scalability can be seen for a number of benchmarks (e.g. `fft`, `ocean`, `volrend`, `cholesky`). This is due to excessive synchronization in the benchmarks beyond 64 cores, and the fact that the ARCSIM simulator can execute tight spin-lock loops at near native speed. It is a well-known fact that the SPLASH-2 benchmarks attract high synchronization costs for large-scale hardware configurations, as shown by other research [29]. The MULTIBENCH results are not affected in the same way due to less synchronization.

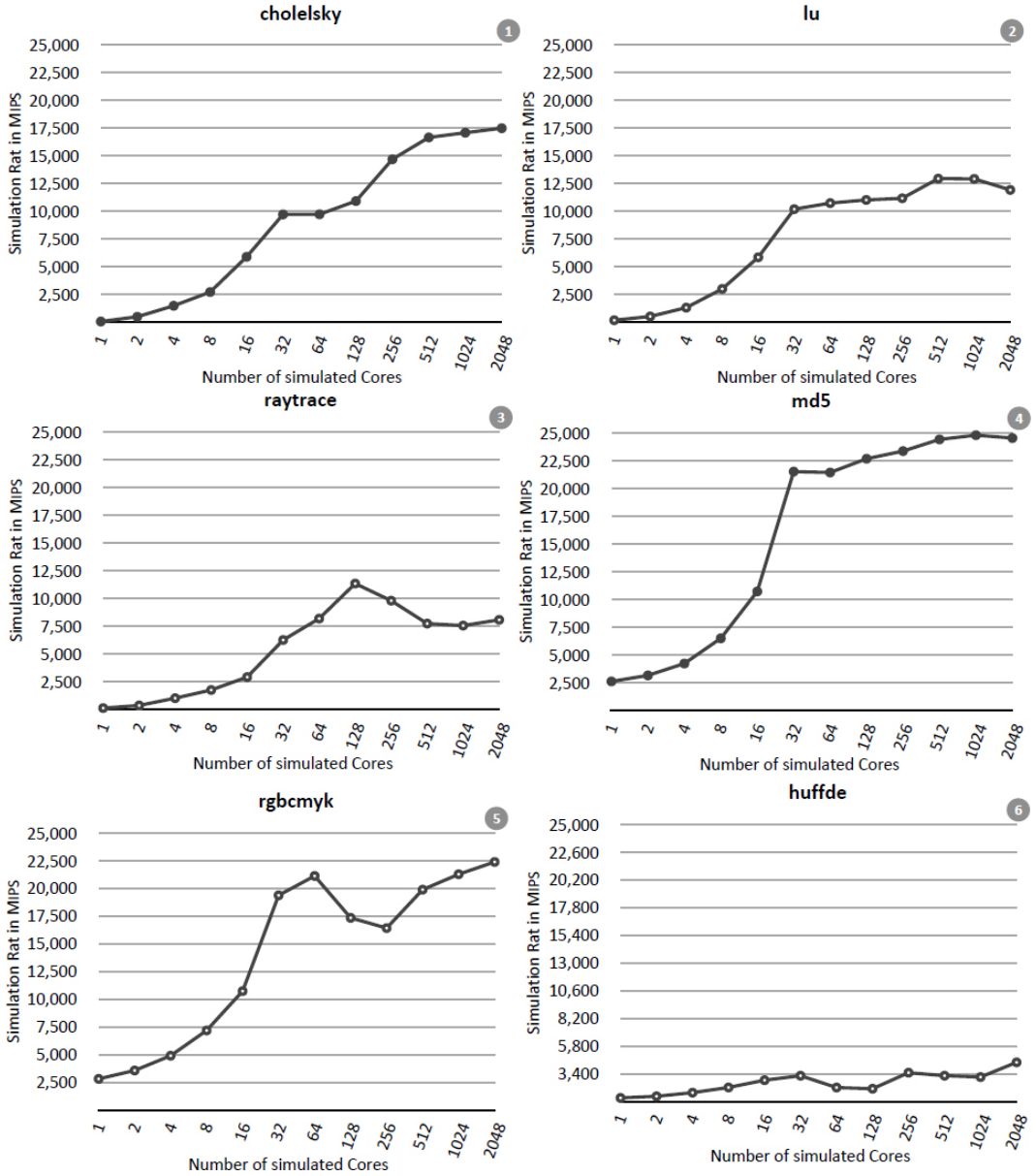


Figure 5.4: Results for selected benchmarks from SPLASH-2 ①②③ and EEMBC MULTIBENCH ④⑤⑥ demonstrating the scalability with the number of simulated target cores.

5.6 Comparison to Native Execution on Real Hardware

Chart ① of Figure 5.5 shows a comparison between the ARCSIM simulator and two hardware platforms (FPGA and ASIP, see Section 3.3.5) in terms of MIPS. The application is a parallelized fractal drawing algorithm, executed across 12 cores. As mentioned in Section 3.5.3, this application was chosen because of its low memory footprint and its embarrassingly parallel nature, thus avoiding application scalability issues.

Actual FPGA performance is 249 MIPS. The performance of a 600 MHz ASIP implementation is also shown which achieves an execution rate of 2,985 MIPS. On the other hand, the instruction set simulator reaches a simulation rate of 6,752 MIPS, thus surpassing a silicon implementation by more than a factor of 2 for this application.

For equivalent configurations, the simulator consistently outperforms the theoretical maximum of the FPGA on a per-core basis. The 12-core FPGA implementation of the multi-core system is capable of 50 MIPS per core. On the contrary, across all benchmarks, the lowest per-core simulation rate for a 16-core target was 105 MIPS, attained in the SPLASH-2 `radiosity` benchmark. These results show that even in the worst case the simulator maintains more than twice the theoretical maximum execution rate of the FPGA. Compared to the average simulation rate of 11,797 MIPS across all benchmarks, the theoretical maximum of 600 MIPS for a 12-core FPGA implementation is an order of magnitude slower.

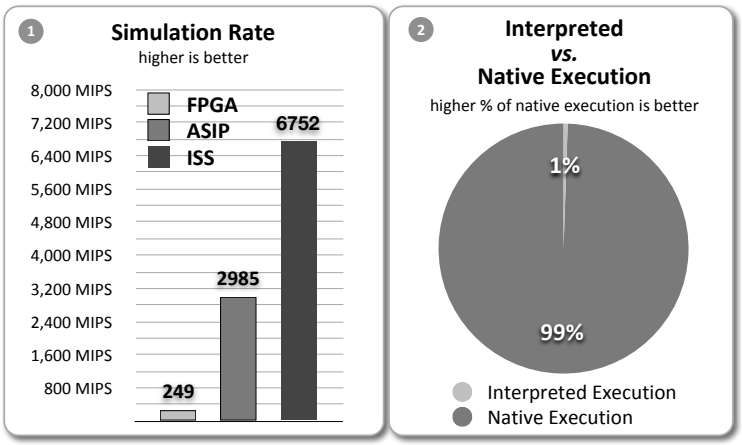


Figure 5.5: Comparison of simulator and hardware implementations. ① shows a comparison of maximum achievable simulation rate in MIPS for a 12-core configuration running a parallel Mandelbrot fractal benchmark on an FPGA, ASIP, and ISS platform. ② depicts the ratio of interpreted vs. natively executed instructions on the ISS platform.

Chapter 6

Conclusions

6.1 Summary and Conclusions

For this thesis an innovative Instruction Set Simulator has been developed, which is capable of simulating multi-core target architectures whilst fully utilizing a multi-core host system. It is based on a carefully designed software architecture, which can handle complex synchronization tasks common to many parallel applications. Every core of the modeled target architecture spawns in a separate thread and is supervised by a main system thread through an event driven communication system.

Various optimizations have been implemented, maintaining high performance and detailed observability utilizing JIT translation mechanics. A centralized JIT DBT task farm runs in parallel with the simulation and provides multiple translation worker threads. The workers are fed with work items dispatched in a shared translation work queue by all participating CPU instances. The dynamic compilation infrastructure uses a hot spot detection mechanism to translate frequently executed code traces to native code in order to accelerate the simulation. A hierarchy of multi-level translation caches helps to avoid redundant work by sharing translated traces between simulated cores. Additionally, a mechanism detects and eliminates duplicate work items in the translation work queue.

An efficient low-level implementation for atomic exchange operations is used to implement a light-weight multi-threading library compatible to POSIX threads, which enables the simulator to run `pthread` target applications without the need to boot an OS or apply any changes to their program code.

The simulator has been evaluated against two industry-standard benchmark suites: EEMBC MULTIBENCH and SPLASH-2 which comprise over 20 benchmarks representing common embedded software tasks and scientific workloads. With its innovative JIT DBT technique utilizing multiple host cores, unprecedented simulator throughput of up to 25,307 MIPS and near-optimal scalability of up to 2048 simulated cores was achieved on a standard 32-core `x86` simulation host. The empirical results show a simulation performance advantage by two orders of magnitude over leading and state-of-the-art FPGA architecture simulation technology [30] for a comparable level of simulation detail.

With the results of the research work done for this thesis, a paper has been published at the IC-SAMOS conference 2011 in Greece:

O.Almer, I.Böhm, T.Edler von Koch, B.Franke, S.Kyle, V.Seeker, C.Thompson and N.Topham: Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation [2011] To appear in Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS'11), Samos, Greece, July 19-22, 2011

6.2 Future work

In future work a detailed interconnection network needs to be implemented to maintain cache coherence between simulated CPUs - the current simulator only works with local timing models per CPU and cache. As introduced in Section 3.3.3 single-core ARCSIM supports multiple different simulation modes. As this thesis has focused on extending the high-speed JIT DBT mode to support multiple cores, future work will aim to extend other modes such as the mode using statistical sampling to drive a cycle-approximate performance model. Single-core ARCSIM is able to simulate the boot-up and interactive operation of a complete Linux-based system. However, the multi-core version still relies on a light-weight multi-threading library providing common operating system tasks but does not support a full SMP Linux.

Appendix A

Code Excerpts

Implementation of a Sender using the Event Manager

```
1  void Sender::main_loop() {
2   SimulationEventManager* event_mgr = get_event_manager();
3   ResultType res = RES_NONE;
4
5   for (;/* ever */;) {
6    res = update();
7
8    // check update results
9    switch(res){
10    case RES_ERROR:
11     // the following blocks until the event is handled
12     event_mgr->send_system_event(EVENT_ERROR);
13    break;
14    case RES_THRESHOLD_REACHED:
15     // the following blocks until the event is handled
16     event_mgr->send_system_event(EVENT_THRESHOLD_REACHED);
17    break;
18   }
19  }
20 }
```

Figure A.1: Sender implementation

Implementation of a Receiver using the Event Manager

```
1  void Receiver::main_loop() {
2   SimulationEventManager* event_mgr = get_event_manager();
3
4   for (;/* ever */;) {
5    // Start handling
6    event_mgr->start_event_handling();
7
8    // The following blocks until we receive an event
9    event_mgr->wait_for_events();
10
11   // Process received events
12   SimulationEventItem* nextItem = NULL;
13   while (event_mgr->get_next_event(&nextItem)) {
14    handle_sim_event(nextItem);        // handle event
15    event_mgr->event_handled(item);   // mark event as handled
16   }
17
18   // End handling
19   event_mgr->end_event_handling();
20  }
21 }
```

Figure A.2: Receiver implementation

Using Scoped Locks to synchronize methods

```
1  class ExternalMemory {
2  private:
3   // Mutex used to synchronize access to memory map
4   //
5   arcsim::util::system::Mutex mem_mtx;
6
7  public:
8
9   (...)
10
11  inline BlockData* get_host_page (uint32 phys_byte_addr)
12  {
13    // ------------------------------------------------
14    // SCOPED LOCK START
15    arcsim::util::system::ScopedLock lock(mem_mtx);
16
17    (...)
18  }
19 }
```

Figure A.3: Scoped Lock Synchronization

Bibliography

[1] B. Gates, "The disappearing computer by bill gates," 2003. [Online]. Available: http://www.microsoft.com/presspass/ofnote/11-02worldin2003.mspx

[2] C. Hughes, V. Pai, and P. Ranganathan. . . , "RSIM: Simulating shared-memory multiprocessors with ILP processors," *Computer*, Jan 2002. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=982915

[3] R. Zhong, Y. Zhu, W. Chen, M. Lin, and W.-F. Wong, "An inter-core communication enabled multi-core simulator based on simplescalar," *Advanced Information Networking and Applications Workshops, International Conference on*, vol. 1, pp. 758–763, 2007.

[4] R. Lantz, "Fast functional simulation with parallel Embra," in *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*, 2008.

[5] J. E. M. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, January 2010.

[6] Synopsys, Inc., "Arcompact instruction set architecture," 700 East Middlefield Road, Mountain View, California 94043, United States. [Online]. Available: http://www.synopsys.com

[7] I. Böhm, B. Franke, and N. P. Topham, "Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator," in *ICSAMOS*, F. J. Kurdahi and J. Takala, Eds. IEEE, 2010, pp. 1–10.

[8] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai, "Protoflex: Fpga-accelerated hybrid functional simulator," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 326, 2007.

[9] R. Covington, S. Dwarkada, J. R. Jump, J. B. Sinclair, and S. Madala, "The efficient simulation of parallel computer systems," in *International Journal in Computer Simulation*, 1991, pp. 31–58.

[10] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of the 2005 USENIX Annual Technical Conference.*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[11] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, and B. Zang, "COREMU: a scalable and portable parallel full-system emulator," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 2011, p. 213–222.

[12] R. Lantz, "Parallel SimOS: Scalability and performance for large system simulation," *www-cs.stanford.edu*, Jan 2007. [Online]. Available: http://www-cs.stanford.edu/~rlantz/papers/lantz-thesis.pdf

[13] 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*, vol. 35, pp. 50–58, February 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=619072.621909

[14] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 31–34, March 2004.

[15] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, pp. 92–99, November 2005. [Online]. Available: http://doi.acm.org/10.1145/1105734.1105747

[16] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52–61, January 2009.

[17] G. Zheng, G. Kakulapati, and L. V. Kalé, "BigSim: A parallel simulator for performance prediction of extremely large parallel machines," *Parallel and Distributed Processing Symposium, International*, vol. 1, p. 78b, 2004.

[18] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter, "FastMP: A multi-core simulation methodology," in *Proceedings of the Workshop on Modeling, Benchmarking and Simulation (MoBS 2006)*, Boston, Massachusetts, 2006.

[19] J. Chen, M. Annavaram, and M. Dubois, "Slacksim: a platform for parallel simulations of cmps on cmps," *SIGARCH Comput. Archit. News*, vol. 37, pp. 20–29, July 2009. [Online]. Available: http://doi.acm.org/10.1145/1577129.1577134

[20] X. Zhu, J. Wu, X. Sui, W. Yin, Q. Wang, and Z. Gong, "PCAsim: A parallel cycle accurate simulation platform for CMPs," in *Proceedings of the 2010 International*

Conference on Computer Design and Applications (ICCDA), June 2010, pp. V1–597 – V1–601.

[21] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The wisconsin wind tunnel: virtual prototyping of parallel computers," in *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '93. New York, NY, USA: ACM, 1993, pp. 48–60.

[22] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus, "Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator," *IEEE Concurrency*, vol. 8, pp. 12–20, October 2000.

[23] M. Chidester and A. George, "Parallel simulation of chip-multiprocessor architectures," *ACM Trans. Model. Comput. Simul.*, vol. 12, pp. 176–200, July 2002. [Online]. Available: http://doi.acm.org/10.1145/643114.643116

[24] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *in Proc. of the Twelfth Int. Symp. on High-Performance Computer Architecture*, 2006, pp. 29–40.

[25] K. Wang, Y. Zhang, and H. Wang. . . , "Parallelization of IBM Mambo system simulator in functional modes," *ACM SIGOPS Operating . . .* , Jan 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1341325

[26] X. Sui, J. Wu, W. Yin, and D. Zhou. . . , "MALsim: A functional-level parallel simulation platform for CMPs," *. . . and Technology (ICCET)*, Jan 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5485532

[27] D. Wentzlaff and A. Agarwal, "Constructing virtual architectures on a tiled processor," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 173–184. [Online]. Available: http://dx.doi.org/10.1109/CGO.2006.11

[28] X. Zhu and S. Malik, "Using a communication architecture specification in an application-driven retargetable prototyping platform for multiprocessing," in *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, ser. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 21 244–. [Online]. Available: http://portal.acm.org/citation.cfm?id=968879.969211

[29] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *SIGARCH Comput. Archit. News*, vol. 37, pp. 10–19, July 2009.

[30] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP gold: an FPGA-based architecture simulator for multiprocessors," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 463–468. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837390

[31] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A case for FAME: FPGA architecture model execution," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 290–301. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815999

[32] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research Accelerator for Multiple Processors," *IEEE Micro*, vol. 27, pp. 46–57, March 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1308421.1308524

[33] D. Chiou, D. Sunwoo, H. Angepat, J. Kim, N. Patil, W. Reinhart, and D. Johnson, "Parallelizing computer system simulators," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1 – 5.

[34] D. Chiou, H. Angepat, N. Patil, and D. Sunwoo, "Accurate functional-first multicore simulators," *IEEE Comput. Archit. Lett.*, vol. 8, pp. 64–67, July 2009. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2009.44

[35] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 15:1–15:32, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1534916.1534925

[36] J. L. Hennessy, D. A. Patterson, and A. C. Arpaci-Dusseau, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2007.

[37] J. Engblom, "Full-System simulation," *European Summer School on embedded Systems.(ESSES'03)*, 2003.

[38] N. Topham and D. Jones, "High speed CPU simulation using JIT binary translation," in *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation (held in conjunction with ISCA-34), San Diego*, 2007.

[39] C. May, "MIMIC: a fast system/370 simulator," in *Papers of the Symposium on Interpreters and interpretive techniques*, 1987, p. 13.

[40] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, p. 13ñ25, 1997.

[41] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, pp. 59–67, February 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=619072.621910

[42] X. Huang, J. E. Moss, K. S. McKinley, S. Blackburn, and D. Burger, "Dynamic simplescalar: Simulating java virtual machines," in *The First Workshop on Managed Run Time Environment Workloads, held in conjunction with CGO*, 2003.

[43] M.-K. Chung and C.-M. Kyung, "Advanced techniques for multiprocessor simulation," *International SoC Design Conference(ISOCC) 2004*, vol. 100, October 2004.

[44] C. Mills, S. C. Ahalt, and J. Fowler, "Compiled instruction set simulation," *Software: Practice and Experience*, vol. 21, no. 8, p. 877ñ889, 1991.

[45] V. Zivojnovic and H. Meyr, "Compiled HW/SW co-simulation," in *Design Automation Conference Proceedings 1996, 33rd*, 1996, pp. 690–695.

[46] M. K. Chung and C. M. Kyung, "Improvement of compiled instruction set simulator by increasing flexibility and reducing compile time," in *Rapid System Prototyping, 2004. Proceedings. 15th IEEE International Workshop on*, 2004, p. 38ñ44.

[47] J. Maebe and K. D. Bosschere, "Instrumenting self-modifying code," *cs/0309029*, Sept. 2003. [Online]. Available: http://arxiv.org/abs/cs/0309029

[48] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 24. New York, NY, USA: ACM, May 1996, p. 68ñ79, ACM ID: 233025.

[49] Y. Nakamura and K. Hosokawa, "Fast FPGA-Emulation-Based simulation environment for custom processors," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E89-A, p. 3464ñ3470, Dec. 2006, ACM ID: 1188411. [Online]. Available: http://portal.acm.org/citation.cfm?id=1188299.1188411

[50] J. D. Davis, L. Hammond, and K. Olukotun, "A flexible architecture for simulation and testing (FAST) multiprocessor systems," in *Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA*, vol. 11, 2005.

[51] 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 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, p. 249ñ261, ACM ID: 1331723. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.16

[52] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, p. 45ñ57, 2002.

[53] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 2003, pp. 84–95.

[54] B. Franke, "Fast cycle-approximate instruction set simulation," in *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, 2008, p. 69ñ78.

[55] D. C. Powell and B. Franke, "Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '09. New York, NY, USA: ACM, 2009, p. 315ñ324, ACM ID: 1629478.

[56] "ARCSIM a flexible ultra-high speed instruction set simulator." [Online]. Available: http://groups.inf.ed.ac.uk/pasta/tools_arcsim.html

[57] "Institute for computing systems architecture, school of informatics." [Online]. Available: http://wcms.inf.ed.ac.uk/icsa/

[58] "PASTA processor automated synthesis by iTerative analysis project." [Online]. Available: http://groups.inf.ed.ac.uk/pasta/

[59] "ENCORE embedded processor." [Online]. Available: http://groups.inf.ed.ac.uk/pasta/hw_encore.html

[60] N. Topham, "EnCore: a low-power extensible embedded processor," Wroclaw, 2009.

[61] I. Böhm, T. J. E. von Koch, B. Franke, and N. Topham, "Parallel trace-based Just-In-Time compilation in a dynamic binary translator," 2010.

[62] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, IEEE/ACM International Symposium on*, vol. 0. Los Alamitos, California, USA: IEEE Computer Society, 2004, p. 75.

[63] D. Jones and N. Topham, "High speed CPU simulation using LTU dynamic binary translation," *High Performance Embedded Architectures and Compilers*, p. 50ñ64, 2009.

[64] "POSIX threads programming." [Online]. Available: https://computing.llnl.gov/tutorials/pthreads

[65] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. O'Reilly Media, Inc., Sept. 1996.

[66] The Embedded Microprocessor Benchmark Consortium, "MultiBench 1.0 Multicore Benchmark Software," 02 February 2010. [Online]. Available: http://www.eembc.org/benchmark/multi_sl.php

[67] T. Halfhill, "Eembc's MultiBench Arrives," *www.MPRonline.com*, vol. The Insider's Guide to Microprocessor Hardware, p. 8, July 2008. [Online]. Available: http://www.eembc.org/press/pressrelease/223001_M30_EEMBC.pdf

[68] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 24–36.

[69] D. H. Bailey, "FFTs in external or hierarchical memory," *The Journal of Supercomputing*, vol. 4, p. 23–35, Mar. 1990, ACM ID: 81781. [Online]. Available: http://dx.doi.org/10.1007/BF00162341

[70] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine CM-2," in *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '91. New York, NY, USA: ACM, 1991, p. 3–16, ACM ID: 113380.

[71] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani, "Unisim: An open simulation environment and library for complex architecture design and collaborative development," *IEEE Comput. Archit. Lett.*, vol. 6, pp. 45–48, July 2007.

Typeset May 6, 2011