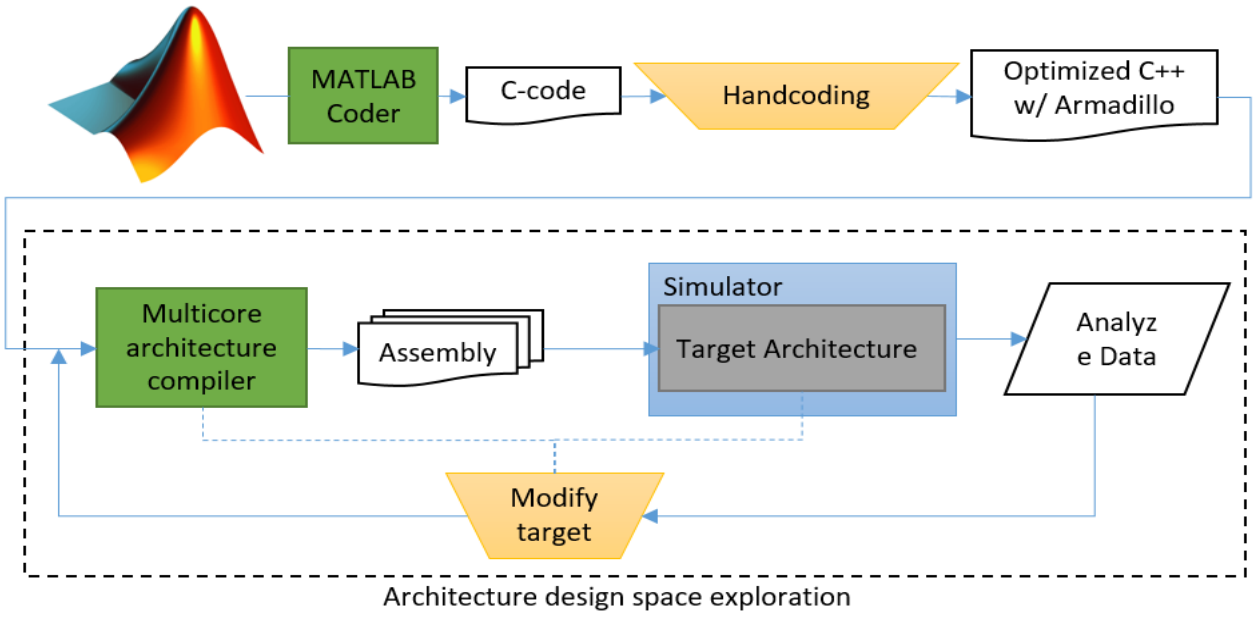
DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2016

Evaluating Gem5 and QEMU Virtual Platforms for ARM Multicore Architectures

JOSÉ LUIS BISMARCK FUENTES MORALES



KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Evaluating Gem5 and QEMU Virtual Platforms for ARM Multicore Architectures

José Luis Bismarck Fuentes Morales

2016-09-25

Master's Thesis

Examiner
Ingo Sander

Academic adviser
Johnny Öberg

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

Abstract

Accurate virtual platforms allow for crucial, early, and inexpensive assessments about the viability and hardware constraints of software/hardware applications. The growth of multicore architectures in both number of cores and relevance in the industry, in turn, demands the emergence of faster and more efficient virtual platforms to make the benefits of single core simulation and emulation available to their multicore successors whilst maintaining accuracy, development costs, time, and efficiency at acceptable levels.

The goal of this thesis is to find optimal virtual platforms to perform hardware design space exploration for multi-core architectures running filtering functions, particularly, a discrete signal filtering Matlab algorithm used for oil surveying applications running on an ARM Cortex-A53 quadcore CPU. In addition to the filtering algorithm, the PARSEC benchmark suite was also used to test platform compliance under workloads with diverse characteristics. Upon reviewing multiple virtual platforms, the gem5 simulator and the QEMU emulator were chosen to be tested due to their ubiquitousness, prominence and flexibility. A Raspberry Pi Model B was used as reference to measure how closely these tools can model a commonly used embedded platform.

The results show that each of the virtual platforms is best suited for different scenarios. The QEMU emulator with KVM support yielded the best performance, albeit requiring access to a host with the same architecture as the target, and not guaranteeing timing accuracy. The most accurate setup was the gem5 simulator using a simplified cache system and an Out-of-Order detailed ARM CPU model.

Keywords: Benchmarking, emulation, GEM5, KVM, Matlab, Multicore, PARSEC, QEMU, signal filtering, simulation, virtual platforms

Acknowledgments

This master's thesis was performed at KTH Royal Institute of Technology.

- I would like to start by thanking my supervisors Johnny Öberg, for providing the research topic for this study, for his support during the development of this Master's thesis.
- I also thank my evaluator, Ingo Sander, for his dedication as a professor, for transmitting me his passion for the area of embedded systems, and for his time and patience evaluating this report.
- Lastly I would like to thank Professor Gerald Q. Maguire Jr. for providing the template for this thesis report, and for his valuable feedback during his courses, which allowed me to improve my writing skills.

Stockholm, April 2016
José Luis Bismarck Fuentes Morales

Table of contents

List of Figures

List of Tables

List of acronyms and abbreviations

ACME	Architecture Compiler for Model Emulation
ARM	Acorn RISC Machine
DTB	Device Tree Blob
EABI	Embedded-Application Binary Interface
FAME	FPGA Architecture Model Execution
FPGA	Field-Programmable Gate Array
GEMS	Multifacet's General Execution-driven Multiprocessor Simulator
HAsim	Hardware Accelerated Simulator
HDL	Hardware Description Language
HF	Hard Floating-Point
ICT	Information and Communication Technology
ILP	Instruction-Level Parallelism
NoC	Network-On-Chip
PARSEC	Princeton Application Repository for Shared-Memory Computers
QEMU	Quick Emulator
RAMP	Research Accelerator for Multiple Processors
RISC	Reduced Instruction Set Computing
RTL	Register-transfer level
SLICC	Specification Language for Implementing Cache Coherence
SoC	System-on-Chip
SPLASH	Stanford ParalleL Applications for SHared memory
UI	User Interface
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

ACME	Architecture Compiler for Model Emulation
FAME	FPGA Architecture Model Execution
FPGA	Field-Programmable Gate Array
HAsim	Hardware Accelerated Simulator
ICT	Information and Communication Technology
ILP	Instruction-Level Parallelism
NoC	Network-On-Chip
RAMP	Research Accelerator for Multiple Processors
RTL	Register-transfer level
HDL	Hardware Description Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

1 Introduction

Accurate computer architecture simulation is a vital component of embedded systems development. It provides rapid, valuable, and resource-lenient information about the performance, implications, and viability of software/hardware implementations. Simulation makes non-intrusive early software debugging possible, as well as providing hardware-specific information, such as cache performance metrics, power consumption, and data latency under different circumstances. This information opens the possibility for architecture-specific optimizations and rapid exploration of the impact of architectural choices, such as different network-on-chip (NoC) topologies, cache configurations, and core choices.

This thesis will explore multiple simulation tools and configurations to assess viable options in terms of turnaround time, ease of use, performance, and scalability for simulating a multicore systems running a Matlab compiled filtering algorithm.

Due to its high relevance in the field of embedded systems, an ARM multicore architecture was chosen as target across tests. Particularly the Cortex-A53, for being ubiquitous in the mobile market, having well developed models in simulation tools, and being easily available in development boards.

Having chosen a Cortex-A53 quad-core as the target CPU to be tested narrowed the number of viable virtual platforms which could perform accurate full-system simulation. The main criteria for selecting tools were the following:

- Having a full system simulation scope
- Support for the target architecture (Cortex-A53)
- Having an active development community
- Open source availability

Upon carefully evaluating multiple simulation and emulation tools, the gem5 simulator and the QEMU emulator were chosen.

The main benchmark application used to test different simulators and configurations was a discrete filtering algorithm written in Matlab and provided by Western GeCo. The program applies a custom filtering algorithm, including a Butterworth filter on a 5000 by 5000 matrix (find the code in Appendix A). A randomly generated matrix was used as input for this algorithm. Relevant information regarding this filtering function is limited to general behavioral information such as memory access patterns and distribution of processing resources. Specific details about the functionality of the Matlab program will not be covered, as they are not relevant to simulation benchmarking.

Additionally, the results of the PARSEC-3.0 benchmark on the tested tools will be presented in order to provide a wider perspective of the models' strengths, weaknesses and limitations.

This report will start by providing the reader with core concepts relevant simulation, including types and scopes of simulation, levels of abstraction and differences between emulation and simulation, as well as the specifics of the testing tools. Next, a summarized overview of the virtual platforms and testing setups will be presented, with brief instructions on how to replicate the setups, including the compilation of the tools, kernels and disk images necessary for the native, gem5 and QEMU versions.

1.1 Background

The context of the study case of this thesis is the need to deploy a Matlab application on a highly parallel architecture intended to be used by Western GeCo for signal filtering in seismic acquisition applications. Figure 1-1 shows the general desired workflow during architecture exploration. Western GeCo is a geophysical services company that provides reservoir imaging, monitoring, and development services.

MathWorks' Matlab is a highly popular software tool for science and engineering. Its eponymous programming language provides a wealth of functions and structures that facilitate mathematical modeling, such as matrix data types and operations, plotting, and generating code for other programming languages, such as Python, Java and C++ (see [1]).

MathWorks' Parallel Computing Toolbox (see [2]) provides explicit instructions and compiler options to allow users to parallelize Matlab code for multicore architectures. However, the code auto-generated by the Parallel Computing Toolbox is poorly optimized. Matlab compilation will not be the focus of this study.

An OpenMP parallelized version generated for a generic ARM Cortex CPU will be utilized with only a few small changes to avoid requiring a Matlab runtime (See Appendixes A and B for details).

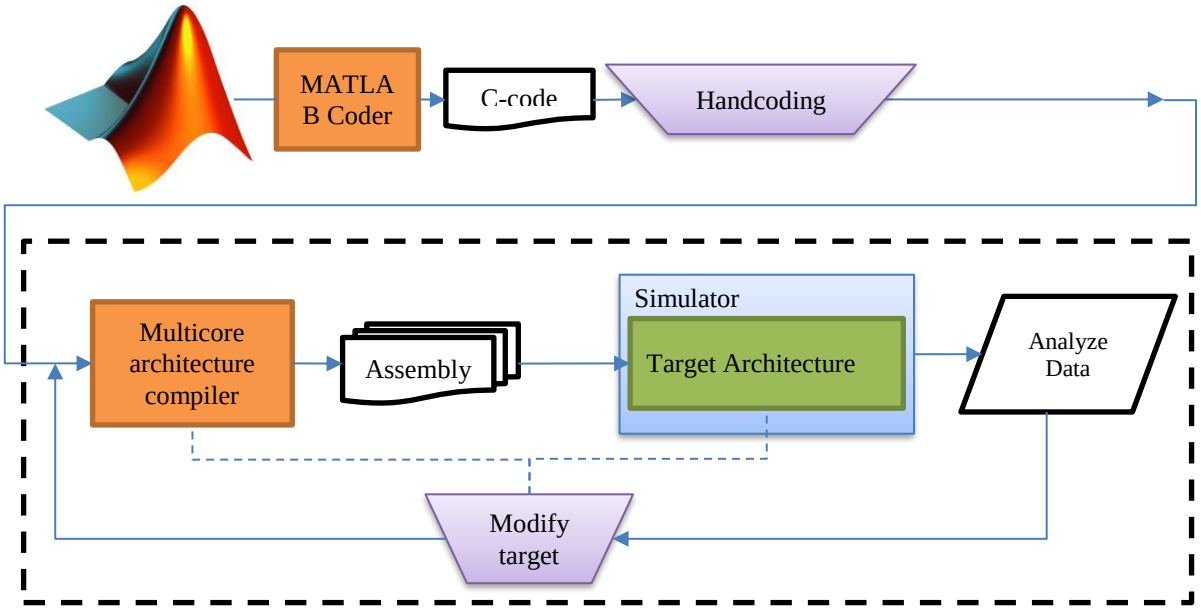


Figure 1-1 This diagram depicts the desired workflow for architecture space exploration of Matlab applications.

The following are terms that will be used throughout this thesis:

Table 1-1 List of terms.

Target	Refers to the target computer architecture being simulated in the simulation environment, including its processors, memory hierarchy and communication framework.
Host	Refers to the host physical computer running the simulation of the target architecture.
Simulator	A tool that attempts to duplicate the behavior of a given target architecture.
Emulator	A tool that attempts to duplicate the hardware and inner workings of a target architecture.
Soft core	A microprocessor core offered as either an RTL description or netlist that can be entirely implemented in an FPGA using logic synthesis. Its specifications and capabilities can often be adjusted before synthesis to suit the needs of the system's intended purpose.
Hard core	Fixed processors implemented in silicon. In contrast to soft processors, hard cores are not limited by an FPGA's speed and can generally outperform soft cores; on the flipside their behavior cannot be modified.
Turnaround time	The time between submission of a simulation for execution and its results output to the user.
Simulation time	The time simulated by a simulation proportional to the number of clock cycles simulated, and the frequency of the simulated core.
Wall-clock time	The amount of time in the real world that it takes to complete a simulation.

1.2 Problem definition

This thesis will address the problem of a filtering application that requires a highly parallel computer architecture to be executed on. Since the computer architecture itself is the variable to test, accurate simulation of computer architectures running the same filtering algorithm would provide crucial and inexpensive insight necessary for choosing an appropriate architecture. Figure 1-2 shows the simulation balance triangle inspired by [3]. An ideal simulation environment needs to be accurate, have a fast execution, be easy to develop for, and have a reasonably short turnaround time.

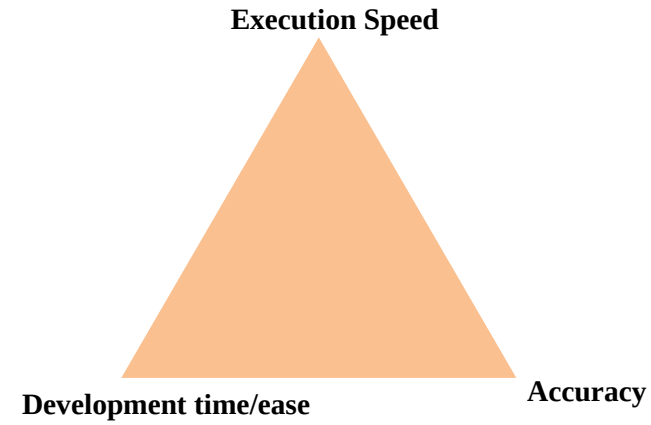


Figure 1-2 Simulation balance triangle

The problem of simulating computer architectures with high levels of parallelism has driven the industry to develop a multitude of potential solutions, with the general trends being increased levels of abstraction, adding FPGA assistance in simulation, performing distributed simulations, and using Kernel-based Virtual Machines (KVM).

KVM is a virtualization infrastructure enabled for certain architectures with supporting virtualization extensions, it can turn a Linux kernel into a hypervisor to greatly speed up emulation. KVM is currently available for Intel, AMD and ARM architectures.

This thesis will compare multiple virtual platforms and configurations to find feasible configurations in order to provide accurate design space exploration for the aforementioned filtering application.

1.3 Purpose

The purpose of this thesis is to determine the most effective way to perform hardware design space exploration via simulation of a Matlab filtering application to be run on a multicore architecture. The features and accuracy of multiple simulating tools and variants will be explored, compared, and contrasted. The results of this thesis will potentially benefit developers who need to produce high-end software/hardware solutions for many core ARM architectures. Efficient simulation will allow them to make preemptive assessments regarding their platform, hardware of choice and how their software would behave with a minimal resource investment.

1.4 Goals

The goal of this project is to determine the best virtual platform solution to perform a design space exploration on a highly parallel computer architecture intended to execute filtering algorithms generated from Matlab depending on the user's goals. This has been divided into the following sub-goals:

1. Find suitable full-system simulators.
2. Benchmark the tools with the Butterworth Matlab filter.
3. Benchmark the tools with PARSEC-3.0.
4. Analyze the results

1.5 Research Methodology

Empirical quantitative research will be employed for this thesis, where multiple tools will be tested. All tests will simulate their most accurate possible simulation/emulator of the target architecture, which will be benchmarked using the Butterworth Matlab filter algorithm, and the PARSEC-3.0 benchmarks suite. A native setup containing the target architecture will also undergo all of the tests to be used as reference for accuracy measurements, see details in Section 3.

A qualitative approach was chosen, as simulation performance is easily measurable, and can be used as a reasonable indicator of compliance to research goals.

1.6 Delimitations

The scope of this thesis will be limited to full-system simulators/emulators, their configurations and viability to fulfil the target application goals. Simulation tools that are limited to a single architecture, or are no longer maintained will be omitted. Specific details regarding the Matlab filtering application, its purposes, logic, and inner workings will not be discussed. Matlab code generation will also not be discussed.

The gem5 simulator and the Quick Emulator (QEMU) emulator will be tested for this study. In the case of gem5, the following CPU models will be benchmarked with the Butterworth filter: AtomicSimple, TimingSimple, Out of Order (O3), and ARMDetailedO3. PARSEC benchmarks will be limited to the AtomicSimpleCPU, as they take an enormous amount of time to complete using other models, and AtomicSimpleCPU proved to be the CPU model with the best balance of turnaround time and accuracy.

QEMU tests will be limited to QEMU-KVM, as it is the only mode where cache statistics may be captured, making it the only relevant mode for this study.

FPGA-Assisted simulation was also considered for this study, however, none of the FPGA assisted simulation tools were successfully ran for various reasons, including licensed software and outdated repositories, a summary of these efforts is provided in Appendix C.

1.7 Structure of the thesis

Table 1-2 Structure of this thesis report.

Chapter	Title	Contents
I	Introduction	1. An introduction to the topic. 2. Brief background contextualization. 3. Definition of terms. 4. Definition of the problem. 5. Research goals. 6. Research methodology. 7. Research delimitation.
II	Background	1. Background information required to understand the study. 2. A bottom-up definition of computer simulation and emulation 3. Evaluation criteria for computer simulators 4. Difference between emulators and simulators. 5. Brief overview of memory consistency models. 6. A presentation of gem5 and its features. 7. A presentation of QEMU and its features 8. An overview of the testing benchmarks. 9. A list of related work.
III	Testing Setups	A summarized tutorial on how to replicate the testing setups for the native, gem5, and QEMU tests.
IV	Experimental Results	Results of empirical testing of multiple simulation tools and benchmarks.
V	Analysis	Analysis of the obtained results
VI	Conclusions	Conclusions based on the analysis and its compliance with the thesis' goals
VII	Failed Attempts	This chapter contains the pre-analysis and summary of failed attempts at benchmarking FPGA-assited and FPGA-based simulation.
VIII	References	Books, papers and online documents referenced in the thesis study.

2 Background

This chapter provides basic background information about simulation of computer architectures, starting from the taxonomy of computer simulation, to its levels of abstraction, and ending with an overview of specific simulation tools. Lastly, this chapter lists literature relevant to this discussion.

2.1 Computer simulation

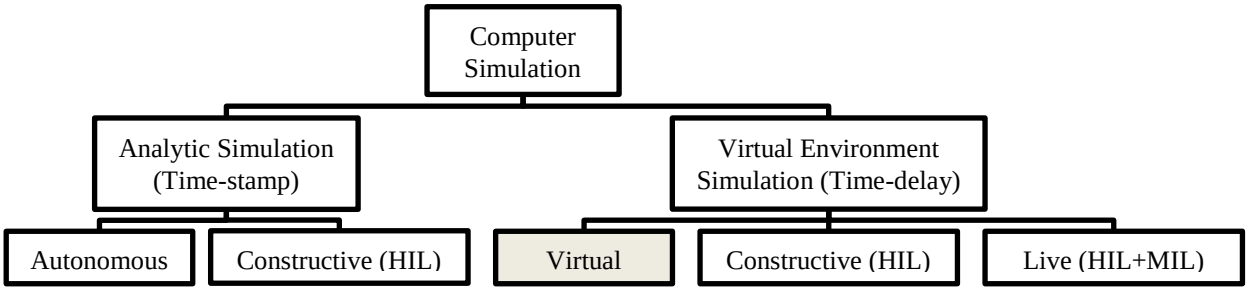


Figure 2-1general computer simulation classification as presented in [1, Fig. 1.4]

From [4, Ch. 1], Figure 2-1 classifies computer simulation into two broad types. Virtual environment (VE), also referred to as time-delayed simulation, works by recreating the simulated sequence of events in a virtual environment with scaled time. In contrast, analytic or time-stamped simulation, operates in real time, trying to replicate the sequence and timing of events as accurately as possible. Modern computer architecture simulators primarily operate with time-delayed simulation, as it is the more viable, hardware independent, and scalable of the two approaches. Regarding interactions, VE simulation is further classified depending on whether it has a human-in-the-loop (HIL) interaction, or machine-in-the-loop software agent interaction. This study will only focus on the category without external agents, also known as virtual simulation.

2.1.1 Service-Oriented Architecture

The virtual platforms evaluated in this thesis are structured in a service-oriented architectural approach to different extents. Service-Oriented Architecture (SOA), is a software design architectural pattern where application components provide services to each other using a communications protocol [5]. Services can have self-contained functionalities which can be swapped depending on the needs of the application, easing the addition and development of features. Virtual platforms can be intuitively implemented using a SOA, where the functions of different virtual components, such as memories, pipeline, and networks on chip (NoCs) are provided by different services.

2.1.2 Simulator Evaluation Criteria

More computer simulation specific literature classifies computer architecture simulation by level of abstraction, where higher levels of abstraction provide faster, less resource-intensive simulation by sacrificing accuracy. Simulators using different levels of abstraction may be chosen depending on the level of simulation detail demanded by the target application. Table 2-1 based on [6, Ch. 1.1] makes a general classification of simulation by levels of abstraction from precise to abstract.

Level	Description
Gate level	Models the individual logic gates and state elements that make the digital circuit of the system.
Cycle level	Operations are described with clock cycle granularity, including the states of individual bits.
Transaction level	Assumes negligible influence from the internal behavior of the processing units to the whole system's behavior when compared to the impact of memory transactions and the overhead of communications between processing units.

Table 2-1Categorization by levels of Abstraction.

Orthogonal to the classification by level of abstraction, computer simulators can be classified based on the scope of their simulation model. Based on [6, Ch. 1.2] , Table 2-2 summarizes this categorization.

Scope	Description
Functional simulator	Limited to simulating program instructions, updating processor registers, and performing operations; does not provide any performance information.
Performance, detailed or cycle-level simulator	Complements the information provided by a functional simulator with performance information, such as clock ticks per instruction (CPI), power consumption and bus traffic.
Full-system simulator	Models and simulates the entirety of the software and hardware systems, including peripherals, cache hierarchies, and operating systems.

Table 2-2 Categorization by scope of simulation models.

2.1.3 Simulator Evaluation Criteria

Since this study will compare multiple simulators that work with different scopes and use different types of models with different levels of abstraction to represent computer architecture systems, I will establish a common ground to compare completely different pieces of software that represent the same systems. To this end, I will use the desired characteristics of a simulator presented in [7, Ch. 2.2], as well as the turnaround time defined in Chapter 1.

1. **Speed**: Is measured in the amount of time it takes to evaluate a model. Can be measured as the number of instructions executed by the simulator per real clock seconds if the level of abstraction of the simulator contains said value.
2. **Accuracy:** Is the measurement of the extent to which a simulator can update its processor's architectural state to match the system being simulated. Depending on the level of abstraction used in the simulator, accuracy can be analyzed at per-cycle, per-instruction, or per-function granularity, each having independent implications.
3. **Flexibility**: Is the ease with which a simulator's model can be modified to evaluate different target functionalities or configurations, enabling rapid and efficient exploration of design options.
4. **Completeness** refers to the extent to which the target hardware can be simulated, including peripherals and its potential interaction with other components and conditions.
5. **Usability:** represents the amount of effort required to setup and use a simulator for a specific purpose. Usability varies depending on the intended use cases, some simulators are more usable for certain tasks than others. All of the previously mentioned factors can contribute to usability. In the case of this study, I will specifically focus on ease of setup and execution of customized multicore architectures.

2.2 Emulators

Emulators are virtual platforms that allow their host system to behave like a given target system [8], they work by creating an abstraction layer between the host and target systems. Emulation focuses on functional accuracy and performance, allowing for a highly accurate functional replica of the target system, including elements such as hardware, peripherals, and operating systems. An emulator is very similar to an accurate full-system simulator, albeit with functionality as its primary goal at the compromise of timing accuracy. The higher performance in emulators when compared to full system simulators is often attained by taking advantage of features such as dynamic translation and kernel-based virtual machines (KVM) when available.

2.2.1 Kernel-based Virtual Machine

A Kernel-based Virtual Machine is a loadable kernel module (kvm.ko) that provides a full virtualization solution leveraging its native CPU. KVM makes it possible to execute multiple virtual machines running unmodified Linux or Windows operating systems, each with its private virtualized hardware: memory, network card, and peripherals.

KVM extensions are currently available for Intel, AMD and ARM architectures.

2.3 Memory consistency

In systems with multiple reading components sharing a memory space, memory consistency ensures a consensus on the order of the data read by the readers in relation to when and by which writer its written. There are multiple consistency models, which's viability depends on their scope, on the scale of the system, and application constraints.

2.3.1 Atomic consistency

Ensures sequential consistency even when multiple read or write operations occur during the same instant of time. If multiple read and write operations occur at the same instant, it can define that reads happen before writes (static atomic consistency), or allow operations to take effect at any point, as long as the resulting access history is equivalent to a serial execution (dynamic atomic consistency) [9].

Atomic consistency is strictest of consistency models, making it generally unfeasible and only used as reference for evaluating other consistency models.

2.3.2 Sequential consistency

First defined by Leslie Lamport in 1979 [10], sequential consistency is the weaker and more feasible form of atomic consistency, and for a long time was the canonical consistency model for many applications [9]. In this model operations of all the processors as if they were executed in a certain sequential order, even if multiple processors perform accesses simultaneously. Each individual processor appears to execute its accesses as specified by its program order.

2.3.3 Causal consistency

Defined by Hutto and Ahamad in 1990 [11], causal consistency is a weaker consistency model than sequential cosnsitency. It interprets write operations as sending messages, and read operations as receiving messages. Memory consistency is achieved when all processors perceive the same order of causally related events, that is, events that should happen one after another, such as a write (message-send) occuring before a read (message receive). Nonetheless, unlike in sequential consistency, events that are not causally related may be observed in different orders under causal consistency. Causal consistency is one of the most difficult models to implement in hardware, as, unlike other consistency models, it was not conceived with a hardware implementation in mind [9].

2.3.4 Pipelined RAM (PRAM) consistency

PRAM is a weaker form of sequential consistency, where consistency is only kept whilst accessing main memory, this scheme relies on having proper cache coherance if caches are implemented. During read operations, the PRAM returns its local copy of the value. Write operations update its local copy and then broadcasts it to the other processors, this scheme maintains a stable and scalable read and write overhead. Processors can know the correct sequence of reads and writes of any other processor, however, they may differ in the order in which other processors made memory accesses.

2.3.5 Cache consistency

Is similar to PRAM consistency, which is also a localized form of sequential consistency, but applied at cache granularity. Some authors such as [9] use cache consistency as a synonymous with cache coherence, they are not the same but imply eachother. Cache coherence ensures that all processors will read valid and updated copies of memory locations despite modifications from other processors, thus ensuring cache consistency in that each processor will read in its cache memory events from each processor in the right

order, but processors may differ in the order in which other processors performed their accesses.

2.3.6 Processor consistency

Is a consistency model that combines PRAM consistency, cache coherance, and cache consistency, thus being stronger than both PRAM and cache consistency, but still weaker than sequential consistency, as the strinct sequential order between processor accesses is not enforced in all cases, see [9] for details. Processor consistency is feasible for both simulation and hardware, and in most cases will yield results identical to sequential consistency.

2.3.7 Slow memory

Named after the time penalty for writes in its implementations that require writes to propagate and be visible immediately to all processors, slow memory is a weaker form of PRAM. It ensures that all processors perceive the correct write history of a processor to each memory location. This algorithm only guarantees physical mutual exclusion.

2.3.8 Weak consistency

Originally proposed by Dubois [12], weak consistency is a hybrid model that relies on the use of synchronization variables in software, and memory fences that ensure atomic accesses to said variables in hardware. Using both of the aforementioned tools, weak consistency can ensure sequential consistency under the following conditions:

1. There is no competing access to the same memory location.
2. Synchronization variables are visible to the memory system.

Weak consistency is highly popular due to its scalability and low hardware and software costs, with the drawback of being weaker than sequential consistency, being unable to handle chaotic memory accesses, and requiring explicit and premeditated use of software synchronization mechanisms. A memory system has weak consistency if it enforces these restrictions [9]:

1. Accesses to synchronization variables are sequentially consistent.
2. A processor may not issue an access a synchronization variable before all previous memory accesses have been completed.
3. A processor may not issue a memory access to shared memory before previously accessing a synchronization variable.

2.3.9 Release consistency

Release consistency is an improved and stronger version of weak consistency [13] where competing memory accesses are divided into acquire, release, and non-synchronizing accesses.

1. **Acquire accesses** perform a synchronization variable access, similar to those in weak consistency, but delaying future accesses only
2. **Release accesses** are also similar to the weak consistency synchronization variable release, but with the fence delaying until all previous accesses have concluded
3. **Non-synchronizing accesses** are competing accesses that do not serve synchronization purposes. This type of accesses can be used to handle chaotic accesses.

2.3.10 Entry consistency

Comparably to slow memory, entry consistency is a location relative weakening of release consistency, where every variable has to be associated with a synchronization variable, which is either a lock or a barrier. These associations can change dynamically. One of the added benefits of entry consistency is that accesses to different critical sections can occur concurrently, it also divides acquire accesses further into:

1. **Exclusive acquisition** blocks all other processes attempting to acquire the synchronized variable.
2. **Non-exclusive acquisition** can grant multiple accesses to the same synchronization variable concurrently.

2.4 Gem5

Is an open source modular simulation tool developed by multiple academic and industrial institutions including the National Science Foundation, AMD, ARM, Hewlett-Packard, IBM, Intel, and Sun, among others [14]. Gem5 contains the combined features of the M5 Simulator [15] and Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset [16]. Gem5 utilizes M5's highly flexible and diverse core simulation infrastructure, supporting most commercial ISAs, including MIPS, Power, SPARC, ARM, ALPHA and x86, the last three including full system features. Complementing M5's simulation features, GEMS provides Gem5 with Ruby and Garnet, which can be used to model detailed memory and interconnect models respectively, including support for detailed simulation features, such as cache coherence protocols and customized NoC topologies.

There is a multitude of experimental modules being developed for gem5 by different institutions, the following is a comprehensive summary of the currently main and trending modules of gem5 as of 2016, see Figure 2-2 Summary of the main trending gem5 modules.

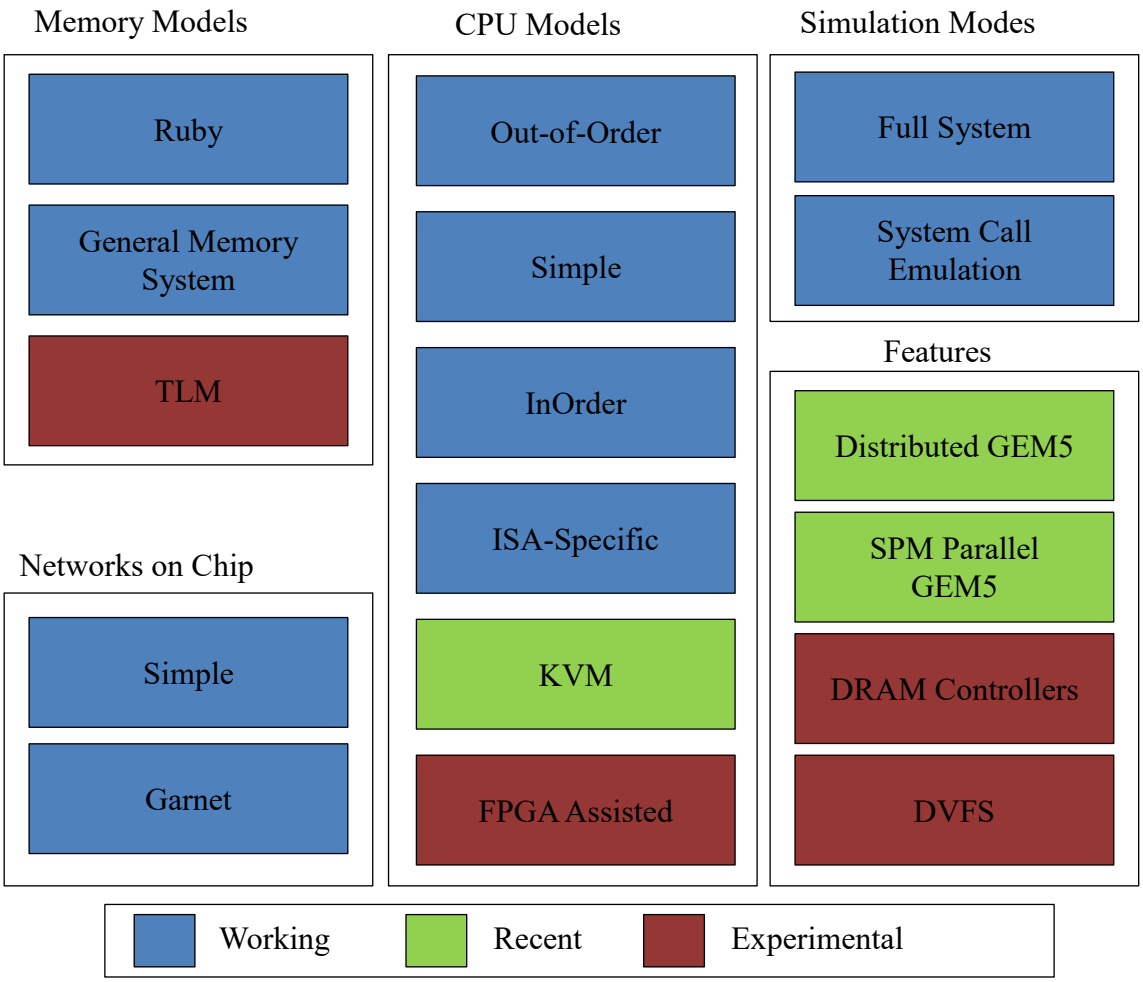


Figure 2-2 Summary of the main trending gem5 modules

Processor		Memory System		
CPU Model	System Mode	Classic	Ruby	
			Simple	Garnet
Atomic Simple	SE	Speed		
	FS			
Timing Simple	SE			
	FS			
In-Order	SE			
	FS			
O3	SE			
	FS			Accuracy

Table 2-3 gem5 trade-offs matrix

The different gem5 modules provide a series of trade-offs between speed, flexibility, and accuracy, which can be summarized with the following matrix from [14], where simulation speed increases towards the top left, and accuracy increases towards the bottom right.

2.4.1 Gem5's base infrastructure

2.4.1.1 SimObjects

Gem5's service-oriented architecture is achieved through an object oriented design using SimObject [14, p. 3]. SimObjects are software objects that share common configuration, initialization, statistics and intercommunication features, they represented by two classes derived from the SimObject base classes of their respective languages: one in Python and one in C++. All major gem5 components are SimObjects, including both concrete and abstract models of hardware components such as cores, caches, NoCs, pipelines, and memory controllers.

The C++ SimObject class contains all the performance-critical functions of the modeled component, it includes the SimObject's state, behavior, and performance-critical simulation model. The Python SimObject class, on the other hand, contains the component's parameters and specifications, and used for script-based configuration.

For details about SimObjects' call sequence, see [17].

2.4.1.2 Domain-Specific Languages

Gem5 provides two domain-specific languages (DSL) for specialized tasks, namely defining instruction set architectures (ISAs) and specifying cache-coherence protocols.

1. ***ISA DSL*** is gem5's ISA description language, inherited from m5. It unifies semantics specification and instruction decoding [14, p. 3]. ISA DSL uses a common C++ base class to describe the multiple stages of instructions, simplifying instruction implementations to a set of declarations of these derived classes, and virtual function overrides for the multiple pipeline stages, such as decode(), execute, etc. The language also allows the specification of a decode tree that combines opcode decoding with the creation of the classes necessary to execute the decoded instructions. See [18] for details.
2. ***Specification Language including Cache Coherence (SLICC)*** is a domain-specific language used for defining caches, memories, and DMA controllers as individual state machines with a memory block granularity that in conjunction enforce coherence protocols [14, p. 3]. SLICC was inherited from GEMS is only necessary when using a Ruby memory system. Protocols are defined as sets of states, events, transitions, and actions. Valid combinations are specified in transition statements, as well as the operations to be performed within and after each transition. See [19] for more information.

2.4.2 Operation Modes

2.4.2.1 System Call Emulation (SE) Mode

Runs individual applications using the user-visible part of the ISA model and common system calls. It uses a simplified address translation model with no scheduling and no other processes. System calls are emulated via the host OS. SE simulation is very fast, and is the

preferred simulation mode for testing base functionality. It is available for the following ISAs: Alpha, ARM, SPARC, MIPS, PowerPC, and x86.

2.4.2.2 Full-System (FS) Mode

Runs a full system model, which can include caches, interrupts, devices, exceptions, fault handlers, and an operating system. Can be used to run Linux distributions. It is the most feature-rich of the two modes, but is restricted to the following ISAs: Alpha, ARM, SPARC, MIPS, and x86.

2.4.3 CPU Models

Gem5 provides multiple CPU models with varying degrees of detail regarding memory access and instruction execution.

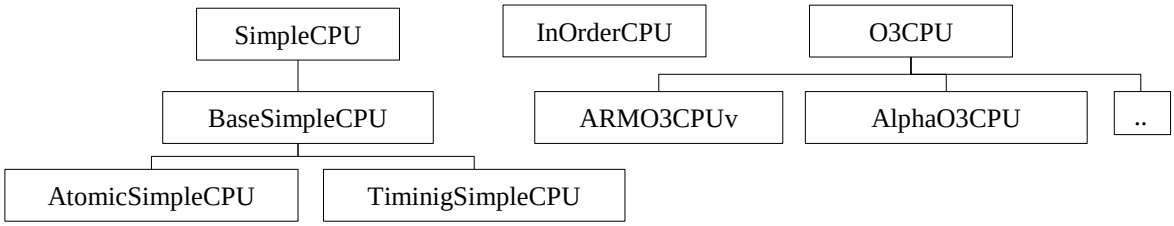


Figure 2-3 Gem5 main CPU families and their hierarchy

2.4.3.1 SimpleCPU

Is the simplest gem5 CPU model, it provides functional correctness with a simplified memory model and support for warm-up periods. This model was split into three classes: BaseSimpleCPU, AtomicSimpleCPU, and TimingSimpleCPU, each with subsequently more detailed memory models.

2.4.3.2 BaseSimpleCPU

Is the parent class of AtomicSimpleCPU and TiminigSimpleCPU. It holds the state and common stats across SimpleCPU models and provides the following functionality:

- Interrupts management and handling.
- Pre and post execute actions.
- Setting up fetch request and incrementing the Program Counter (PC).
- ExecContext interface [20] between the ISA and the CPU state and provides full-system features.

2.4.3.3 AtomicSimpleCPU

Is a subclass of the BaseSimpleCPU that implements the “tick” function to define the state of the CPU after every clock cycle. And models memory read/write functions using atomic memory accesses.

Execution model: Simple callback to ISA implementation, Figure 2-4 depicts its execution flow as seen in [21].

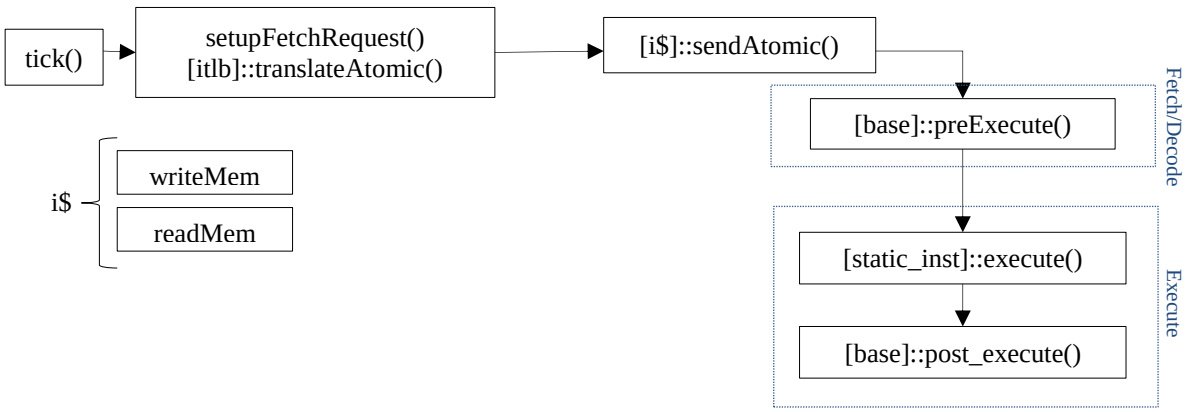


Figure 2-4 Flow diagram depicting the AtomicSimpleCPU execution flow upon the tick function call. I$ contains the current instruction data, and is updated from calls inside static_inst execute method to helper functions. itlb is the instruction translation lookaside buffer. Base refers to the base execution context "ExecContext". static_inst is the static functional execution model of the decoded instruction.

Memory access model: It estimates overall cache access times from latency estimates.

2.4.3.4 TimingSimpleCPU

A subclass of the BaseSimpleCPU that implements the same functions as AtomicSimple, but has a more detailed memory model by using timing memory accesses.

Execution model: Figure 2-5 has a similar execution flow to SimpleAtomic, with the inclusion of timing callbacks for each stage with acknowledge mechanisms [21].

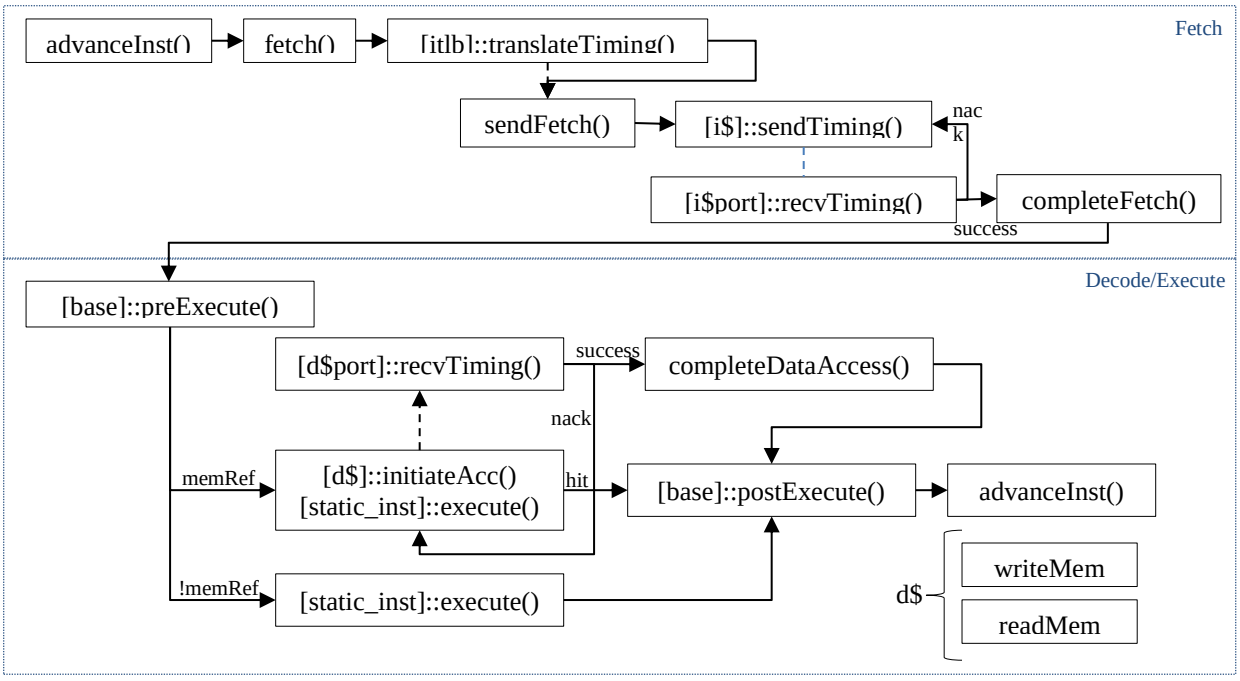


Figure 2-5 Flow diagram depicting the TimingSimpleCPU execution flow upon the tick function call. i$ contains the current instruction, and is updated from calls inside static_inst execute method to helper functions, d$ is the instruction decoded from i$. itlb is the instruction translation lookaside buffer. Base refers to the base execution context "ExecContext". static_inst is the static functional execution model of the decoded instruction.

Memory access model: It stalls on cache accesses and waits for the memory system to respond prior to proceeding, it implements functions to send requests and handle the cache responses.

2.4.3.5 InOrderCPU

A generic framework to simulate in-order pipelines with arbitrary descriptions. Operates on a "tick" granularity. Provides a higher level of detail than the SimpleCPU models, with abstractions for the individual pipeline components, such as ALU, FPU, and Branch predictor into a "resource" object requested by each instruction. Makes possible modeling custom pipelines without having to design a complete CPU. This model was not used for this thesis, details can be found at [22].

2.4.3.6 O3CPU

Models an Out-of-Order CPU, loosely based on the Alpha 21264 (See [23] for details). Similarly to the InOrderCPU, it operates on tick granularity. Includes models for five pipeline stages as follows:

Fetch: Performs branch prediction, fetches instructions from instruction memory, selecting a thread to fetch from based on a previously configured policy.

Decode: Decodes instructions and handles early resolution of PC-relative unconditional branches.

Rename: Renames instructions if possible. Stalls if there are not enough registers to rename to, or if back-end resources have filled up. Handles serializing instructions.

Issue/Execute/Writeback: Combines dispatching instructions to the instruction queue, executing, and writing back into a single stage.

Commit: Handles instructions faults and hazards, and commits them. Redirects the front-end in the case of a branch misprediction.

Additionally, the O3CPU pipeline makes use of the following components:

Branch predictor	Predicts branches via the selected method between a tournament, local, or global predictor. Has a branch target buffer and a return address stack.
Reorder buffer	Keeps instructions in program order and handles squashing.
Instruction queue	Predicts instruction dependencies avoids data hazards.
Load-store queue	Handles data forwarding and ensures loads/stores happen in the intended order
Functional units	Determines which instructions can be issued and their latency.
Memory dependence prediction using store sets	Predicts which instructions are ready to issue, and notifies the instruction queue.
ISA-Dependent Functions	In addition, O3 may implement ISA-dependent functions, for features unique to a given architecture.

2.4.4 Memory Models

Gem5 inherits two different and complementary memory systems: M5's "classic" memory system, and GEMS' Ruby memory system [24].

2.4.4.1 Memory consistency in gem5

Gem5 uses processor consistency in both of its memory implementations. This is enforced by the cache coherence protocols: An idealized, zero-delay MOESI for its classic memory system; and a variety of other architecture-specific cache coherence protocols for Ruby, including MSI, MESI, MOESI, AMD's hammer, among others. Processor consistency is furthermore enforced by inclusion of a memory checker monitor (mem_checker_monitor.hh and .cc) in the standard bus object definition. The memory checker monitor creates a host thread that observes read and write operations and guarantees that reads observe values from permissible writes. Depicts an example of a gem5 implementation of a multicore architecture with shared memory and its monitors.

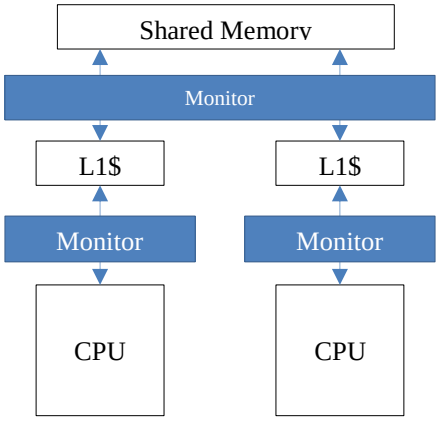


Figure 2-6 Gem 5 memory monitors, and their place in memory buses.

2.4.4.2 Classic Memory System

All Classic memory objects inherit from the MemObject superclass, and are connected using port objects. MemObject ports support both point-to-point connections between two MemObjects, and bus/crossbar interconnections between two or more MemObjects. The MOESI protocol is used as the sole cache coherence mechanism, with an abstract implementation without transient states and with instantaneous state transitions upon snoops.

2.4.4.3 Ruby Memory System

Ruby trades-off simulation speed for accuracy, flexibility, and support for a large number of systems. Ruby cache models are specified using SLICC (Specification Language for Implementing Cache Coherence), a domain-specific language for cache coherence protocols, which modularizes the definitions for cache memories, DMA controllers, and individual per-memory-block state machines to implement the coherence protocols.

Ruby uses RubyPort objects to connect the CPU and devices to the memory. Objects within a Ruby memory system are interconnected using MessageBuffers, which, unlike RubyPorts, include message queues to model timing accesses with greater detail than in Classic Memory. Ruby can be enabled by using argument **-ruby** in both SE and FS configurations.

Gem5's memory models can be compared as follows:

	Pros	Cons
Classic	-Supports atomic accesses -Faster -Easy to configure	-Lack of flexibility -Low coherence protocol fidelity. -Very difficult to modify
Ruby	-High flexibility -High coherence protocol fidelity	-Very slow -Poor documentation. -Adding custom protocols requires learning an undocumented domain-specific language.

2.4.5 Interconnection Networks

As with memory systems, gem5 provides two options, the classic interconnect network model that simplified switch modeling, and the more detailed and resource intensive Garnet interconnection network model inherited from GEMS. Both models offer mesh, mesh with directory corners, torus, crossbar, and point-to-point network topologies See [25] for details.

Network.py contains all valid network parameters. General parameters valid for both networks include

number_of_virtual_networks: Maximum number of virtual networks. The real number of virtual networks is determined by the interconnect networking protocol.

control_msg_size: Size in bytes of control messages. Defaults to 8.

2.4.5.1 Simple Networks

Simple network models traversal in a hop-by-hop fashion, meaning it models the delays duing each node hop. This model is less precise than garnet in that it simplifies switches into an abstract perfect switch model with no delays. Switch models can be found in simple/PerfetSwitch.cc, link models are found in simple/Throttle.cc. Flow control is achieved by monitoring available bandwidth and buffers before sending.

In addition to generic network parameters in Network.py, simple network parameters are specified in simple/SimpleNetowrk.py:

buffer_size: Input and output buffer sizes. 0 implies unlimited buffering.

endpoint_bandwidth: Bandwidth at network endpoints .

adaptive_routing: Enables adaptive routing based on the occupancy of output buffers.

2.4.5.2 Garnet Networks

Garnet is a more detailed interconnect network module inherited from GEMS, which, in addition to buffers and links, includes models of different levels of details for router pipelines. Its original research paper can be found in [26]. It supoports modeling variable link bandwidths, multicast messages, and is currently limited to modelling only deterministic routing using routing tables.

Garnet uses the generic network parameters in Network.py, as well as the following common configuration parameters for both of its pipeline models which can be found in garnet/BaseGarnetNetwork.py:

nl_flit_size: The size of network-level units messages are broken into to be sent throughout the network [26] (flits) defined in bytes. Defaults to 16 bytes. Must be the same as bandwidth_factor.

vcs_per_vnet: Number of virtual channels (VC) on each virtual network. Defaults to 4.

The Garnet interconnection network infrastructure provides two different pipeline models for the user to choose from depending on simulation needs :

The fixed-pipeline model is intended for evaluating low-level interconnection networks. Models a detailed 5-stage virtual channel (VC) router with credit-based flow control. This model is accurate and is encouraged to be used as the default model when accuracy is of importance. Fixed pipelines may use the following configuration parameters specified in garnet/fixed-pipeline/GarnetNetwork.py:

buffers_per_data_vc: specifies the number of flit buffers per virtual channel used for data messages, can have a value between 1 and 5. Defaults to 4.

buffers_per_ctrl_vc: specifies the number of flit buffers per virtual channel used for control messages. This value can only be 1.

The flexible-pipeline model is intended for performance and flexibility, it provides an abstraction of all interconnection model network models where pipeline depth can be adjusted, similar to gem5 classic interconnect model, albeit with a more detailed router pipeline that may range from a single to multiple cycles . This model is efficient and prefered for early architectural evaluations concerning topologies and pipeline depths. Flexible pipelines have the following parameters available, specified in garnet/flexible-pipeline/GarnetNetwork.py:

buffer_size: size of virtual channel buffers, 0 implies unlimited buffering.

number_of_pipe_stages: specifies the number of pipeline stages per router. Defaults to 4.

2.4.6 Building Gem5

Gem5 can be built for multiple architectures, including ALPHA, ARM, MIPS, POWER, SPARC, and X86, many of which have multiple variations depending on coherence protocols, X86_MESI_Three_level, for example, will build an X86 binary with support for a 3 level Ruby cache with MESI coherence, to see all possible build options, open the variables folder.

Orthogonally from the architecture, gem5 can build the following executables depending on simulation needs:

gem5.debug: Used for debugging without optimizations. Has a fast, unoptimized compilation, but the slowest execution.

gem5.opt: Binary with debugging and optimization. Is much faster than gem5.debug and retains all debugging capabilities.

gem5.prof: A lot like the gem5.opt binary, but also includes profiling support.

gem5.perf : Contains the same capabilities as gem5.prof, but is aimed for CPU and heap profiling using Google's perftools.

gem5.fast: Removes all debugging and tracing support in favor for link time optimization, and the fastest execution time out of all available binaries.

Having fulfilled all of gem5's dependencies [27], gem5 can be built using the following scons command. Where <arch> is replaced with any of the available architectures, and <binary> with any of the aforementioned gem5 targets.

```
scons build/<arch>/gem5.<binary>
```

After successfully compiling a gem5 binary, the binary can be run as follows:

```
<gem5 binary> [gem5 options] <simulation script> [script options]
```

Where <gem5 binary> is the route to your compiled binary e.g. build/ARM/gem5.opt, <simulation script> is the path to the simulation script to be run. All of the available gem5 options can be found using the -h flag. For more information about the simulation options, as well as the script options for full-system and syscall-emulation scripts see [28].

2.5 QEMU

QEMU (Quick Emulator) is an open source machine emulator and virtualizer written by Fabrice Bellard. Most of its parts are licensed under GNU General Public License (GPL), others under other GPL-compatible licenses.

QEMU is a hosted virtual machine monitor: It emulates CPUs through dynamic binary translation and provides a set of device models, enabling it to run a variety of unmodified guest operating systems. It also can be used together with KVM in order to run virtual machines at near-native speed (requiring hardware virtualization extensions on x86 machines). QEMU can also be used purely for CPU emulation for user-level processes, allowing applications compiled for one architecture to be run on another.

QEMU features three distinct operating modes:

2.5.1 User mode emulation

With fast cross-compilation in mind, User mode Allows QEMU to run single programs that were compiled for a different instruction set than the host's. System calls are thunked for endianness and for 32/64 bit mismatches. Currently supports multiple Linux and Darwin/OS X ISAs.

2.5.2 System emulation

Emulates a full computer system, including peripherals, interrupts, and fault handlers, among other features. The CPU is emulated using dynamic binary translation. System-specific aspects are provided via "machine" device models, enabling this mode to run a several unmodified guest operating systems, including Linux, Solaris, Microsoft Windows, DOS, and BSD; its supported ISAs include: x86, MIPS, 32-bit ARMv7, ARMv8, PowerPC, SPARC, ETRAX CRIS and MicroBlaze.

2.5.3 KVM Hosting

To allow for faster emulation speeds, if the host and target machines have the same architecture, QEMU can support running a KVM (Kernel-based Virtual Machine) Hosting. KVM is a full virtualization solution, where the execution of the guest is done by the host CPU directly using virtualization extensions. This is achieved by loading a kernel module that provides the core virtualization infrastructure, and a processor specific module on an isolated core inside the host machine. QEMU currently supports Intel-V and AMD-V KVM, however, ARM-KVM is also possible.

2.5.4 QEMU's Memory Model

QEMU's system emulation operates using a very rudimentary memory API, lacking cache and NoC models entirely. QEMU defines four different types of memory regions [29]:

1. **Containers** model buses, bridges, or controllers. Each container contains a list of memory regions, each at each own address.
2. **RAM memory** is modeled by simple byte arrays that can be accessed by the guest.
3. **MMIO regions** model memory areas where accesses have side effects, or where logic needs to be performed on the accessed values.
4. **Aliases** are used to model remapped memory

2.5.5 Running QEMU

QEMU can be installed through most Linux repositories, or alternatively it can be cloned from git clone git://git.qemu-project.org/qemu.git.

QEMU can be run using the qemu command, see [30] for a complete list of all possible options and support commands.

2.6 Benchmarks

The study case for this thesis will be a 2D discrete Butterworth filter operating on a randomly generated 5000 x 500 matrix, see Appendix A for details. In order to have a deeper insight into the accuracy and speed of gem5 and QEMU's ARM models, more complex and standardized benchmarks were used. The Princeton Application Repository for Shared-Memory Computers (PARSEC) was chosen for its wide acceptance in multiprocessor benchmarking.

2.6.1 Butterworth Filtering Algorithm

Functions Tic and Toc were removed for lack of support from Matlab Coder. The Butterworth filter C-code was generated using the Matlab coder app from Matlab R2014a. A generic ARM-compatible Cortex architecture was used as target for compatibility purposes. To simplify compilation, all functions were bundled into a single file. See Appendix A and Appendix B for the original Matlab file and its C-code counterpart.

2.6.2 PARSEC-3.0

Princeton Application Repository for Shared-Memory Computers (PARSEC) is a freely available community benchmark suite created by Intel and Princeton University, and used to evaluate and develop chip-multiprocessors with a focus on behavioral insight over raw performance [31]. PARSEC is comprised of multithreaded emerging workloads, with a diverse selection of multithreaded applications. PARSEC can be built to run using multiple parallel development C language tools, including: PThreads, OpenMP and Intel TBB. Being a newer suite than the other popular multiprocessor benchmark, SPLASH-2, PARSEC uses more programs that are more up-to-date, offering a wider variety of parallelization approaches, and a broader range of application domains.

PARSEC can be compiled and run with different parallelization options, input sizes, and number of threads using the *parsecmgmt* command, see its man page at [32].

Program	Application Domain	Parallelization		Data Usage	
		Model	Granularity	Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	low	low
bodytrack	Computer Vision	data-parallel	medium	high	medium
canneal	Engineering	unstructured	fine	high	high
dedup	Enterprise Storage	pipeline	medium	high	high
facesim	Animation	data-parallel	coarse	low	medium
ferret	Similarity Search	pipeline	medium	high	high
fluidanimate	Animation	data-parallel	fine	low	medium
freqmine	Data Mining	data-parallel	medium	high	medium
streamcluster	Data Mining	data-parallel	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	low	low
vips	Media Processing	data-parallel	coarse	low	medium
x264	Media Processing	pipeline	coarse	high	high

Table 2-4 Qualitative summary of the inherent key characteristics of PARSEC benchmark suite programs as seen in [19, p. 3]

2.7 Related Work

Memory consistency models [9]:

This paper has an extensive description, review and discussion of consistency models, and their usage in software. A special emphasis is made on weak consistency using hybrid models, that is, using a software tool that requires hardware component to support it. This paper also provides examples of circumstances during which each consistency model holds up. One notable thing is that this author considers cache consistency and cache coherence as the same thing.

The gem5 simulator [14] :

This is the gem5 introductory paper. It contains a brief history of the gem5 simulator, including its origins, and the features it inherited from the M5 and GEMS simulators, as well as the changes that had to be made to merge both tools. Gem5's service-oriented architecture (SAO) achieved through SimObjects is explained here, where there can be multiple interchangeable models for ISAs, memory models, interconnect networks and even simulation modes.

Accuracy evaluation of GEM5 simulator system [33]**:**

This paper performs an accuracy evaluation of gem5. The author uses an STE Nova A9500 evaluation board as reference and compares the results of both using multiple benchmarks, mostly focused on single-core performance.

An Optimization Approach for QEMU [34]**:**

This work helps the reader understand how QEMU works. It provides an overview on how it allocates resources and then focuses on how binary translation works and proposes several ways in which it can be improved.

PARSEC specifications paper [31]:

This is the original PARSEC paper. It starts by introducing the need rationale behind PARSEC, and the need for updated benchmarks for scientific highly parallel modern applications. It goes through each of the PARSEC workloads, its origins, why it is relevant, and how it has unique data characteristics compared to other workloads. It also talks about how to compile PARSEC and the details of utilizing the parsecmgmt command.

SPLASH-2 vs PARSEC [35]:

This paper provides an objective comparison between the PARSEC and SPLASH-2 benchmark suites. Being the precursor of PARSEC, it is necessary to also review SPLASH-2, an older benchmark for highly parallel shared memory applications. This paper compares both in terms of performance, coverage and relevance in the current landscape of parallel computing benchmarks.

3 Testing Setups

3.1 Overview

The benchmark programs were ran on three different platforms namely: A gem5 full-system simulator, a QEMU emulator, and a Raspberry Pi 3 board for reference. The system was modeled with the following specifications:

Table 3-1 Simulation specifications

Feature	Value
CPU	4 x ARMv8 cores
Clock frequency	700MHz
L1 Cache	32kB per core
L2 Cache	512kB shared
Cache coherence protocol	MOESI
RAM Memory	1GB
Topology	Bus
Operating System	Raspbian OS

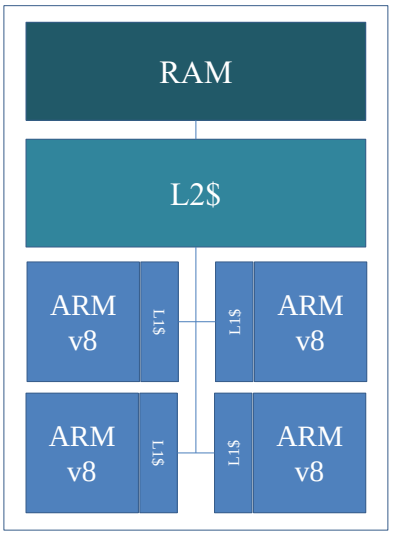


Figure 3-1 Simplified simulation diagram

Both the benchmarks and Butterworth filter were cross-compiled using ARM's embedded-application binary interface (EABI) hard floating point (HF) compiler (`arm-linux-gnueabihf`).

3.1.1 Performance analysis tools

There is a multitude of tools to perform memory statistics, with agility and low footprint in mind, Linux's perf command was chosen. The `perf` command was updated to version 3.18, as it provides an improved version of the `stats` subcommand, which measures the programs' execution time and cache statistics with enough depth to suit the purposes of this study.

All benchmarks were run as: `perf_3.18 stat -d <command>`

In the case of the gem5 setup, the simulated machine cannot access cache statistics, therefore a complementary cache survey was made using the m5 commands provided by gem5. Nonetheless the calls to `perf_3.18` were not removed in order to preserve timings as similarly as possible across platforms. An ARM version of the M5Ops API was included in the gem5 disk image. Cache statistics in gem5 were collected using `m5 resetstats` at the beginning of program execution, and `m5 dumpstats` at the end of execution. This instructs gem5 to collect a multitude of statistics about the program during that period that get dumped into a stats.txt file inside the `m5out` folder. See Appendix C for the full stats dump of the arm_o3 run of the Butterworth filter program.

3.1.2 Disk Images

In order to keep consistency between setups, all of the testing platforms ran a Raspbian Linux distribution. A set of simplified Linux kernels and variations of the same disk image were built for each platform. To normalize testing, nondeterministic hardware aspects such as Dynamic Voltage and Frequency Scaling (DVFS), peripherals, and Bluetooth were disabled. Additional changes had to be done to the kernel used for KVM experiments, as well as to the disk image used for the gem5 tests.

The base disk image was obtained by performing a standard installation using the disk image provided by Raspbian's New Out Of the Box Software (NOOBS) operating system installer [36]. NOOBS' installation generated a disk image with 6 partitions as follows:

Table 3-2 NOOBS disk image partitions

Partition	Size	Type	Purpose
p01	1,1G	W95 FAT	Raspbian Installation iso for first run
p02	13,8G	Extended	Allows more than 4 partitions (Holds logical partitions)
p05	32M	Linux	Settings, installer and pre-boot
p06	63M	W95 FAT	Boot sector, board configuration, kernel, and DTB
p07	13,7G	Linux	Raspbian Linux root

The specifics on how other disk images were created for each setup from the NOOBS image is be presented below.

3.2 Native setup

In order to have a physical multicore platform to compare benchmark results with, a Raspberry Pi 3 model B Board was used as a base reference model. It features a Broadcom BCM28378R1FBG (See Figure 3-3) system-on-chip, with a 1.2 GHz 64-/32-bit quad-core ARM Cortex-A53 (ARMv8), with a 32kB Level 1 cache for each individual core, and a multiport 512kB Level 2 cache shared between all four cores diagram on Figure 3-2.

Figure 3-3 Raspberry Pi 3 Model B

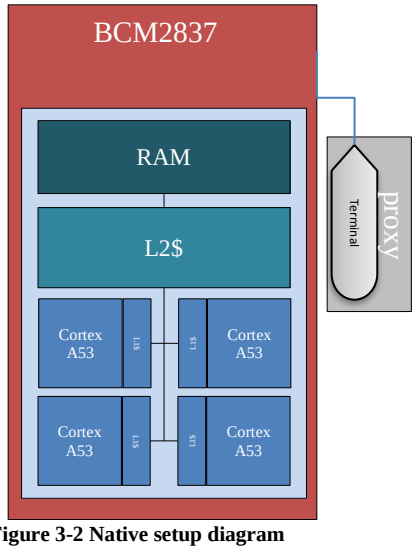


Figure 3-2 Native setup diagram

ARMv8 is the most recent and best supported ARM version in gem5, with the caveat of being limited to a maximum of four cores in gem5's Full-System mode. Additionally, QEMU 2.7.0 supports all AArch64 and AArch32 architectures, including the Cortex-A53. It also includes a machine model for the Raspberry Pi 2 board, which is identical to the Raspberry Pi 3 board in every aspect except its processors, which can be configured to match the reference model's (Cortex-A53). Its high compatibility with gem5 and QEMU, and its ease of configuration and deployment make a Raspberry Pi 3 running Raspbian Linux a suitable choice to use as reference architecture.

The native setup runs on a mostly unmodified NOOBS disk image, with the only major change being the DVFS was disabled to consistently underclick the Raspberry Pi 3 to 700MHz by changing /boot/config.txt Line 43 to : `arm_freq=700`.

3.3 Gem5 setup

3.3.1 Gem5 build

For the gem5 tests, an unmodified gem5 stable_2015_09_03 build was utilized. A standard gem5 ARM build for debugging and optimization was compiled as follows:

```
scons build/ARM/gem5.opt
```

The gem5 model used gem5's Versatille Express machine model VExpress_GEM5_V1, which represents a generic board running at most 4 ARMv8 CPUs with a maximum of 2GB or RAM memory. In addition, a custom AArch64 kernel and device tree blob (DTB) file had to be compiled to meet gem5's requirements based on the instructions presented in [37]. Since there are currently no Ruby/Garnet modules for ARM CPUs in gem5, classic cache were used, along with the default bus topology. The default topology and caches are suitable for these tests, as they match the Cortex-A53's MOESI cache coherence protocol [38, Ch. 14.3] and bus topology [38, Ch. 14.4]. A disk image from gem5 was built by replicating partition p07 of the NOOBS image and applying the changes recommended in [39].

3.3.2 Launch command

The gem5 environment was launched with the following command:

```
sudo ./build/ARM/gem5.opt configs/example/fs.py \
    --kernel=/home/vyz/gem5/tmp/linux-arm-gem5/vmlinux \
    --machine-type=VExpress_GEM5_V1 \
     --dtb-file=$PWD/system/arm/dt/armv8_gem5_v1_1cpu.dtb \
     --caches --l2cache  --num-l2caches=1 --l2_size='512kB' \
    --l1d_size='32kB'  --l1i_size='32kB' \
    --disk-image=/media/vyz/OS/gem5raspibenching.img --root-device=/dev/sda\
     --sys-clock='700MHz' --cpu-clock='700MHz' --num-cpus=4
```

3.3.3 Boot Intricacies and Communicating with the Simulation Environment

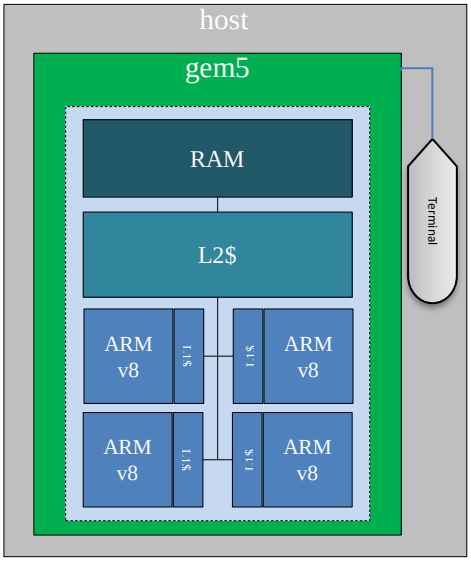


Figure 3-4 Gem5 setup diagram

Once issued the command, another terminal window may be used to establish a telnet connection to the simulation environment through port 3456 with command:

```
m5term localhost 3456
```

During Linux boot, the bootloader will timeout waiting for device `dev-mmcblk0p6`, this behavior is expected, as NOOBS boots on this partition by default, which will be missing in the gem5 single-partition disk image. The file system will nonetheless get properly mounted, and Raspbian will boot in emergency mode as a root user. Where after issueing command `su - pi`, the default pi user will get logged in without credentials, from this point on, commands can be issued as normal.

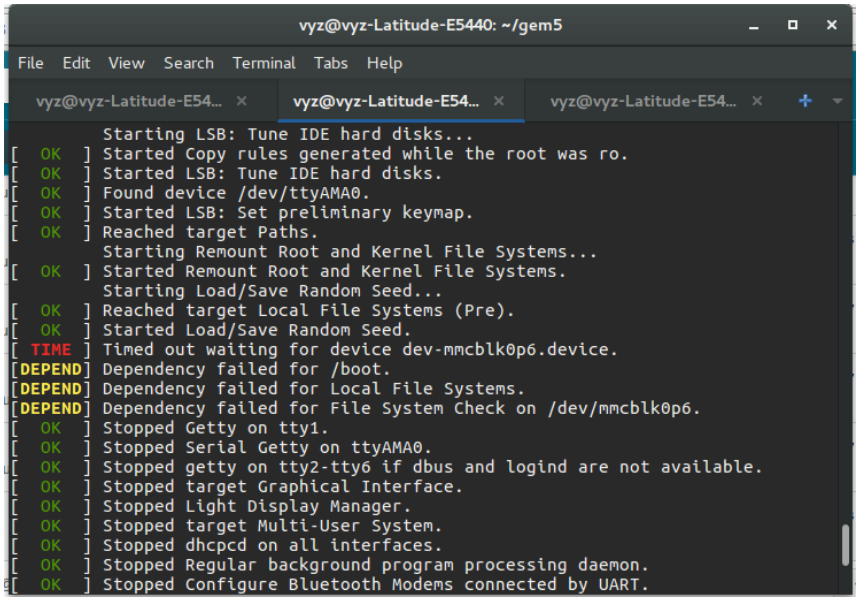


Figure 3-5 Screenshot of Raspbian booting on athe gem5 FS environment. The timeout can be seen in red font.

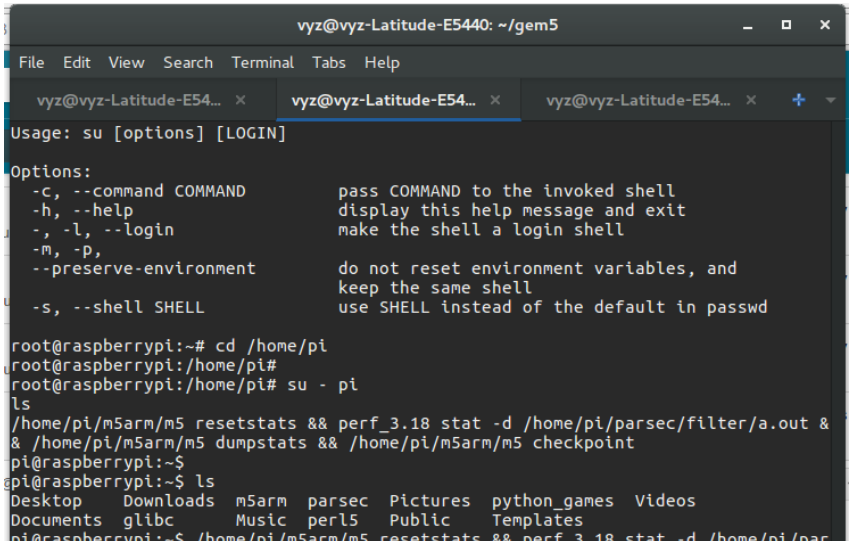


Figure 3-6 Screenshot of Raspbian successfully booting on the gem5 environment, the switch from root to pi user, and launching the Butterworth filterprogram

Once the system boots successfully, command `m5 switchcpu` can be used to switch from the default AtomicSimpleCPU to one of the other models, however it is only possible to access ARM_O3_detailed by providing the `-cpu-type=arm_detailed` argument upon launch. A diagram with the launch system will be generated on m5out, see Appendix D for the full diagram for this system.

3.4 QEMU-KVM setup

QEMU models are entirely optimized for performance, almost entirely neglecting fidelity to the target architecture's hardware. QEMU cannot model processors running at frequencies different from the host's, or featuring hardware components different from those of the host machine. It lacks simulation features such as cache hierarchies and topologies, all of these are inherited from the host machine. Nonetheless, QEMU can still be used for design space exploration to a limited extent when used in conjunction with KVM. KVM allows QEMU to execute commands using one of the cores of the host architecture, achieving almost native performance, and allowing to emulate processors with the same architecture but a different number of cores, or to simulate multiple systems simultaneously; one of the host machine's CPUs must be isolated to enable the host machine to run QEMU-KVM. To test QEMU-KVM, our target architecture was simulated inside the native Raspberry Pi platform, several changes had to be made to the kernel to enable this configuration, a diagram of the emulation scheme can be seen in Figure 3-7. The QEMU tests were carried out using QEMU version 2.7.0-rc2.

3.4.1 Isolating a Core and Enabling KVM

The QEMU-KVM target machine ran using the same image used in the gem5 simulation. The host machine required compiling a new kernel to enable the Cortex-A53's virtualization and KVM capabilities. Following the instructions at [40], we began building a new kernel based on the existing configuration of the Raspberry Pi's existing kernel7:

1. Cloning the Raspberry Pi linux kernel sources.
2. Running `make bcm2709_defconfig` to replicate the default Broadcom 2709 configuration on the build configuration file.
3. Running the interactive configuration script: `make menuconfig`. This opens a UI to manually enable or disable kernel features.
4. Based on [41] the following options were enabled:

- Patch physical to virtual translations at runtime
- General setup -> Control Group support
- System Type -> Support for Large Physical Address Extension
- Boot options -> Use appended device tree blob to zImage (EXPERIMENTAL)
- Boot options -> Supplement the appended DTB with traditional ATAG information
- Device Drivers -> Block devices -> Loopback device support
- Virtualization
- Virtualization -> Kernel-based Virtual Machine (KVM) support.

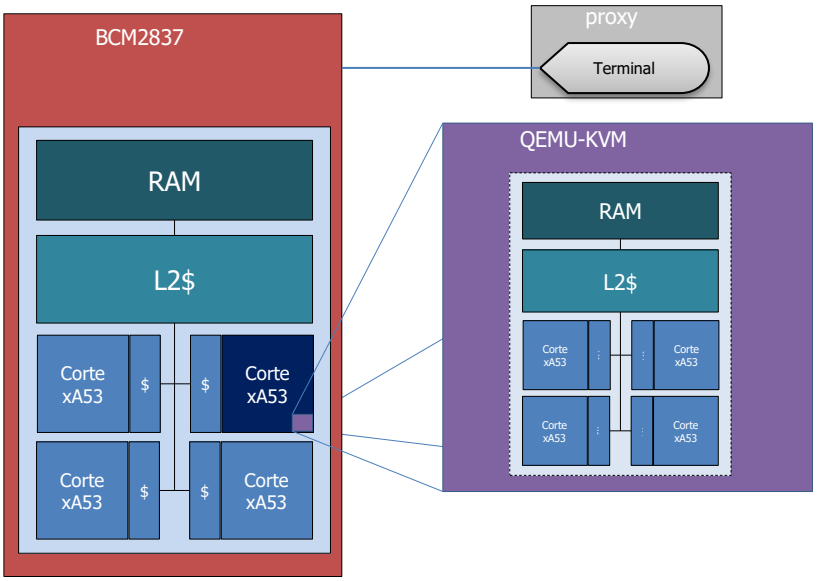


Figure 3-7 QEMU-KVM inside a Raspberry Pi setup. QEMU runs inside the isolated core inside the Raspberry Pi.

Having saved the kernel configuration, the new kernel and DTB files can be copied for deployment on the host board:

```
make -j4 zImage modules dtbs
sudo make modules_install
sudo cp arch/arm/boot/dts/*.dtb /boot/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
sudo scripts/mkknlimg arch/arm/boot/zImage /boot/$KERNEL.img
```

3.4.2 Launch Command

Having the new kernel files installed on the host machine, and a QEMU installation ready, QEMU-KVM is ran with the following command:

```
sudo ./qemu-system-arm  -smp 4 -m 512 -M raspi2 \
   -enable-kvm -cpu host \
   -kernel /home/pi/raspbian-boot/kernel7.img \
   -dtb /home/pi/raspbian-boot/bcm2709-rpi-2-b.dtb \
   -append "root=/dev/sda console=ttyAMA0 rootwait"  \
   -drive if=none,file=/media/pi/4EF2C1E1F2C1CD81/gem5raspibenching.img,\
   id=factory -monitor null -serial stdio –nographic
```

Once the kernel boots, the timeout mentioned in 3.3.3 will occur as well, and the system will continue to boot in the same fashion. It is also worth noticing the the DTB for a Raspberry Pi 2 was used instead, as the Raspberry Pi 3 DTB presented crashed during boot, both files were automatically generated while during kernel compilation. Also, RAM memory had to be reduced, as utilizing the same amount of RAM memory as the host system significantly slowed down performance.

4 Results

4.1 Butterworth Filter Results

The data reported by perf_3.18 stat was used for the native and QEMU-KVM statistics.

For the gem5 statistics, the branch miss rate and L1 miss rate are not stated explicitly in the stats.txt file the values were computer from other statistics:

$$BranchMissRate = \frac{1}{4}\sum_{i=0}^{3}\frac{cpu_i.branchPred.condIncorrect + cpu_i.branchPred.indirectMispredicted}{cpu_i.branchPred.lookups + cpu_i.branchPred.indirectLookups}$$

$$L1Accesses = \sum_{i=0}^{3} cpu_i.icache.tags.data\_accesses + cpu_i.dcache.tags.data\_accesses$$

$$L1MissRate = \frac{1}{4}\sum_{i=0}^{3}\frac{cpu_i.icache.overall\_miss\_rate \ * \ cpu_i.icache.tags.data\_accesses}{L1Accesses} + \frac{cpu_i.dcache.overall\_miss\_rate \ * \ cpu_i.dcache.tags.data\_accesses}{L1Accesses}$$

	Native	gem5 Atomic	gem5 O3	gem5 Timing	gem5 ARM_O3	QEMU-KVM
Task-clock (msec)	15.89	26.31	33.72	79.80	16.89	23.74
Context Switches	450	176	155	124	112	9,923
Page Faults	52090	48,995	48,995	48,995	48,995	56,679
Cycles	48722596486	1029443957	40474636802	95794409876	20304279721	48801763938
Instructions	30757857764	4741482507	31202267806	30996374858	32427013613	30950193237
Branch misses	2.50%	20.63%	Not reported	Not reported	17.59%	3.11%
L1-cache miss ratio	0.04%	0.25%	0.14%	0.15%	1.29%	0.05%
LLC-cache miss ratio	17.48%	24.1557	9.71%	9.84%	6.70%	20.30%
Wall-clock Time (seconds)	15.89	4858.74	21911.94	40529.07	124294.18	23.74

Table 4-1 Butterworth filter results

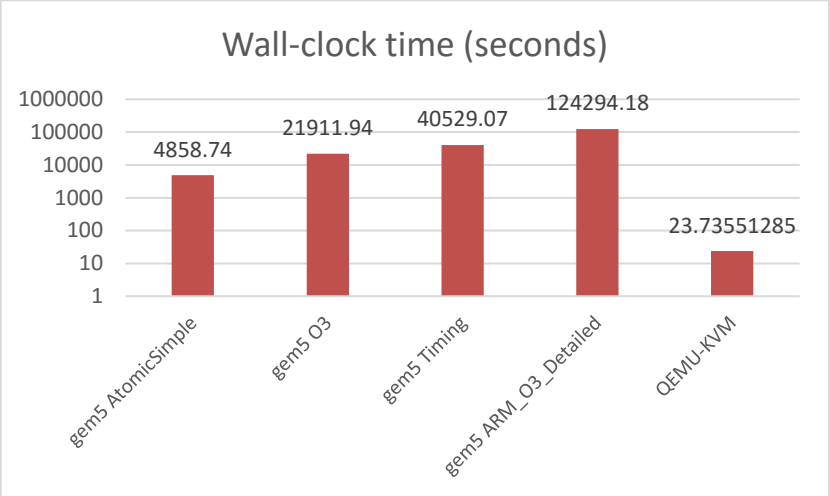


Figure 4-3Wall-clock time in seconds (The smaller the faster).

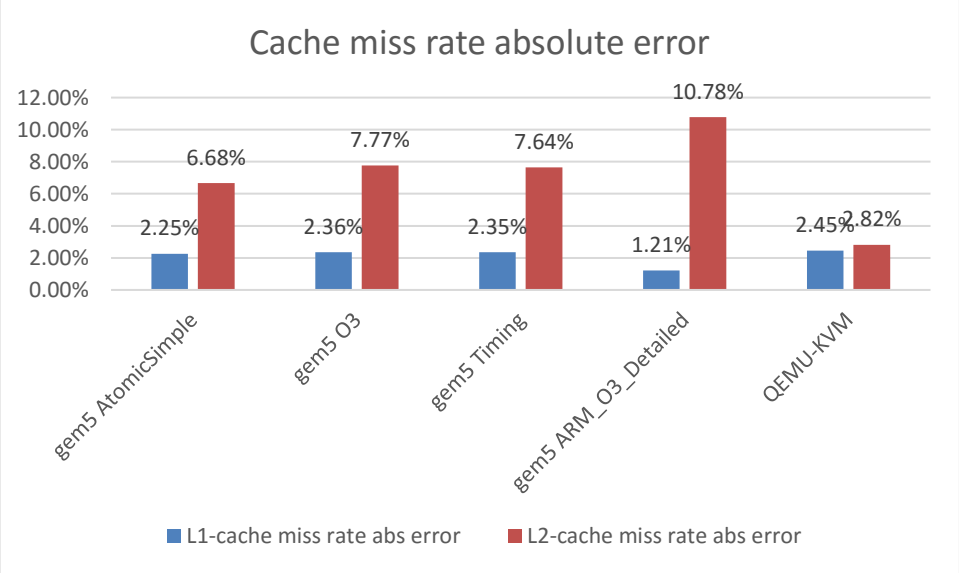


Figure 4-2 L1 and L2 cache miss rates absolute errors (The smaller the better).

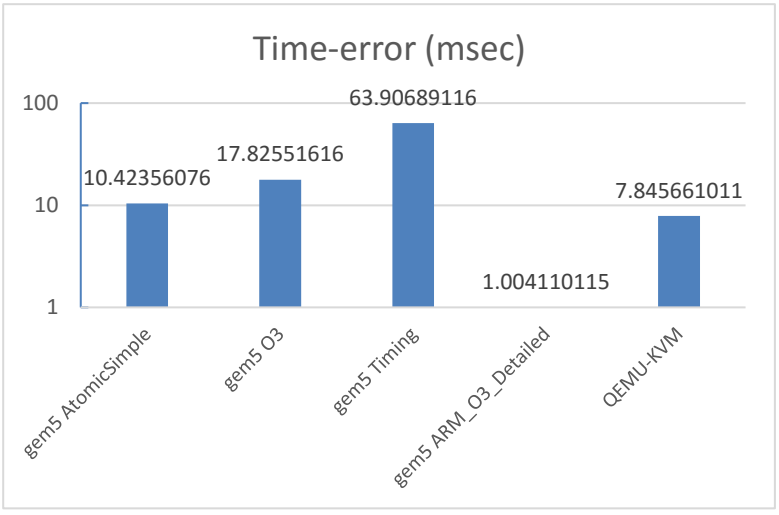


Figure 4-1 Timing absolute error (The smaller the better).

4.2 PARSEC-3.0 Results

The PARSEC-3.0 benchmarks suite was compiled using the arm-gnueabihf-gcc compiler. The benchmarks were ran with PThreads parallelization for all the programs that supported it, and standard gcc parallelization for those that did not. The simdev input set was used for all tests, to keep the benchmarks short and relevant and to be able to simulate all the programs with limited computing resources.

	Native	*QEMU-KVM*	*gem5 AtomicSimple*
Blackscholes	3.985753201	6.30942189	1.491913995
Bodytrack	3.00261287	3.513820372	1.896859488
Canneal	3.124369336	2.939644298	1.433103602
Dedup	2.936092132	3.20508842	1.697196325
Facesim	8.0887201	10.97898935	27.68273514
Ferret	5.383012795	5.990720574	1.581884743
Fluidanimate	3.249598865	3.976957451	1.948901535
Freqmine	3.281712041	2.999685244	1.48596255
Streamcluster	2.654575941	8.149727671	12.4909651
Swaptions	2.631475326	3.149110677	1.424015404
Vips	3.124119867	3.954280245	1.627009393
X264	2.832238671	3.23695357	1.447588245

Table 4-2 PARSEC-3.0 Benchmark suite timing results

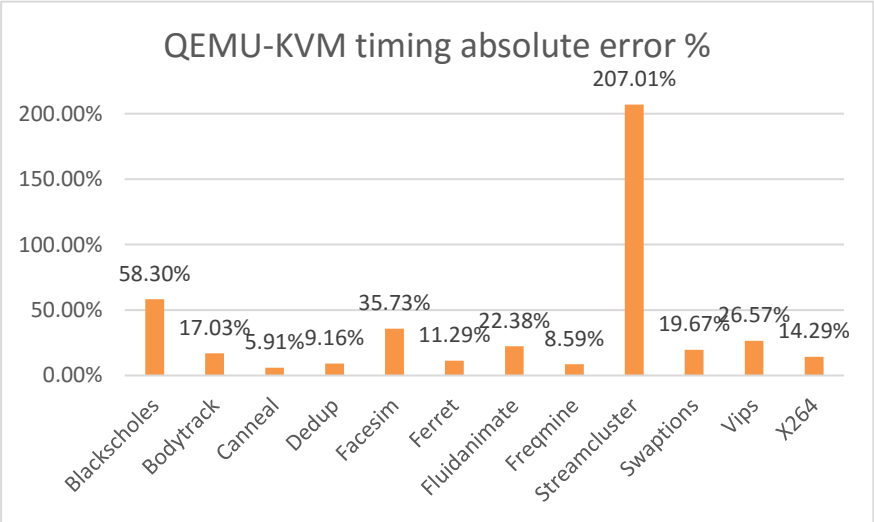


Figure 4-4 QEMU-KVM absolute error percentage. Lower is better.

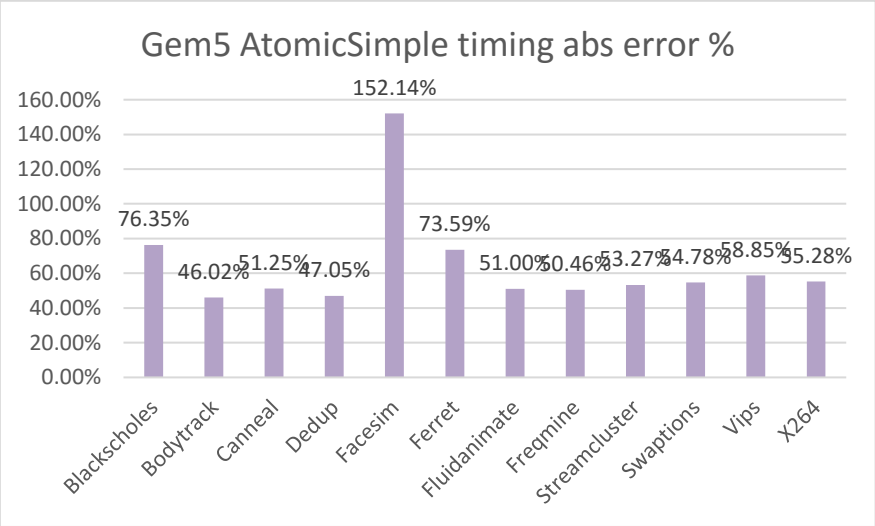


Figure 4-5 Gem5 AtomicSimpleCPU absolute error percentage. Lower is better.

5 Analysis

5.1 QEMU-KVM

The QEMU-KVM emulator benchmarks show promising performance statistics, paired with a high level of accuracy when it comes to non-timing aspects of the target CPU. The lack of timing compliance can be attributed to the KVM executing on an isolated core, and the lack of a precise target time model in QEMU, this is a consistent source of error thorough the QEMU-KVM tests. The Streamcluster PARSEC benchmarks shows a particularly high error percentage (See Figure 4-4). This increased error percentage may be attributed to the large quantity of private reads in a data parallel setting, where KVM has to simulate four cores with four distinct private memory sets using a single host core.

5.2 AtomicSimpleCPU

Gem5's AtomicSimple proved to be a choice decently balanced between performance and speed, it outperformed the more detailed TimingSimple and O3 models in both categories. When running the PARSEC benchmark, AtomicSimple consistently returned lower timing values, and had a much lower wall-clock time cost. The errors in AtomicSimple can be attributed to the atomic modeling of its memory accesses, which progressively stacks as it runs programs with more and longer memory latencies. There is a notable correlation between coarse parallelism and higher margins of error in Figure 4-5. This behavior is especially notable in the Facesim benchmark, where the error percentage spikes to more than twice of its average. This can be attributed to the combination of coarse parallelization granularity and a large working set in a data-parallel application. AtomicSimple's idealized memory accesses reflect poorly the real task overhead of a program with these characteristics.

5.3 TimingSimpleCPU

TimingSimpleCPU comes with multiple problems, the main one being its timing estimation, which depends on pre-calculated generic cache statistics, which end up severely overestimating memory access times. In addition, its model lacks branch statistics, making it less informative than the other models. It is less accurate than AtomicSimple and comes at a higher wall-clock time cost.

5.4 O3CPU

Gem5's Out-of-Order (O3) CPU, contrary to what the gem5 documentation would make intuitive, stands in the middle between TimingSimple and Atomic in terms of accuracy and speed. It also lacks branch statistics.

5.5 ARMO3CPU

The ARM Out-of-Order (O3) detailed CPU model is optimized to closely resemble the out-of-order pipeline of an ARMv7 CPU, which yield the highest timing accuracy out of all the benchmarked models. As seen in Figure 4-3, the gem5 ARM O3 CPU model has the most faithful timing model by a considerable margin. The level of detail exhibited by the ARM O3 CPU timing comes at the price of a colossal wall-clock time, requiring over 34.5 hours to

complete the program. Its memory statistics remain on par with the other gem5 models, as they operate on idealized versions of the MOESI protocol.

6 Conclusions

Both gem5 and QEMU-KVM proved to be suitable tools for simulating ARM multicore architectures to varying degrees of efficiency and accuracy depending on the needs and resources of the application being modeled.

In scenarios where efficiency is paramount, and a KVM-supported host machine with the same type of architecture as the target is available, then QEMU-KVM is a viable choice for full-system simulation, despite being an emulator.

If a higher level of compliance to timing is required, and enough processing resources are available, gem5's ARMO3CPU is a highly desirable choice, which may further improve in the future with the addition of a more accurate Ruby memory model, and Garnet networks.

For scenarios with limited resources and intermediate accuracy needs, gem5's AtomicSimple CPU can be a viable balanced choice, with a manageable time demand and a very simple setup for a full system simulator. Despite being presented as more detailed alternatives to AtomicSimple, the benchmarks show that there is no advantage in terms of accuracy nor speed of using the TimingSimple or O3 models for this architecture. Experimentation with multiple CPU models is advised when simulating with gem5, as accuracy outcomes may yield unexpected results.

My final subjective assessment on both virtual platforming tools can be summarized with the radar charts on Figure 6-1. Gem5 continues to be the preferred virtual platform for accurate and extensible simulation, albeit with a steep learning curve. In comparison, QEMU is mostly inflexible and won't yield reliable memory profile information unless KVM with a matching host is used, on the other hand, it will outperform gem5 in most cases and is a far more user friendly tool with more documentation and the option of having a user interface.

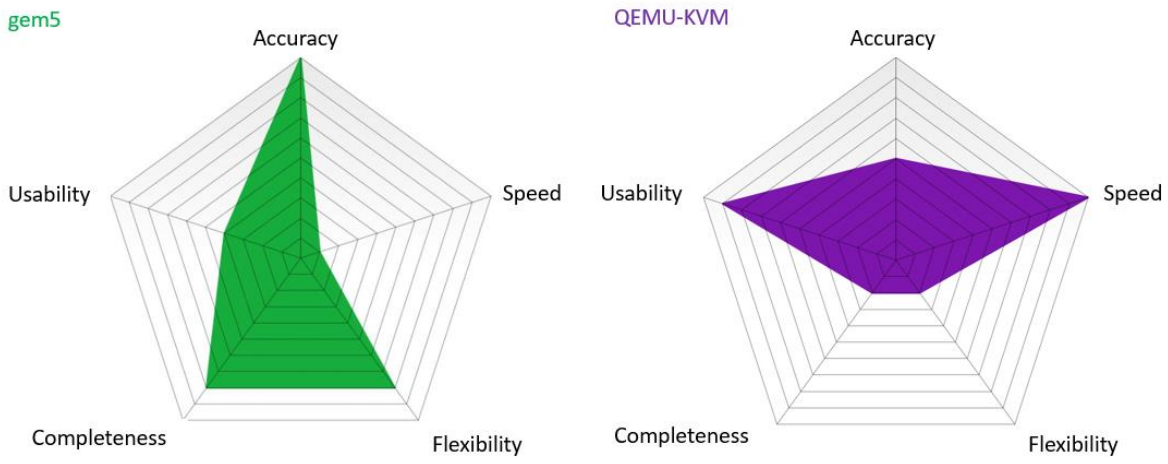


Figure 6-1 Final assessment radar charts

7 Failed Attempts: FPGA-Assisted Simulation

This thesis was originally going to be a broader evaluation of all types of simulators, hence, FPGA-based and FPGA-assisted simulation were pre-evaluated for viability. Due to their low maturity resulting in low usability, and their lack of flexibility, documentation and general relevance or support, these tools were discarded. For the reasons explained on Table 7-1 Pre-evaluation of FPGA-based and FPGA-assisted simulation tools.

7.1.1 FPGA-based Simulation

Field Programmable Gate Arrays (FPGAs)-based simulation is a strong candidate for the next evolutionary trend in simulation. FPGAs are programmable hardware devices that can be used to implement arbitrary logic functions, FPGAs can be modified and incrementally debugged with an ease comparable to software development. FPGA functions can perform multiple orders of magnitude faster than their software-only counterparts, and with a much smaller cost than that of producing a dedicated circuit. With the continuous improvement of FPGAs in recent years, authors such as [42] propose taking advantage of the flexibility and potential for hardware parallelism of FPGAs to provide highly efficient and accurate computer architecture simulations. As illustrated in [7], workload can be partitioned in a variety of ways to best fit the simulated architecture, [7, Ch. 2.5] presents the following ways:

Spatial Partitioning

Is the most intuitive way of partitioning. Allows the simulator to take advantage of the target system's inherent parallelism by making partitions based on the spatial boundaries of the target system, e.g. per processing unit, per group of cores that share a cache level etc.

Temporal Partitioning

If the target application allows it. Temporal partition can split a single performance simulation into a set of parallel simulations, each simulating a different portion of the target time. Another option is to distribute multiple independent experiments with different targets or conditions into different simulations running at once.

Functional/Timing Partitioning

Can partition the system simulation into its two major high-level models. Having a partition that simulates the system's functionality, and another one that simulates its timing and low-level performance aspects, such as power, temperature, deadlines etc. This type of partition is inspired by the fact that functionality often changes slowly, in contrast, the micro architecture changes frequently. Functional/Timing partitioning allows for great flexibility in testing different micro-architectural constraints independently.

Hybrid Partitioning

It is possible and often favorable to combine multiple partitioning approaches to test different aspects of the system both in isolation and in conjunction.

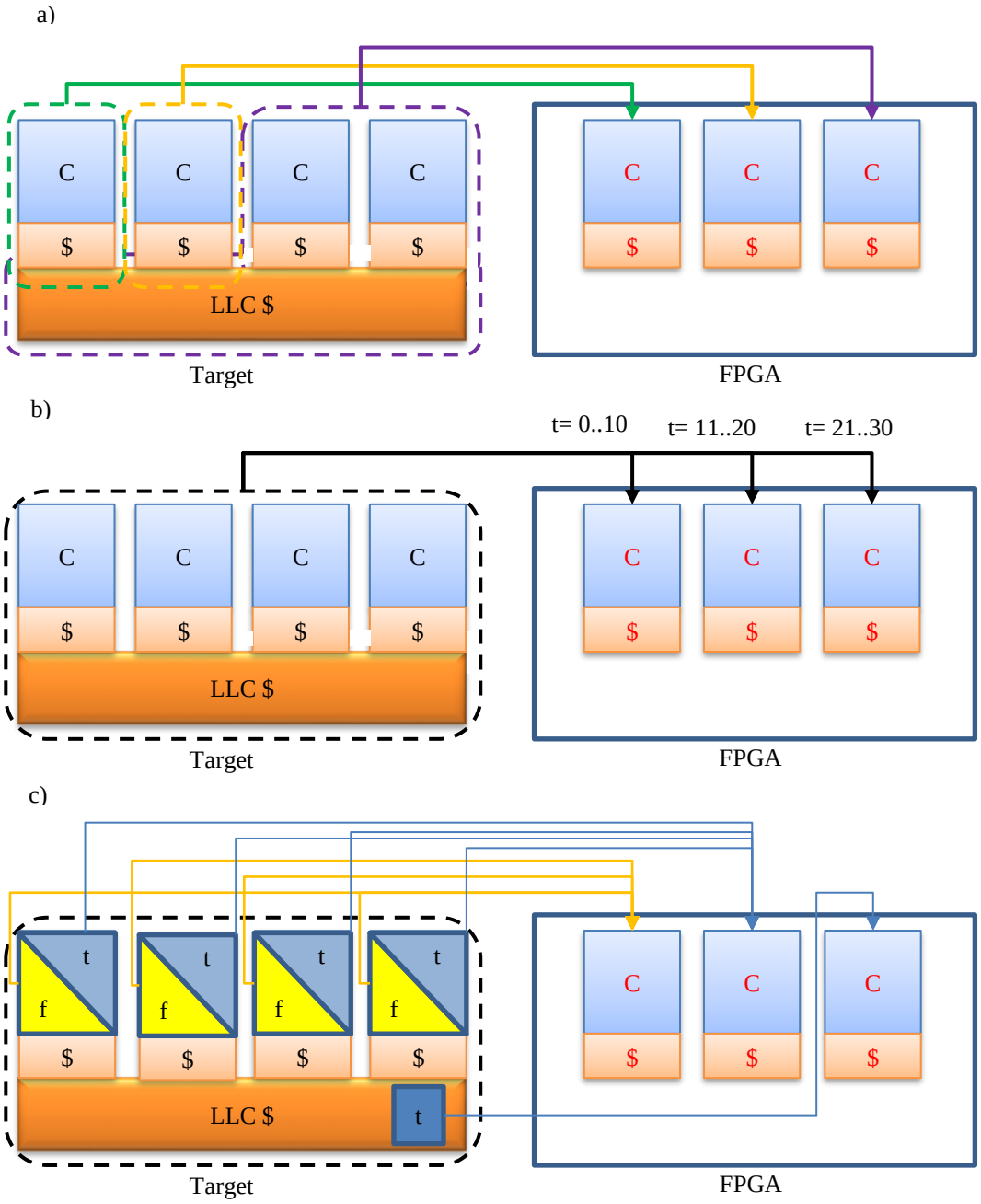


Figure 7-7-1The three main types of FPGA-based simulation partitioning

7.1.2 FPGA-Assisted Simulation

Combining traditional computer architecture simulation with FPGA-based simulation brings forth a highly efficient approach with the benefits of both. FPGA-assisted simulation uses both a software simulator executing in a host computer's hard core, and an FPGA that may contain soft cores and hardware accelerators. As depicted in Figure 7-1, simulated

partitions may be distributed between both the FPGA and the host, allowing for a large number of possible partitions. Different FPGA-assisted simulators support different types of partitions.

The failed attempts made on tools not included in the thesis report can be summarized as follows:

1. **HASim: FPGA-assisted gem5→** Unusable, only working implementation for Alpha architecture, never made it work in the DE2 board.
2. **Full system gem5+Ruby+Garnet Intel Haswell quad-core with 3 level MESIF protocol. →** SLICC has no documentation. Boot time with patched 3-level MESI > 5 days. Not enough time/processing resources.
3. **Including Simics tests →** No access to a recent/relevant version of Simics

Name	Scope & Abstraction Level	Software simulator	Partition Mapping	Status	Comments
HASim	Full-system cycle-accurate	Gem 5	Functional(FPGA/SW)/Timing(FPGA/SW)	Only model: AtomicSimpleCPU for Alpha architectures. No Cyclone IV Support	high-detail multicore simulator that uses time-division multiplexing
ProtoFlex	Full-system functional	Simics	Functional(FPGA) /Timing(SW)	Exclusive to Ultra SPARC ISA	Is a Simics-like open source simulator which includes FPGA acceleration via transplanting (see [43]).
RAMP-gold	Full-system cycle-accurate	-	Functional(FPGA) /Timing(FPGA)	Exclusive to SPARC v8 ISA	FPGA-based simulator directed by the host.

Table 7-1 Pre-evaluation of FPGA-based and FPGA-assisted simulation tools.

8 References

[1] “MATLAB - MathWorks - MathWorks Nordic,” *Matlab - MathWorks*, 2016. [Online]. Available: http://se.mathworks.com/products/matlab/. [Accessed: 18-Apr-2016]

[2] “Parallel Computing Toolbox™ User’s Guide.” Mathworks, Mar-2016.

[3] M. Adler, M. Pellauer, K. E. Fleming, A. Parashar, and J. Emer, “HAsim: FPGA-Based Micro-Architecture Simulator.” Intel, 19-Jan-2012 [Online]. Available: http://www.gem5.org/wiki/images/1/19/201212_HAsim_GEM5.pdf. [Accessed: 17-Apr-2016]

[4] J. Banks, Ed., *Discrete-event system simulation*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2001.

[5] A. T. Velte, T. J. Velte, and R. C. Elsenpeter, *Cloud computing: a practical approach*. New York: McGraw-Hill, 2010.

[6] R. Leupers and O. Temam, Eds., *Processor and System-on-Chip Simulation*. Boston, MA: Springer US, 2010 [Online]. Available: http://link.springer.com/10.1007/978-1-4419-6175-4. [Accessed: 10-Apr-2016]

[7] H. Angepat, D. Chiou, E. S. Chung, and J. C. Hoe, *FPGA-accelerated simulation of computer systems*. 2014 [Online]. Available: http://site.ebrary.com/id/10927924. [Accessed: 10-Apr-2016]

[8] “What is emulation? | Koninklijke Bibliotheek.” [Online]. Available: https://www.kb.nl/en/organisation/research-expertise/research-on-digitisation-and-digital-preservation/emulation/what-is-emulation. [Accessed: 19-Sep-2016]

[9] D. Mosberger, “Memory consistency models,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 27, no. 1, pp. 18–26, Jan. 1993 [Online]. Available: http://portal.acm.org/citation.cfm?doid=160551.160553. [Accessed: 21-Sep-2016]

[10] Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979 [Online]. Available: http://ieeexplore.ieee.org/document/1675439/. [Accessed: 23-Sep-2016]

[11] P. W. Hutto and M. Ahamad, “Slow memory: weakening consistency to enhance concurrency in distributed shared memories,” 1990, pp. 302–309 [Online]. Available: http://ieeexplore.ieee.org/document/89297/. [Accessed: 23-Sep-2016]

[12] M. Dubois, C. Scheurich, and F. Briggs, “Memory access buffering in multiprocessors,” *ACM SIGARCH Comput. Archit. News*, vol. 14, no. 2, pp. 434–442, Jun. 1986 [Online]. Available: http://portal.acm.org/citation.cfm?doid=17356.17406. [Accessed: 23-Sep-2016]

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, “Memory consistency and event ordering in scalable shared-memory multiprocessors,” *ACM SIGARCH Comput. Archit. News*, vol. 18, no. 3, pp. 15–26, May 1990 [Online]. Available: http://portal.acm.org/citation.cfm?doid=325096.325102. [Accessed: 23-Sep-2016]

[14] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, “The gem5 simulator,” *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1, Aug. 2011 [Online]. Available: http://dl.acm.org/citation.cfm?doid=2024716.2024718. [Accessed: 08-Aug-2016]

[15] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, “The M5 Simulator: Modeling Networked Systems,” *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul. 2006 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1677503. [Accessed: 08-Aug-2016]

[16] 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," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, p. 92, Nov. 2005 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1105734.1105747. [Accessed: 08-Aug-2016]
[17]"SimObjects - gem5." [Online]. Available: http://m5sim.org/SimObjects. [Accessed: 21-Sep-2016]
[18] "The M5 ISA description language - gem5." [Online]. Available: http://www.m5sim.org/The_M5_ISA_description_language. [Accessed: 21-Sep-2016]
[19] "SLICC - gem5." [Online]. Available: http://www.m5sim.org/SLICC. [Accessed: 21-Sep-2016]
[20] "Execution Basics - gem5." [Online]. Available: http://www.gem5.org/Execution_Basics#ExecContext. [Accessed: 08-Aug-2016]
[21]"SimpleCPU - gem5." [Online]. Available: http://www.m5sim.org/SimpleCPU. [Accessed: 21-Sep-2016]
[22] "InOrder - gem5." [Online]. Available: http://www.m5sim.org/InOrder. [Accessed: 21-Sep-2016]
[23] "O3CPU - gem5." [Online]. Available: http://www.gem5.org/O3CPU. [Accessed: 08-Aug-2016]
[24] "General Memory System - gem5." [Online]. Available: http://www.m5sim.org/General_Memory_System. [Accessed: 08-Aug-2016]
[25] "Interconnection Network - gem5." [Online]. Available: http://www.m5sim.org/Interconnection_Network. [Accessed: 23-Sep-2016]
[26] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," 2009, pp. 33–42 [Online]. Available: http://ieeexplore.ieee.org/document/4919636/. [Accessed: 24-Sep-2016]
[27] "Dependencies - gem5." [Online]. Available: http://www.m5sim.org/Dependencies. [Accessed: 25-Sep-2016]
[28] "Running gem5 - gem5." [Online]. Available: http://www.m5sim.org/Running_gem5. [Accessed: 25-Sep-2016]
[29] "Features/Memory API - QEMU." [Online]. Available: http://wiki.qemu.org/Features/Memory_API. [Accessed: 08-Aug-2016]
[30] "QEMU Emulator User Documentation." [Online]. Available: http://wiki.qemu.org/download/qemu-doc.html. [Accessed: 25-Sep-2016]
[31]C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," 2008, p. 72 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1454115.1454128. [Accessed: 08-Aug-2016]
[32] "Manpage of PARSECMGMT." [Online]. Available: http://parsec.cs.princeton.edu/doc/man/man1/parsecmgmt.1.html. [Accessed: 25-Sep-2016]
[33] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," 2012, pp. 1–7 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6322869. [Accessed: 06-Aug-2016]
[34] Y. Hu, H. Jin, Z. Yu, and H. Zheng, "An Optimization Approach for QEMU," 2009, pp. 129–132 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5455191. [Accessed: 10-Aug-2016]
[35] C. Bienia, S. Kumar, and Kai Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors," 2008, pp. 47–56 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4636090. [Accessed: 06-Aug-2016]
[36] "Download NOOBS for Raspberry Pi," *Raspberry Pi*. [Online]. Available: https://www.raspberrypi.org/downloads/noobs/. [Accessed: 09-Aug-2016]

[37] “ARM Linux Kernel - gem5.” [Online]. Available: http://www.gem5.org/ARM_Linux_Kernel. [Accessed: 09-Aug-2016]
[38] “ARM Information Center.” [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch14s04.html. [Accessed: 10-Aug-2016]
[39] “Ubuntu Disk Image for ARM Full System - gem5.” [Online]. Available: http://www.gem5.org/Ubuntu_Disk_Image_for_ARM_Full_System. [Accessed: 10-Aug-2016]
[40] “Kernel Building - Raspberry Pi Documentation.” [Online]. Available: https://www.raspberrypi.org/documentation/linux/kernel/building.md. [Accessed: 10-Aug-2016]
[41]“Enabling KVM virtualization for Raspberry Pi 2 – flexVDI.” [Online]. Available: http://blog.flexvdi.com/2015/03/17/enabling-kvm-virtualization-on-the-raspberry-pi-2/. [Accessed: 08-Aug-2016]
[42] “DART: A Programmable Architecture for NoC Simulation on FPGAs,” *IEEE Trans. Comput.*, vol. 63, no. 3, pp. 664–678, Mar. 2014 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6212455. [Accessed: 11-Apr-2016]
[43] Carnegie Mellon Electrical and Computer Engineering Department, “Home - ProtoFlex,” *Protoflex*, 25-May-2014. [Online]. Available: http://www.ece.cmu.edu/~protoflex/doku.php. [Accessed: 19-Apr-2016]

Appendix A: Butterworth Filter Matlab Code

```
% filters each column of the matrix "x" with filter b,a
%
% assumptions: a, b are row vectors
% size(x,1) > 3*(max(length(a), length(b)) -1)

function filtfilt_bn

[b,a] = butter(6, 0.5);
b
a
x = rand(5000,5000);
%x = x';
%x

tic
y = filtfiltbn(b,a,x);
toc

%y-x

end

% filters each column of the matrix "x" with filter b,a
%
% assumptions: a, b are row vectors
% size(x,1) > 3*(max(length(a), length(b)) -1)


% filters each column of the matrix "x" with filter b,a
%
% assumptions: a, b are row vectors
% size(x,1) > 3*(max(length(a), length(b)) -1)


function y = filtfiltbn(b,a,x)


[nt,nx] = size(x);

nb = length(b);
na = length(a);
nfilt = max(nb,na);
nfact = 3*(nfilt-1);  % length of edge transients

bpad = zeros(1,nfilt);
apad = zeros(1,nfilt);
bpad(1:nb) = b;
apad(1:na) = a;

zi = ( eye(nfilt-1) - [-apad(2:nfilt).' [eye(nfilt-2); zeros(1,nfilt-2)]] ) \ ...
    ( bpad(2:nfilt).' - apad(2:nfilt).'*bpad(1) );

y = zeros(nt,nx);
for k=1:nx
    xpad = [2*x(1,k)-x((nfact+1):-1:2,k);x(:,k);2*x(nt,k)-x((nt-1):-1:nt-nfact,k)];

    xpad = filteric1D(bpad,apad,xpad, zi*xpad(1));
    xpad = flipud(filteric1D(bpad,apad, flipud(xpad), zi*xpad(end)));

    y(:,k) = xpad(nfact+(1:nt));
end
```

```
end



function y = filteric1D(b,a,x,zi)
np = size(x,1);
y = zeros(np,1);
ord = length(zi);

for m=1:ord
    y(m) = b(1)*x(m)+zi(1);
    for n=1:ord-1
        zi(n) = b(n+1)*x(m) + zi(n+1) - a(n+1)*y(m);
    end
    zi(ord) = b(ord+1)*x(m) - a(ord+1)*y(m);
end

for i=ord+1:np
    y(i) = 0.0;
    for j = 0:ord
        y(i) = y(i) + b(j+1) * x(i - j);
    end
    for j = 1:ord
        y(i) = y(i) - a(j+1) * y(i - j);
    end
end

end
```

Appendix B: Butterworth Filter Generated C code

```
/*
 * File: filtfiltMain.c
 *
 * MATLAB Coder version            : 2.6
 * C/C++ source code generated on  : 27-May-2016 11:30:17
 */

/*
 * Include files
 */
#
include "filtfiltMain.h"

/*
 * Variable Definitions
 */
static unsigned int method;
static unsigned int state;
static unsigned int b_state[2];
static unsigned int c_state[625];
static boolean_T state_not_empty;
omp_nest_lock_t emlrtNestLockGlobal;

/*
 * Function Declarations
 */
static void b_eye(double I[25]);
static void b_flipud(double x[5036]);
static void b_rand(double r[25000000]);
static void eml_rand_init(void);
static void eml_rand_mcg16807_stateful_init(void);
static double eml_rand_mt19937ar(unsigned int d_state[625]);
static void eml_rand_shr3cong_stateful_init(void);
static void eye(double I[36]);
static void filteric1D(const double b[7],
  const double a[7],
    const double x[5036], double zi[6],
      double y[5036]);
static void flipud(const double x[5036], double b_x[5036]);
static void mldivide(const double A[36], double B[6]);
static void twister_state_vector(unsigned int mt[625], double seed);

/*
 * Function Definitions
 */

/*
 * Arguments    : double I[25]
 * Return Type  : void
 */
static void
b_eye(double I[25]) {
  int k;
  memset( & I[0], 0, 25 U * sizeof(double));
  for (k = 0; k < 5; k++) {
    I[k + 5 * k] = 1.0;
  }
}
```

```
/*
 * Arguments    : double x[5036]
 * Return Type  : void
 */
static void
b_flipud(double x[5036]) {
  int i;
  double xtmp;
  for (i = 0; i < 2518; i++) {
    xtmp = x[i];
    x[i] = x[5035 - i];
    x[5035 - i] = xtmp;
  }
}

/*
 * Arguments    : double r[25000000]
 * Return Type  : void
 */
static void
b_rand(double r[25000000]) {
  int k;
  int hi;
  unsigned int test1;
  unsigned int test2;
  if (method == 4 U) {
    for (k = 0; k < 25000000; k++) {
      hi = (int)(state / 127773 U);
      test1 = 16807 U * (state - hi * 127773 U);
      test2 = 2836 U * hi;
      if (test1 < test2) {
        state = (test1 - test2) + 2147483647 U;
      } else {
        state = test1 - test2;
      }

      r[k] = (double) state * 4.6566128752457969E-10;
    }
  } else if (method == 5 U) {
    for (k = 0; k < 25000000; k++) {
      test1 = 69069 U * b_state[0] + 1234567 U;
      test2 = b_state[1] ^ b_state[1] << 13;
      test2 ^= test2 >> 17;
      test2 ^= test2 << 5;
      b_state[0] = test1;
      b_state[1] = test2;
      r[k] = (double)(test1 + test2) * 2.328306436538696E-10;
    }
  } else {
    if (!state_not_empty) {
      memset( & c_state[0], 0, 625 U * sizeof(unsigned int));
      twister_state_vector(c_state, 5489.0);
      state_not_empty = true;
    }

    for (k = 0; k < 25000000; k++) {
      r[k] = eml_rand_mt19937ar(c_state);
    }
  }
}
```

```
/*
 * Arguments    : void
 * Return Type  : void
 */
static void
eml_rand_init(void) {
  method = 7 U;
}

/*
 * Arguments    : void
 * Return Type  : void
 */
static void
eml_rand_mcg16807_stateful_init(void) {
  state = 1144108930 U;
}

/*
 * Arguments    : unsigned int d_state[625]
 * Return Type  : double
 */
static double
eml_rand_mt19937ar(unsigned int d_state[625]) {
  double r;
  int32_T exitg1;
  unsigned int u[2];
  int k;
  unsigned int mti;
  int kk;
  unsigned int y;
  unsigned int b_y;
  unsigned int c_y;
  unsigned int d_y;
  boolean_T isvalid;
  boolean_T exitg2;

  /*
   * ========================= COPYRIGHT NOTICE
   * ============================
   */
  /*
   * This is a uniform (0,1) pseudorandom number generator based on:
   */
  /*
   */
  /*
   * A C-program for MT19937, with initialization improved 2002/1/26.
   */
  /*
   * Coded by Takuji Nishimura and Makoto Matsumoto.
   */
  /*
   */
  /*
   * Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
   */
  /*
   * All rights reserved.
   */
```

```
/*
 */
/*
 * Redistribution and use in source and binary forms, with or without
 */
/*
 * modification, are permitted provided that the following conditions
 */
/*
 * are met:
 */
/*
 */
/*
 * 1. Redistributions of source code must retain the above copyright
 */
/*
 * notice, this list of conditions and the following disclaimer.
 */
/*
 */
/*
 * 2. Redistributions in binary form must reproduce the above
 * copyright
 */
/*
 * notice, this list of conditions and the following disclaimer
 */
/*
 * in the documentation and/or other materials provided with the
 */
/*
 * distribution.
 */
/*
 */
/*
 * 3. The names of its contributors may not be used to endorse or
 */
/*
 * promote products derived from this software without specific
 */
/*
 * prior written permission.
 */
/*
 */
/*
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 */
/*
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 */
/*
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR
 */
/*
 * A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE
 * COPYRIGHT
 */
```

```
/*
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL,
 */
/*
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 */
/*
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE,
 */
/*
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY
 */
/*
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 */
/*
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
 * USE
 */
/*
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 */
/*
 */
/*
 * ============================== END ================================
 */
do {
  exitg1 = 0;
  for (k = 0; k < 2; k++) {
    mti = d_state[624] + 1 U;
    if (mti >= 625 U) {
      for (kk = 0; kk < 227; kk++) {
        y = (d_state[kk] & 2147483648 U) | (d_state[1 + kk] &
          2147483647 U);
        if ((int)(y & 1 U) == 0) {
          b_y = y >> 1 U;
        } else {
          b_y = y >> 1 U ^ 2567483615 U;
        }

        d_state[kk] = d_state[397 + kk] ^ b_y;
      }

      for (kk = 0; kk < 396; kk++) {
        y = (d_state[227 + kk] & 2147483648 U) |
          (d_state[228 + kk] & 2147483647 U);
        if ((int)(y & 1 U) == 0) {
          c_y = y >> 1 U;
        } else {
          c_y = y >> 1 U ^ 2567483615 U;
        }

        d_state[227 + kk] = d_state[kk] ^ c_y;
      }

      y = (d_state[623] & 2147483648 U) | (d_state[0] &
```

```
          2147483647 U);
        if ((int)(y & 1 U) == 0) {
          d_y = y >> 1 U;
        } else {
          d_y = y >> 1 U ^ 2567483615 U;
        }

        d_state[623] = d_state[396] ^ d_y;
        mti = 1 U;
      }

      y = d_state[(int) mti - 1];
      d_state[624] = mti;
      y ^= y >> 11 U;
      y ^= y << 7 U & 2636928640 U;
      y ^= y << 15 U & 4022730752 U;
      y ^= y >> 18 U;
      u[k] = y;
    }

    r = 1.1102230246251565E-16 * ((double)(u[0] >> 5 U) *
      6.7108864E+7 + (double)
      (u[1] >> 6 U));
    if (r == 0.0) {
      if ((d_state[624] >= 1 U) && (d_state[624] < 625 U)) {
        isvalid = true;
      } else {
        isvalid = false;
      }

      if (isvalid) {
        isvalid = false;
        k = 1;
        exitg2 = false;
        while ((!exitg2) && (k < 625)) {
          if (d_state[k - 1] == 0 U) {
            k++;
          } else {
            isvalid = true;
            exitg2 = true;
          }
        }
      }

      if (!isvalid) {
        twister_state_vector(d_state, 5489.0);
      }
    } else {
      exitg1 = 1;
    }
  } while (exitg1 == 0);

  return r;
}

/*
 * Arguments    : void
 * Return Type  : void
 */
static void
```

```
eml_rand_shr3cong_stateful_init(void) {
  int i3;
  for (i3 = 0; i3 < 2; i3++) {
    b_state[i3] = 362436069 U + 158852560 U * i3;
  }
}

/*
 * Arguments    : double I[36]
 * Return Type  : void
 */
static void
eye(double I[36]) {
  int k;
  memset( & I[0], 0, 36 U * sizeof(double));
  for (k = 0; k < 6; k++) {
    I[k + 6 * k] = 1.0;
  }
}

/*
 * Arguments    : const double b[7]
 *                const double a[7]
 *                const double x[5036]
 *                double zi[6]
 *                double y[5036]
 * Return Type  : void
 */
static void
filteric1D(const double b[7],
  const double a[7],
    const double x[5036],
      double zi[6], double y[5036]) {
  int m;
  int j;

  /*
   * 'filtfiltMain:68' np = size(x,1);
   */
  /*
   * 'filtfiltMain:69' y = zeros(np,1);
   */
  memset( & y[0], 0, 5036 U * sizeof(double));

  /*
   * 'filtfiltMain:70' ord = length(zi);
   */
  /*
   * 'filtfiltMain:72' for m=1:ord
   */
  for (m = 0; m < 6; m++) {
    /*
     * 'filtfiltMain:73' y(m) = b(1)*x(m)+zi(1);
     */
    y[m] = b[0] * x[m] + zi[0];

    /*
     * 'filtfiltMain:74' for n=1:ord-1
     */
    for (j = 0; j < 5; j++) {
```

```
      /*
       * 'filtfiltMain:75' zi(n) = b(n+1)*x(m) + zi(n+1) -
       * a(n+1)*y(m);
       */
      zi[j] = (b[1 + j] * x[m] + zi[1 + j]) - a[1 + j] * y[m];
    }

    /*
     * 'filtfiltMain:77' zi(ord) = b(ord+1)*x(m) - a(ord+1)*y(m);
     */
    zi[5] = b[6] * x[m] - a[6] * y[m];
  }

  /*
   * 'filtfiltMain:80' for i=ord+1:np
   */
  for (m = 0; m < 5030; m++) {
    /*
     * 'filtfiltMain:81' y(i) = 0.0;
     */
    y[6 + m] = 0.0;

    /*
     * 'filtfiltMain:82' for j = 0:ord
     */
    for (j = 0; j < 7; j++) {
      /*
       * 'filtfiltMain:83' y(i) = y(i) + b(j+1) * x(i - j);
       */
      y[6 + m] += b[j] * x[(m - j) + 6];
    }

    /*
     * 'filtfiltMain:85' for j = 1:ord
     */
    for (j = 0; j < 6; j++) {
      /*
       * 'filtfiltMain:86' y(i) = y(i) - a(j+1) * y(i - j);
       */
      y[6 + m] -= a[1 + j] * y[(m - j) + 5];
    }
  }
}

/*
 * Arguments    : const double x[5036]
 *                double b_x[5036]
 * Return Type  : void
 */
static void
flipud(const double x[5036], double b_x[5036]) {
  memcpy( & b_x[0], & x[0], 5036 U * sizeof(double));
  b_flipud(b_x);
}

/*
 * Arguments    : const double A[36]
 *                double B[6]
 * Return Type  : void
 */
```

```
static void
mldivide(const double A[36], double B[6]) {
  double b_A[36];
  signed char ipiv[6];
  int i4;
  int j;
  int c;
  int kAcol;
  int ix;
  double temp;
  int k;
  double s;
  int jy;
  int ijA;
  memcpy( & b_A[0], & A[0], 36 U * sizeof(double));
  for (i4 = 0; i4 < 6; i4++) {
    ipiv[i4] = (signed char)(1 + i4);
  }

  for (j = 0; j < 5; j++) {
    c = j * 7;
    kAcol = 0;
    ix = c;
    temp = fabs(b_A[c]);
    for (k = 2; k <= 6 - j; k++) {
      ix++;
      s = fabs(b_A[ix]);
      if (s > temp) {
        kAcol = k - 1;
        temp = s;
      }
    }

    if (b_A[c + kAcol] != 0.0) {
      if (kAcol != 0) {
        ipiv[j] = (signed char)((j + kAcol) + 1);
        ix = j;
        kAcol += j;
        for (k = 0; k < 6; k++) {
          temp = b_A[ix];
          b_A[ix] = b_A[kAcol];
          b_A[kAcol] = temp;
          ix += 6;
          kAcol += 6;
        }
      }

      i4 = (c - j) + 6;
      for (jy = c + 1; jy + 1 <= i4; jy++) {
        b_A[jy] /= b_A[c];
      }
    }

    kAcol = c;
    jy = c + 6;
    for (k = 1; k <= 5 - j; k++) {
      temp = b_A[jy];
      if (b_A[jy] != 0.0) {
        ix = c + 1;
        i4 = (kAcol - j) + 12;
        for (ijA = 7 + kAcol; ijA + 1 <= i4; ijA++) {
```

```
          b_A[ijA] += b_A[ix] * -temp;
          ix++;
        }
      }

      jy += 6;
      kAcol += 6;
    }
  }

  for (kAcol = 0; kAcol < 5; kAcol++) {
    if (ipiv[kAcol] != kAcol + 1) {
      temp = B[kAcol];
      B[kAcol] = B[ipiv[kAcol] - 1];
      B[ipiv[kAcol] - 1] = temp;
    }
  }

  for (k = 0; k < 6; k++) {
    kAcol = 6 * k;
    if (B[k] != 0.0) {
      for (jy = k + 1; jy + 1 < 7; jy++) {
        B[jy] -= B[k] * b_A[jy + kAcol];
      }
    }
  }

  for (k = 5; k > -1; k += -1) {
    kAcol = 6 * k;
    if (B[k] != 0.0) {
      B[k] /= b_A[k + kAcol];
      for (jy = 0; jy + 1 <= k; jy++) {
        B[jy] -= B[k] * b_A[jy + kAcol];
      }
    }
  }
}

/*
 * Arguments    : unsigned int mt[625]
 *                double seed
 * Return Type  : void
 */
static void
twister_state_vector(unsigned int mt[625], double seed) {
  unsigned int r;
  int mti;
  if (seed < 4.294967296E+9) {
    if (seed >= 0.0) {
      r = (unsigned int) seed;
    } else {
      r = 0 U;
    }
  } else {
    r = MAX_uint32_T;
  }

  mt[0] = r;
  for (mti = 0; mti < 623; mti++) {
    r = (r ^ r >> 30 U) * 1812433253 U + (1 + mti);
```

```
    mt[1 + mti] = r;
  }

  mt[624] = 624 U;
}

/*
 * Arguments    : void
 * Return Type  : void
 */
void
filtfiltMain(void) {
  static double x[25000000];
  double bpad[7];
  double apad[7];
  int i0;
  static
  const double b[7] = {
    0.029588223638660763,
    0.17752934183196459,
    0.44382335457991146,
    0.59176447277321531,
    0.44382335457991146,
    0.17752934183196459,
    0.029588223638660763
  };

  static
  const double a[7] = {
    1.0,
    -1.0495077029659683E-15,
    0.77769596185567358,
    -5.001668708391269E-16,
    0.11419942506243402,
    -7.6242296460891413E-17,
    0.0017509259561828323
  };

  double dv0[36];
  double dv1[25];
  double b_apad[36];
  double zi[6];
  int i1;
  double dv2[36];
  static double xpad[5036];
  int k;
  double d0;
  double d1;
  int i2;
  double b_zi[6];
  int i;
  static double b_xpad[5036];
  double dv3[5036];
  double dv4[5036];
  double dv5[5036];
  double unusedExpr[5036];

  /*
   * filters each column of the matrix "x" with filter b,a
   */
```

```
/*
 */
/*
 * assumptions: a, b are row vectors
 */
/*
 * size(x,1) > 3*(max(length(a), length(b)) -1)
 */
/*
 * 'filtfiltMain:8' [b,a] = butter(6, 0.5);
 */
/*
 * 'filtfiltMain:9' b
 */
/*
 * 'filtfiltMain:10' a
 */
/*
 * 'filtfiltMain:11' x = rand(5000,5000);
 */
b_rand(x);
// Tic
time_t tstart,
tend,
lapse;
tstart = time(0);
/*
 * x = x';
 */
/*
 * x
 */
/*
 * tic
 */
/*
 * 'filtfiltMain:16' y = filtfiltbn(b,a,x);
 */
/*
 * filters each column of the matrix "x" with filter b,a
 */
/*
 */
/*
 * assumptions: a, b are row vectors
 */
/*
 * size(x,1) > 3*(max(length(a), length(b)) -1)
 */
/*
 * filters each column of the matrix "x" with filter b,a
 */
/*
 */
/*
 * assumptions: a, b are row vectors
 */
/*
 * size(x,1) > 3*(max(length(a), length(b)) -1)
 */
/*
```

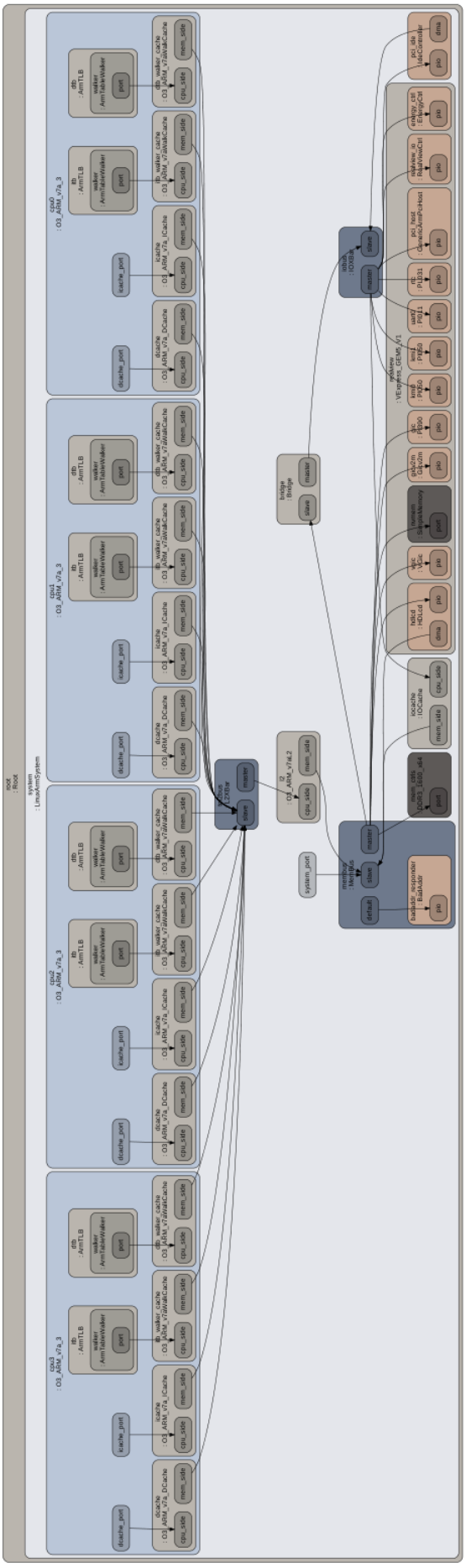
```
 * 'filtfiltMain:38' [nt,nx] = size(x);
 */
/*
 * 'filtfiltMain:40' nb = length(b);
 */
/*
 * 'filtfiltMain:41' na = length(a);
 */
/*
 * 'filtfiltMain:42' nfilt = max(nb,na);
 */
/*
 * 'filtfiltMain:43' nfact = 3*(nfilt-1);
 */
/*
 * length of edge transients
 */
/*
 * 'filtfiltMain:45' bpad = zeros(1,nfilt);
 */
/*
 * 'filtfiltMain:46' apad = zeros(1,nfilt);
 */
/*
 * 'filtfiltMain:47' bpad(1:nb) = b;
 */
for (i0 = 0; i0 < 7; i0++) {
  bpad[i0] = b[i0];

  /*
   * 'filtfiltMain:48' apad(1:na) = a;
   */
  apad[i0] = a[i0];
}

/*
 * 'filtfiltMain:50' zi = ( eye(nfilt-1) - [-apad(2:nfilt).'
 * [eye(nfilt-2); zeros(1,nfilt-2)]] ) \ ...
 */
/*
 * 'filtfiltMain:51' ( bpad(2:nfilt).' - apad(2:nfilt).'*bpad(1) );
 */
eye(dv0);
b_eye(dv1);
for (i0 = 0; i0 < 6; i0++) {
  zi[i0] = bpad[1 + i0] - apad[1 + i0] * bpad[0];
  b_apad[i0] = -apad[1 + i0];
}

for (i0 = 0; i0 < 5; i0++) {
  for (i1 = 0; i1 < 5; i1++) {
    b_apad[i1 + 6 * (i0 + 1)] = dv1[i1 + 5 * i0];
  }

  b_apad[5 + 6 * (i0 + 1)] = 0.0;
}

for (i0 = 0; i0 < 6; i0++) {
  for (i1 = 0; i1 < 6; i1++) {
    dv2[i1 + 6 * i0] = dv0[i1 + 6 * i0] - b_apad[i1 + 6 * i0];
```

```
    }
  }

  mldivide(dv2, zi);

  /*
   * 'filtfiltMain:53' y = zeros(nt,nx);
   */
  #
  pragma omp parallel
  for\
  num_threads(omp_get_max_threads())\
  private(xpad, d0, d1, i2, i)\
  firstprivate(b_xpad, b_zi, dv3, dv4, dv5, unusedExpr)

  for (k = 0; k < 5000; k++) {
    /*
     * 'filtfiltMain:55' xpad =
     * [2*x(1,k)-x((nfact+1):-1:2,k);x(:,k);2*x(nt,k)-x((nt-1):-1:nt-
nfact,k)];
     */
    d0 = 2.0 * x[5000 * k];
    d1 = 2.0 * x[4999 + 5000 * k];
    for (i2 = 0; i2 < 18; i2++) {
      xpad[i2] = d0 - x[(5000 * k - i2) + 18];
    }

    memcpy( & xpad[18], & x[5000 * k], 5000 U * sizeof(double));
    for (i2 = 0; i2 < 18; i2++) {
      xpad[i2 + 5018] = d1 - x[(5000 * k - i2) + 4998];
    }

    /*
     * 'filtfiltMain:57' xpad = filteric1D(bpad,apad,xpad,
     * zi*xpad(1));
     */
    for (i = 0; i < 6; i++) {
      b_zi[i] = zi[i] * xpad[0];
    }

    memcpy( & b_xpad[0], & xpad[0], 5036 U * sizeof(double));
    filteric1D(bpad, apad, b_xpad, b_zi, xpad);

    /*
     * 'filtfiltMain:58' xpad = flipud(filteric1D(bpad,apad,
     * flipud(xpad), zi*xpad(end)));
     */
    memcpy( & dv3[0], & xpad[0], 5036 U * sizeof(double));
    b_flipud(dv3);
    for (i = 0; i < 6; i++) {
      b_zi[i] = zi[i] * xpad[5035];
    }

    memcpy( & dv4[0], & dv3[0], 5036 U * sizeof(double));
    filteric1D(bpad, apad, dv4, b_zi, dv5);
    flipud(dv5, unusedExpr);

    /*
     * 'filtfiltMain:60' y(:,k) = xpad(nfact+(1:nt));
     */
```

```
  }

  /*
   * toc
   */
  tend = time(0);
  lapse = difftime(tend, tstart);
  printf("It took %ld second(s).", (long) lapse);
  /*
   * y-x
   */
}

/*
 * Arguments    : void
 * Return Type  : void
 */
void
filtfiltMain_initialize(void) {
  omp_init_nest_lock( & emlrtNestLockGlobal);
  state_not_empty = false;
  eml_rand_init();
  eml_rand_mcg16807_stateful_init();
  eml_rand_shr3cong_stateful_init();
}

/*
 * Arguments    : void
 * Return Type  : void
 */
void
filtfiltMain_terminate(void) {
  omp_destroy_nest_lock( & emlrtNestLockGlobal);
}

int
main(void) {
  filtfiltMain();
  return 0;
}

   /*
    * File: filtfiltMain.h
    *
    * MATLAB Coder version            : 2.6
    * C/C++ source code generated on  : 27-May-2016 11:30:17
    */

   #ifndef __FILTFILTMAIN_H__
   #define __FILTFILTMAIN_H__

   /* Include files */
   #include <math.h>
   #include <stddef.h>
   #include <stdlib.h>
   #include <string.h>
   #include<stdio.h>
   #include <time.h>
   #include "rtwtypes.h"
```

```
#include "omp.h"
#include "filtfiltMain_types.h"

/* Variable Declarations */
extern omp_nest_lock_t emlrtNestLockGlobal;

/* Function Declarations */
extern void filtfiltMain(void);
extern void filtfiltMain_initialize(void);
extern void filtfiltMain_terminate(void);

#endif

/*
 * File trailer for filtfiltMain.h
 *
 * [EOF]
 */
```

Appendix C: Gem5 ARM O3 deployment diagram



TRITA TRITA-ICT-EX-2016:165