Escola Tècnica Superior
d'Enginyers de Telecomunicació
de Barcelona

PROJECTE FINAL DE CARRERA

qemu-tower: Emulator for multicore hardware/software co-designed processors

GRADE: Enginyeria Electrònica

AUTHOR: Alex Barceló Cuerda
DIRECTOR: Antonio González Colás

Departament d'Arquitectura de Computadors

Barcelona, 2013

a Antonio González, por brindarme esta oportunidad

a Daniel y Fernando, por que ha sido un placer trabajar con ellos

Resum del Projecte

En aquest document es presenta l'entorn de desenvolupament `qemu-tower`, un emulador per a sistemes de múltiples nuclis de tipus Hardware/Software co-design.

El projecte descrit en aquest document és un entorn de desenvolupament ric en funcionalitats per a HW/SW co-designs. El principal objectiu de la configuració són les característiques multicore, que suposen una millora qualitativa respecte el "state-of-the-art" previ.

L'emulador HW/SW co-design té capacitat per a simular tant les característiques *Hardware* del co-disseny com el comportament de la capa de virtualització *Software*. Aquest entorn de desenvolupament té com a objectiu ser una eina per a investigadors en aquest camp i desenvolupadors de HW/SW co-designs. L'eina els permetrà provar, modificar i evaluar co-dissenys, ja sigui de forma global o de qualsevol part que el composin.

Resumen del Proyecto

Este proyecto presenta el entorno de desarrollo `qemu-tower`, un emulador para sistemas multinúcleo de tipo Hardware/Software co-design.

El proyecto descrito en este documento es un entorno de desarrollo rico en funcionalidades para HW/SW co-designs. El principal objectivo de la configuración son las características multinúcleo, que suponen una mejora cualitativa respecto el "state-of-the-art" previo.

El emulador HW/SW co-design tiene capacidad para simular tanto las características *Hardware* del co-diseño como el comportamiento de la capa de virtualización *Software*. Este entorno de desarrollo tiene como objetivo ser una herramienta para investigadores en este campo y para desarrolladores de HW/SW co-designs. La herramienta les permetirá probar, modificar y evaluar co-diseños, ya sea de forma global o de cualquier parte que los compongan.

Abstract

This document presents the `qemu-tower` framework, a Hardware/Software co-design emulator for multicore systems.

The project described in this document is a feature-rich framework for HW/SW co-designs. The main goal of the configuration are multicore capabilities, which results in a qualitative improvement relative to the previous state-of-the-art.

The HW/SW co-design emulator is capable of simulating both the *Hardware* characteristics of the co-design and the *Software Layer* virtualization behavior. This framework aims to be a tool for researchers in this field and developers of HW/SW co-designs. It will allow them to test, modify and profile co-designs, either as a whole or any part of them.

Because the multicore goal, the framework allows a multicore hardware model and also a multicore-aware software layer.

Contents

List of Figures

Chapter 5

NOMENCLATURE

ESL Emulation Software Layer

I/O Input/Output, referring to all read and write transactions of a system, either from/to memory, disk, devices...

ISA Instruction Set Architecture. It includes the specific computer architecture opcodes amongst other hardware dependant aspects

NPTL Native POSIX Thread Library, feature in the Linux kernel with good POSIX Threads support

OS Operating System

POSIX Portable Operating System Interface, set of standards for compatibility between Unix-like operating systems

QEMU Quick EMUlator, a free and open-source software product that performs hardware virtualization

RISC Reduced instruction Set Computing, ISA with a set of simple instructions

TB Translation Block, elemental group of instructions translated and executed as a whole by QEMU

TCG Tiny Code Generator (initially a backend for a C compiler, then simplified to be used in QEMU)

TLB Translation Look-aside Buffer

VLIW Very Long Instruction Word, processor type (ISA)

VM Virtual Machine

CHAPTER

1

INTRODUCTION

In the last years, performance of modern computers has been increasing rapidly. One objective of industry is providing high performance processors while keeping the production cost reasonable.

Nowadays, the cost of improving a single core performance is very high. Current micro-architectures require features with significant increase in power and complexity to get diminishing results on single thread performance. On the other hand, applications tend to become more and more parallel and multithread. Most Operating Systems are also multi-user and/or multi-application.

This has shifted the processor market more and more into multicore. First only for certain high-end environments, like servers for computational intensive tasks. Since then, the number of multicore devices has been increasing and today virtually everything is becoming multicore: mobiles, laptops, tablets, servers, game stations... The software, because it tends to be highly parallel, is notably faster when executed on this processors.

An example to illustrate this improvement: have an OS (e.g. a mobile one) which is, in a certain moment, executing two different applications, each one with two threads. Each thread has 100 instructions. In a monolithic core, the 400 instructions have to be executed one after the other (typically in a pipe-lined fashion). But consider a four core (multicore) system. A multicore system allows multiple instructions (independent and

from different execution threads) to be simultaneously processed by its different cores. This is what we call TLP (Thread Level Parallelism). Consider the above example, where the different applications are independent and the threads are also independent. Then the four-core processor, can execute the four threads in parallel (one core per thread). The instruction parallelism is done at a thread level (hence the name TLP) and most parallelism decisions are in the software, by design.

The improvement of using a multicore approach in a multithread environment is immediate and the complexity increase is not as severe as the counterpart would be: improving a single processor by a 4x factor. Designing a multicore system is not a trivial task, but the cores can be more or less stock designs. Since exploiting TLP on these applications has been shown to be more power and area efficient than continue improving single thread performance, processor designs are trending towards multicore architectures that better exploit thread level parallelism.

However, the single-thread performance is something that has to be taken into account, and may be critical in some scenarios. To improve a single core performance the ILP (Instruction Level Parallelism) has to be improved. Virtualization is an open field which has many applications and, given an adequate implementation, is capable to improve the ILP of a certain system.

1.1 Virtualization solutions

Nowadays virtualization is used in a wide range of applications and environments. We will provide here a brief explanation of the basic concepts and the most common virtualization solutions. The nomenclature used in this section matches the one used in [2]. There are two major types of *Virtual Machines*: 1.1.1 *Process Virtual Machines* and 1.1.2 *System Virtual Machines*.

A general Machine Interface can be seen in figure 1.1. This figure includes two different interfaces and, inherently, two different descriptions of "machine". Those interfaces are two different ways of understanding the inner workings of the system. The second figure refers to a common [20] description from either the hardware or the Operating System point of view. From this Point of View, "machine" is what can be found under the Instruction Set Architecture –typically, it is closely related to the hardware. The first figure refers to a more appropriate description of machine in some situations. The concept of Application Binary Interface (ABI) is the interface applications use to communicate with the machine. The ABI merges both the Operating System calls and also the non-privileged instruction subset of the ISA –*User ISA*. The ABI interface is a common concept in *Process Virtual Machines*, and the ISA is more widely used in *System Virtual Machines*.

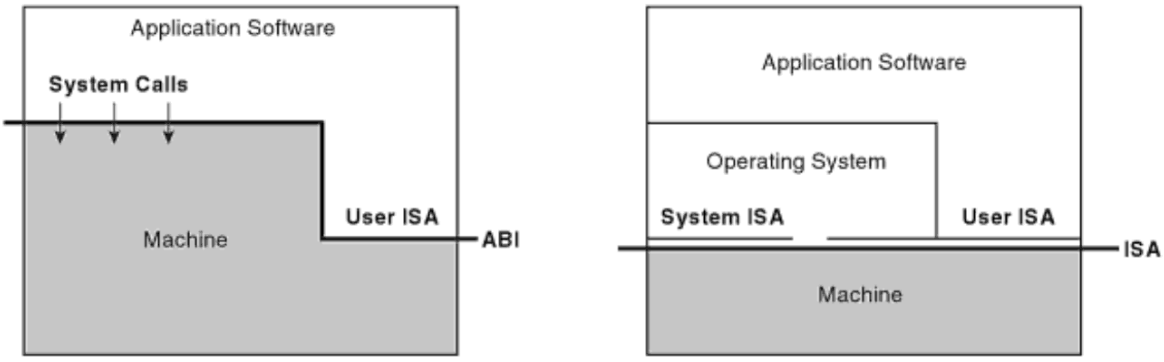


Figure 1.1: Machine Interfaces –Application Binary Interface (ABI) & Instruction Set Architecture (ISA)

The underlying layer capable of executing application software may be a software which is capable of understanding both system calls and the *User ISA* –this is called a Process Virtual Machine. On another scenario, if the underlying software is capable of mimicking a full ISA, then this software is called a System Virtual Machine. In this last case, the software can run an *Operating System* along some *Application Software*. More detailed explanation can be read in subsections 1.1.1 and 1.1.2. The base system is called *host*, Hardware and/or Operating System. The term *guest* is used to refer to the Software being virtualized.

1.1.1 Process Virtual Machines

A *Process Virtual Machine* provides a *virtual* ABI. We use the term *virtual* ABI because this Virtualization Solution provides, from the point of view of the Application Software, a certain ABI. But the key point of virtualization is that the ABI provided is independent from the hardware used or the running Operating System. A simple diagram of this schema can be seen in figure 1.2.

A couple examples of existing Process Virtual Machine follow:

Wine [3] It runs in most POSIX Operating Systems, and it provides an ABI that mimics the one of a Microsoft Windows system. This allows users in a Linux *host* to execute *guest* binaries of Microsoft Windows.

Java VM [4] Sun Microsystems presents a full machine specification for its Virtual Machine. This virtualization solution provides the ability to use platform independent software. This means that a developer can distribute platform independent binaries and any user having the Java VM is able to execute a *guest* application.

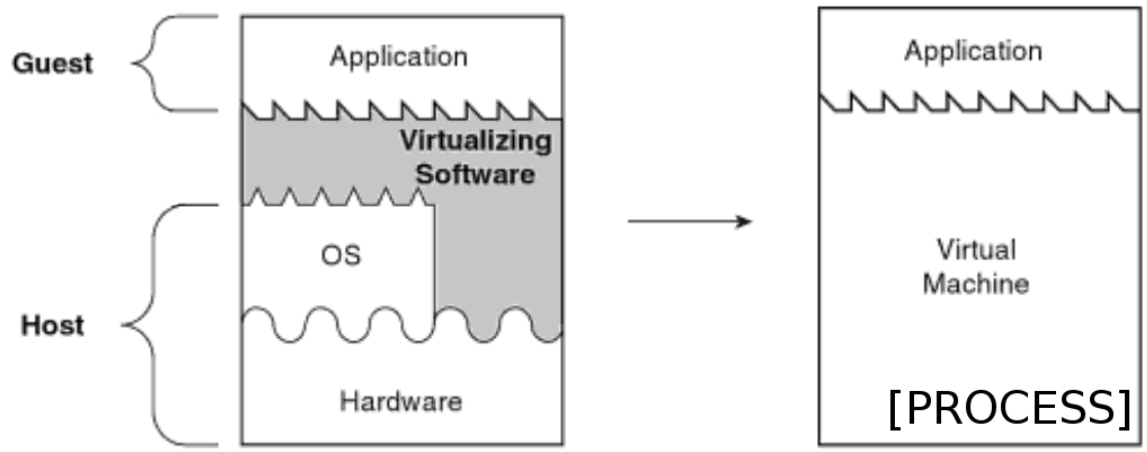


Figure 1.2: Process Virtual Machine

1.1.2 System Virtual Machines

A *System Virtual Machine* provides a full instruction set architecture, both User ISA and System ISA. The idea of a virtualizing software layer is shared with *Process Virtual Machines*, but as we can see in figure 1.3, the interface presented by the Virtualizing Software is able to support a *guest* Operating System. This guest OS can run applications, just as if the OS was running over physical hardware.

The Virtualizing Software can be a user-space application, running as would run any regular application, like VirtualBox [5]. Or it can also be closer to the hardware,

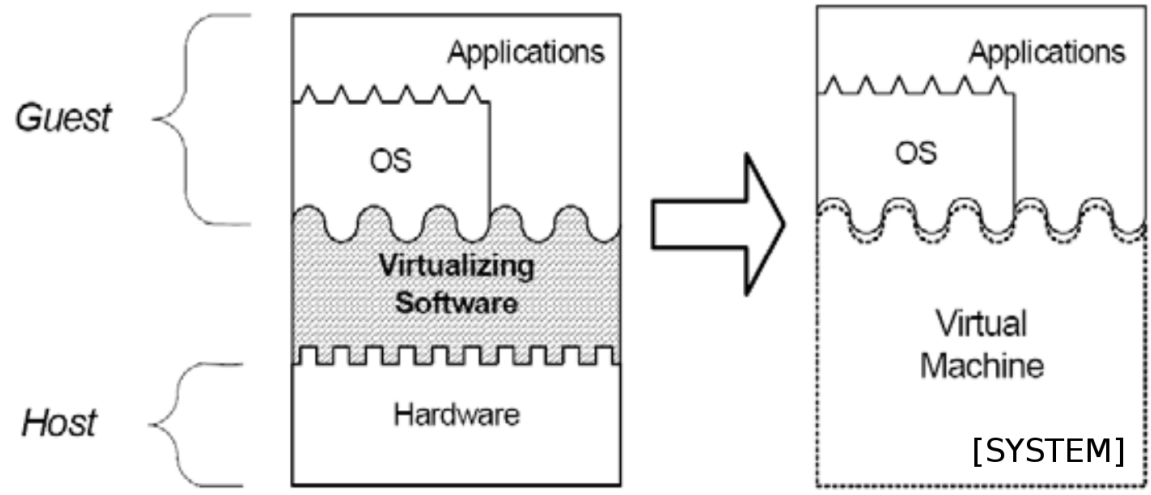


Figure 1.3: System Virtual Machine

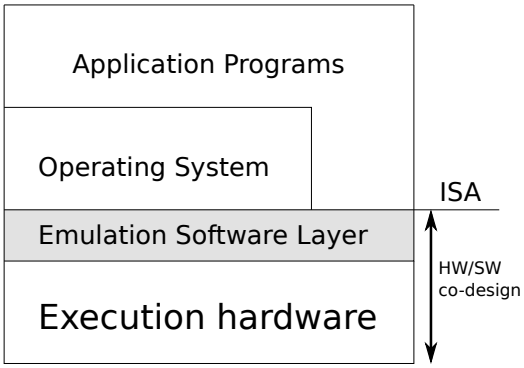


Figure 1.4: Hardware/Software co-design System Architecture

integrated in the kernel, like Xen [6] or KVM [7].

There is a scenario where the "Virtualizing Software" is a software layer highly coupled with the hardware features. The system in this case is called a Hardware/Software co-design processor. The system architecture can be seen in figure 1.4. This system is a subtype of System VM characteristics. But in HW/SW co-design the hardware is designed specifically for a certain characteristics and the software is developed accordingly. Typically, there would be specific hardware for certain applications or performance constraints and an Emulation Software Layer capable of binary translation and dynamic optimization. Hence, the "Host" is not general purpose standard hardware but a special design. The ISA is part of the design decisions of the HW/SW co-design decisions, as it depends on the binary translation of the ESL –it does not depend entirely on the physical hardware, as happens in traditional hardware systems. This design allows a high performance development that can satisfy special needs and demanding design constraints.

1.2 Hardware/Software co-designed machines

Hardware/Software co-designed machines are a subtype of System Virtual Machines where a software layer –called the *Emulation Software Layer* or ESL– is implemented tightly coupled with the architecture underneath. In this model some of the hardware complexity is moved to the ESL, resulting in a simpler hardware. This simpler hardware requires lower power and area than conventional big cores, but its performance is also limited. It is ESL responsibility to modify the code accordingly with the aim to improve the performance.

In any real physical design, the available area is always limited and there are physical integration constraints. Several implementations of Hardware/Software co-design

cope with those constraints. Those alternatives micro-architecture, combined with a multicore design, can lead to high performance systems while keeping low the single-thread performance loss.

Some aspects of dynamic binary optimization are thoroughly analysed in DAISY [8]. As can be read from IBM [9]:

> DAISY (Dynamically Architected Instruction Set from Yorktown) is an offshoot of this work, and aims to make VLIW[1] and other novel ILP architectures 100% compatible with popular existing architectures such as PowerPC, x86, and S/390, as well as the Java Virtual Machine.

Dynamic binary optimization is a key aspect of the Emulation Software Layer. A high Instruction Level Parallelism can be ensured in good co-designs.

More development efforts from IBM itself is BOA (Binary-translation Optimized Architecture) [10]. It establishes a design of an architecture with a high frequency processor along with a binary translator. The high frequency processor can be achieved thanks to a simple core design, which leads to a very high speed hardware design. The simplification of the core implies a performance loss, but these designs are expected to provide binary translation software. The benefits of the binary translation overcome the performance loss due to hardware.

Transmeta company designed and commercialized two HW/SW co-design systems. The first generation was named Crusoe and the 2nd generation Efficeon. The co-designs microprocessors of those Transmeta systems were VLIW-based. The Emulation Software Layer exposed a standard x86 Instruction Set Architecture. This x86 compatibility allowed the user to run unmodified standard x86 Operating Systems. The chosen VLIW hardware kept the power consumption of the processor very low, while Transmeta showed off high ILP and good performance. The software layer of the co-design was called Code Morphing software (CMS) and was responsible for the dynamic translation.

In this project, we will focus on a simulator for a new generation of Hardware/Software co-designed machines. The research area of Hardware/Software is, like many others, an open field that needs the existence of simulation environments and similar research tools. The introduced "qemu-tower" simulator is aided to help the development of HW/SW co-designs, both the Hardware characteristics and the Emulation Software Layer code. In addition, the simulator allows multicore co-designs simulation. The whole simulator framework will simulate the multicore Hardware layer and the Software Layer as a whole co-design system.

[1] Very Long Instruction Word, type of architecture which uses long instruction opcodes which bundle several number of operations. The hardware multiplicity of execution units is related to the size and characteristics of these micro-architecture opcodes.

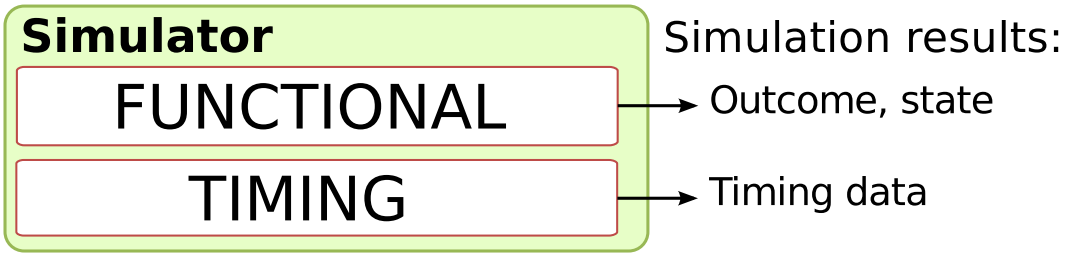


Figure 1.5: General aspects of a simulator

1.3 Simulation environments

In many engineering design stages, good and reliable simulators are greatly helpful. Those tools provide quick and low cost ways to check, design, test, and predict the reaction and characteristics of a system. To be able to provide those *simulation results*, some information about the system to simulate is required. Just like knowing the gravity acceleration is needed in order to calculate and simulate the falling of a ball, the characteristics of the micro-architecture are crucial for appropriate simulation results.

First of all, we can divide a simulator structure into two basic aspects (depicted also in figure 1.5):

Functional The simulation must yield a certain outcome. The result of a simulation comes from the functional aspect of it. A *functional simulator* highlights the resulting state of the simulation on top of other aspects.

Timing Most simulation results here is an interest of the timing aspect of the simulation, which has to be managed by the simulator framework.

The term emulator is normally used in computing referring to software capable to replicate the behavior of a certain system (guest) in top of another (host). In some aspects, we can consider most emulators to be simulators for computer execution. And those emulators are, at the same time, run in a computer. The objective of running an emulation is, in most cases, obtaining the outcome of the simulated program. To name and describe a common widely used emulator:

MAME [11] A gaming emulator which provides the chance to users to play games for nowadays obsolete hardware (arcade machines).

Up to this point, we have discussed some functional characteristics of simulation systems. But the timing aspect is also important in a simulator. We will present two

major scenarios of the *timing aspect* of computer science emulations: **Timing-directed** and **trace-driven**.

When the time flow is bundled in a simulation and its flow affects the functional aspect, the simulation is normally timing-directed. Network simulation is typically implemented in this fashion, because the speed of its links and the processing time of the packets can have a deep impact on the network health, and also on simulation aspects like routing decisions. Micro-arhictecture tend to be implemented in timing-directed schemas, as the time flow can affect the executed trace. E.g. a synchronization stage between threads, where the executed instructions by the waiting loop can vary depending on the time waited.

The other possible timing emulation presented is the *trace-driven*. This scenario is used when the time flow is not relevant for the functional. This is applicable in scenarios where the execution of the simulation is not affected by its timing. A computer science example: the time flow in the simulation of a monolithic core doing a Sieve of Eratosthenes is irrelevant, because the speed of the core does not impact the algorithm itself. In those scenarios, the timing data can be calculated by the time of each operation. In cases similar to this example, a trace-driven simulation is adequate. This simulation type uses the "dead" execution trace –once it has already been executed. With this already-executed trace, all the timing results are obtained.

1.3.1 An example

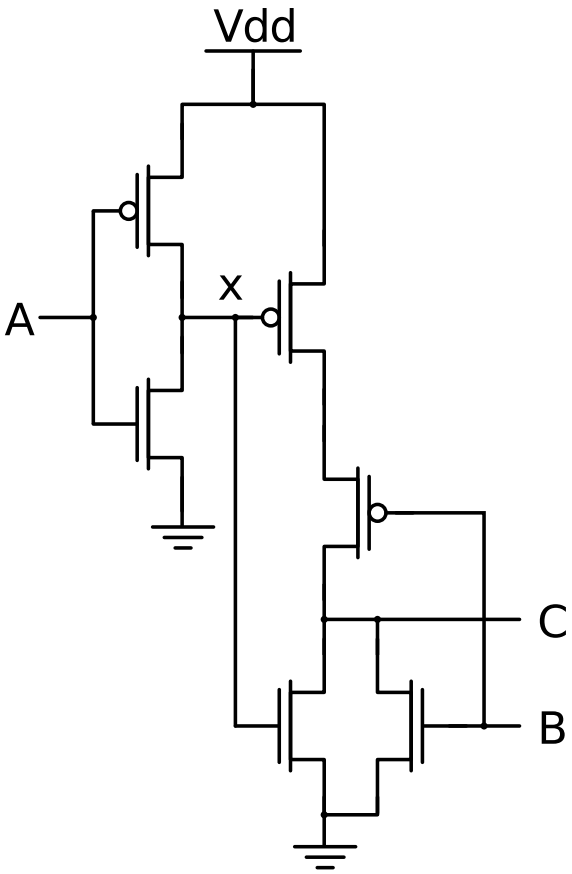


Figure 1.6: Simple electronic circuit

The characteristics of a timing directed are greatly relevant to this project, as will be seen in next chapters. The term functional simulation will also be used abundantly.

To better illustrate those two terms, a small example will be analysed here. An electronic example is chosen because of its simplicity, but many other examples (from a lot of fields) might have been used here. In figure 1.6 the basic circuit is shown.

The truth table is the following:

A	B	C
0	0	0
0	1	0
1	0	1
1	1	0

The logic function behind this circuit is the following one:

$$C = A \cdot \overline{B} \tag{1.1}$$

In a typical functional simulation, the process would use the equation 1.1. Some further data, like the propagation time, may also be used for a functional simulation, but the block (the circuit used in this example) is not exploded into its internal electronic elements and aspects.

Given an input data vector for A and B, the functional simulation would run as follows:

A	B	C
0	0	0
1	0	1
1	1	0
0	0	0

The simulation is simple, but may be incomplete if a feedback loop is included. Or may be unusable if switching times are critical between inputs and output.

Consider now a basic timing-directed simulation. We will consider everything ideal except a propagation time of transistors of 1 time unit (t.u.). The input data vector is identical, with transitions every $10\,t.u.$

t.u.	event	A	B	x	C
0		0	0	1	0
	...				
10	A=1	**1**	0	1	0
11	propagation	1	0	**0**	0
12	propagation	1	0	0	**1**
13	established	1	0	0	1
	...				
20	B=1	1	**1**	0	1
21	propagation	1	1	0	**0**
22	established	1	1	0	0
	...				
30	A=0; B=0	**0**	**0**	0	0
31	propagation	0	0	**1**	**1**
32	propagation	0	0	1	**0**
33	established	0	0	1	0

We can see that the timing-directed simulation shows a glitch at 31 timing units, after the transition from input $(A, B) = (1, 1)$ to input $(A, B) = (0, 0)$. Because not all the signals change simultaneously, this spurious is generated before internal nodes are established, specially due to the **x** node.

In this example, a glitch has been detected thanks to the timing-directed simulation. The presence of some feedback loop, or interest for the time flow data, may enforce the use of a timing-directed simulation strategy. We have to keep in mind that using this kind of simulation adds some complexity to the process.

Generally, a functional simulation provides basically the result of a simulation. In micro-architecture, that means that the machine state of the execution is obtained. A timing-directed simulation is more complex, but the result includes the accurate timing data. It can also expose some details that are kept hidden or simplified in a functional simulation.

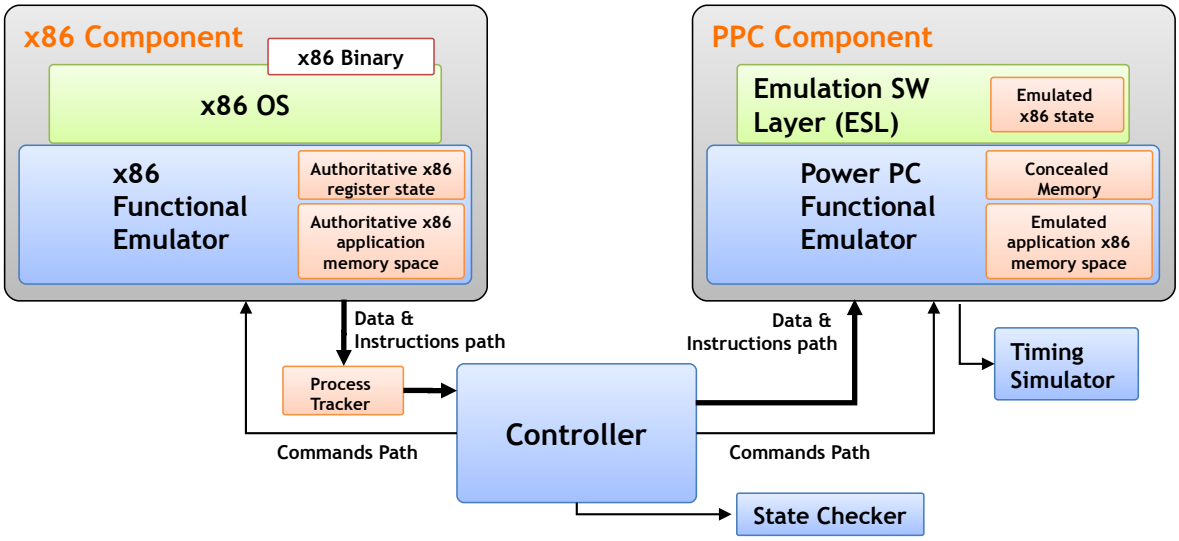


Figure 1.7: DARCO structure (from [1])

1.3.2 Simulation process in HW/SW co-design field

Unfortunately, research on the area of hardware/software co-design is sometimes cumbersome due to the lack of simulators available. DARCO project [1] is a relatively recent activity that is building a simulator suitable for research on this area. This is the state-of-the-art of simulators on the Hardware/Software co-design field.

The objective of the DARCO simulator is to provide an accurate realistic response of the system. The basic emulation blocks are functional emulators because there is no need for simulation data of internal parts of the emulated layers.

1.3.3 DARCO project

The internal structure of DARCO can be seen on figure 1.7. DARCO is a software tool to simulate a Hardware/Software co-design. A x86 compatible system running an Operating System is simulated. This Operating System, including its applications, can be called the "Guest". The HW/SW system simulated in the DARCO project includes an Emulation Software Layer and a PowerPC-based hardware.

The existence of the ESL block is primary in any HW/SW co-design, as it is one of the main points of the hybrid design of those systems. The use of PowerPC-based hardware, instead of any other micro-architecture, is a discussed design decision [1].

1.3.3.1 Emulation and device management

The DARCO *x86 Functional Emulator* is responsible for running the OS which executes the applications under simulation. DARCO is ready to run an unmodified x86 OS through this block, which also does the device emulation and memory management.

From a technical simulation point of view, this block can run autonomously. This can be interesting for debugging and, because the system is simpler and less blocks are involved, can realize an authoritative functional simulation. An authoritative simulation is interesting to double-check the correctness of a non-authoritative simulation –in our case, the result of a HW/SW co-design simulation.

From the HW/SW co-design development environment, there is still no hardware involved. This means that, at this stage, there is not yet any relevant data of HW/SW simulation.

1.3.3.2 Simulation data

The executed instructions of the *x86 Functional Emulator* are passed to a controller which forwards them to the ESL. This ESL performs the dynamic translation of instruction, issuing PowerPC opcodes. The PPC opcodes would be the ones executed by the Hardware layer, in a real hardware design. In the DARCO structure a *Power PC Functional Emulator* replaces the real hardware.

The *Timing Simulator* receives the instruction stream and provides the detailed execution times for the given x86 binary run in the x86 OS. This data is what we want to retrieve from the DARCO simulation.

1.3.3.3 Multicore pitfall

The way the different blocks are engaged prevents the x86 emulation to be affected by the *PPC Component* or the *Timing Simulator*. The layout of the DARCO emulator is an example of a *trace-driven simulation*, as has already been presented in 1.3.

As opposed as what would happen in a *timing-directed simulation*, the DARCO set up does not include any feedback from the timing module into the execution flow control. The trace, as the flow suggests, is analysed once it has already been processed.

Trace-driven simulations are straightforward and DARCO achieves great results. The use of a dead-trace allows a simple schema which includes an authoritative simulation. DARCO project is feature rich and is very robust in its typical applications.

Unfortunately, the trace-driven schema used in DARCO project is not suited for multiprocessor simulation. Unless certain demanding conditions can be satisfied [12], trace-driven simulations are not well-suited for multicore/multiprocess simulation. Generally, the behaviour of the different cores traces depends on all the previous execution. Because the timing simulator and the ESL are entangled with the trace, the controller cannot know beforehand which trace and which jumps the code will follow, as one instruction of one core may affect the trace of another core –e.g. in synchronization stashes of multithreaded applications.

In this project we address the need of a simulation environment for multicore Hardware/Software co-designs. We have remarked in this section that the state-of-the-art tool is not suited for multicore. Not only it is not suited for multicore, but it cannot be modified to embrace multicore hardware.

1.4 Project objectives

Hardware/Software co-design is an open field which can provide new generations of higher performance processors with reduced power consumption while keeping a relatively low complexity.

The goal of this project is providing a Simulation Environment for the HW/SW co-design research field. This tool will be capable to simulate both, and the combination of, Hardware and Software layers. The focus is a proof-of-concept functional emulator framework capable to deal with the Emulation Software Layer translation and the multicore hardware execution. To allow the multicore constraints this framework must allow a timing directed schema and multiple core control.

Tools with the aforementioned features are not currently available. We have discussed that DARCO, as the state-of-the-art, lacks the key *multicore* capability. Moreover, we have seen that it is not possible to extend DARCO simulator to cover multicore designs. For this reason we present a new framework environment which allows research on this field and also aids the development of new Hardware/Software co-designs.

1.5 Structure of this document

This document is organized as follows:

This first chapter has covered the introduction of concepts and motivations. The objectives of the project are presented here.

In Chapter 2 the architecture basics of this framework are outlined. The logic blocks with which the framework is built are designated and their relationships explained.

Chapter 3 discusses the design decisions of this project. The motivation behind the choices are justified in this chapter, along some basic technical description of them.

The project stages are detailed in Chapter 4. First some pre-development tasks are described and then detailed explanation follows on the development stages of the project.

Results of the project can be found in Chapter 5. The chapter deals with a couple aspects of emulation tests. Those examples include high-level sequence diagrams and additional technical debugging output.

Chapter 6 contains the conclusions of this project.

Appendix A contains a list of public information regarding improvements to the backend emulator tool of the project. We have developed those patches for this project and then published and reverted to the community.

CHAPTER

2

OVERVIEW OF THE PROJECT

2.1 Basic layout

The diagram of the emulation architecture of the framework this project develops is presented in figure 2.1.

We will differentiate the following basic blocks already represented in the figure:

1. Operating System
2. System Emulation
3. Hardware Emulation
4. Timing simulator and controller

2.2 Building blocks

First of all, we will briefly overview the basic blocks which compose this project. There are 4 basic blocks, which can be seen in figure 2.1. To describe better the *System Emulation* block, its basic sub-blocks are presented: *Main control* and one *CPU Emulation* per-CPU.

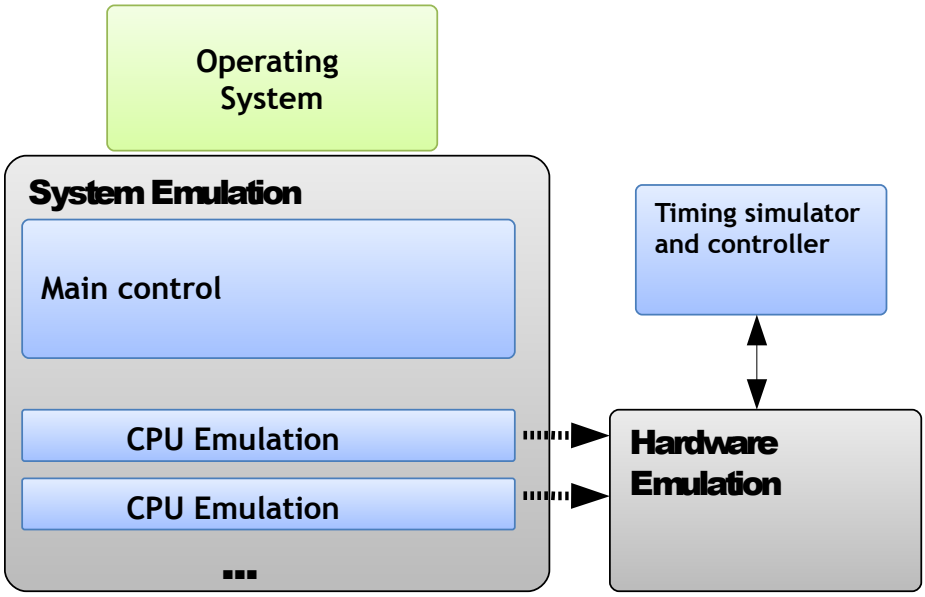


Figure 2.1: Layout of the HW/SW co-design simulator

2.2.1 Operating System

The Operating System is the simulated environment. We want to achieve emulation with an unmodified operating system. That means that any stock OS compatible with the chosen ISA will work inside the emulator. The Operating System being virtualized is generally referred as "guest OS".

The Instruction Set Architecture of the Guest Operating System must match the Hardware/Software co-design. The use of a really standard ISA is a standard choice, as this allows the system to be compatible with existing software.

Because a complete OS is being simulated, there is the need of device management. To work correctly, an OS expects a certain list of peripherals: screen, keyboard, disks and storage, clock, memory... Some of those are mandatory, and some of those aren't. As this guest is being run inside a simulator, those devices will not be physical devices. Instead, the simulation will provide virtual devices which will mimic the real behaviour of those.

It is worth noting that thorough device emulation is not part of a real life HW/SW co-design system, but a feature needed in a full system emulation framework. A HW/SW co-design processor does not use virtual devices, because peripherals are real hardware of the system. The simulation framework does not use real hardware peripherals, so device emulation is needed to achieve the full system emulation.

2.2.2 System Emulation

As we have stated, there is the need of some peripherals virtualization along some general control of the simulation. The *Main control* sub-block is responsible of those features of the simulation.

The Emulation System Layer works at the *System Emulation* level. In figure 2.1 the *CPU Emulation* sub-blocks are shown, which correspond to the Emulation Software Layer result. Those sub-blocks must be supported by some kind of hardware –either phisical or virtualized.

2.2.3 Hardware Emulation

Because the project aims to be a full-fledged Hardware/Software co-design simulation framework, there is the need of including virtualization for its hardware layer.

The hardware Instruction Set Architecture is decoupled from the guest requirements because of the Emulation Software Layer. That allows the designer to choose any hardware ISA. One wise choice to address the performance issues presented in the first chapter is the use of simple hardware. Using a moderate-to-low complexity core can improve its power performance or reduce its integration area. Those are features pursued in the computer architecture field.

The use of virtualization, in the most general case, impacts performance. In addition, the use of simple cores in lowers the expected Instruction Level Parallelism of a system. Fortunately, the presence of a custom Emulation Software Layer increases the expected Instruction Level Parallelism and overcomes the performance hit. In addition, the potential Thread Level Parallelism is increased, as the cores used are less power-hungry and more easily integrated.

2.2.4 Timing simulator and controller

In some cases, the different CPUs may be entangled by atomic instructions, locks and memory accesses. The time spent in each instruction by each CPU is not a trivial matter. In those cases, there is the need of some type of controller. Ideally, the project would follow a realistic model of a certain processor. This is the objective of the timing-directed simulation and requires the presence of this timing simulator responsible of tracking all the time flow.

A realistic and feature-full timing simulator is not the focus of the current project, but enabling the functionality in the overall design is.

2.3 Execution flow

The blocks of the framework have been presented, but there is yet no explanation of the simulation work-how.

To correctly simulate the guest Operating System is the focus of a simulation framework. The multi-layered nature of the framework makes things complex. To better understand the global characteristics of the framework two different point of views will be commented here:

Emulation Software Layer Which is the junction between the guest Operating System and the Hardware. It exhibits a certain ISA and is designed to be executed in top of a certain Hardware.

Hardware This is, in a real physical Hardware/Software co-design, the supporting hardware of all the software. In a simulation framework, we do have a functional simulation of the hardware (see 2.2.3).

2.3.1 ESL Point of View

One of the main purposes of the ESL is dynamic translation of instructions. One important and basic task of this layer in a Hardware/Software co-design is generating Translation Blocks (TB). A Translation Block is a group of that have already been translated into host architecture. Additionally, these blocks can be optimized.

A more simple approach consist in a per-instruction interpretation, resulting in a straightforward emulation. But this kind of emulation yields bad performance results because of the overload. In the Hardware/Software co-design field, binary optimization is a performance requirement. The Emulation Software Layer is responsible of the dynamic optimization of Translation Blocks.

Once a TB is generated, it is saved. To use a TB, a jump is issued and the code of the TB is executed. The TB exits, generally, by jumping back into management code. The ESL is responsible of tracking all the TB and is generally able to reuse them and optimize the most used ones, amongst other performance optimization features.

This project, by providing a simulation framework, provide a way to study the ESL behaviour under arbitrary application software, thus simplifying the research and development of new and improved Emulation Software Layers.

2.3.2 Hardware Point of View

To the hardware entity, everything software described previously become, basically, instructions to be executed. The instructions includes both the ESL management routines and the Translation Blocks.

The hardware is executing management instructions of the ESL. This is, from the hardware execution point of view, read and write memory access: instructions are read from the guests OS and running application, and when necessary, TBs are built into available memory. When one TB is ready and should be executed, the hardware finds a jump into a different memory area which contains a TB.

Because of the presence of a timing simulator –already presented in the previous section– the hardware instruction execution speed is governed by the simulation time.

2.4 Combining the blocks

In a real Hardware/Software co-design the combination of blocks would be ensuring proper compatibility between the Emulation Software Layer and the Hardware. The Instruction Set Architecture must match, and any custom feature used by the ESL must have its counterpart in the Hardware. This is one of the strengths of HW/SW co-design and can lead to very specific designs, and high adaptability systems can be achieved.

The framework emulation blocks must also match. Instead of having a combination of Hardware and Software, the framework has a combination of Hardware emulation and Software emulation. Same ISA and feature constraints apply. There is an extra device emulation block that must fit the two emulation blocks. Because of this aims to be a simulating framework and not a mere functional emulator, there is an additional timing simulation aspect (see 2.2.4) which needs arrangements between both the *hardware emulation* and the *timing controller* blocks. The presented combination of blocks allows the desired timing-directed simulation of the framework.

CHAPTER

3

DESIGN DECISIONS

Once a basic diagram is drawn, technical aspects and practical requirements have to be analysed. In this section we will present and justify most decision made during the project planning and some decisions regarding development.

In figure 3.1 the layout of the qemu-tower simulator can be observed. The basic layout was presented in last chapter figure 2.1, and now the diagram includes some of the design decisions that will be discussed in following sections.

3.1 Hardware

The chosen hardware model for this simulation framework is `Power PC` family. It is a fairly common hardware, and its characteristics can be appropriate in a Hardware/Software co-design. This design decision is common to the DARCO project [1], and some advantages of this decision are discussed in further detail.

3.2 Exposed ISA

The exposed Instruction Set Architecture is i386 mainly because it is nowadays the most used ISA. Using this instruction set allows the designs to run any major

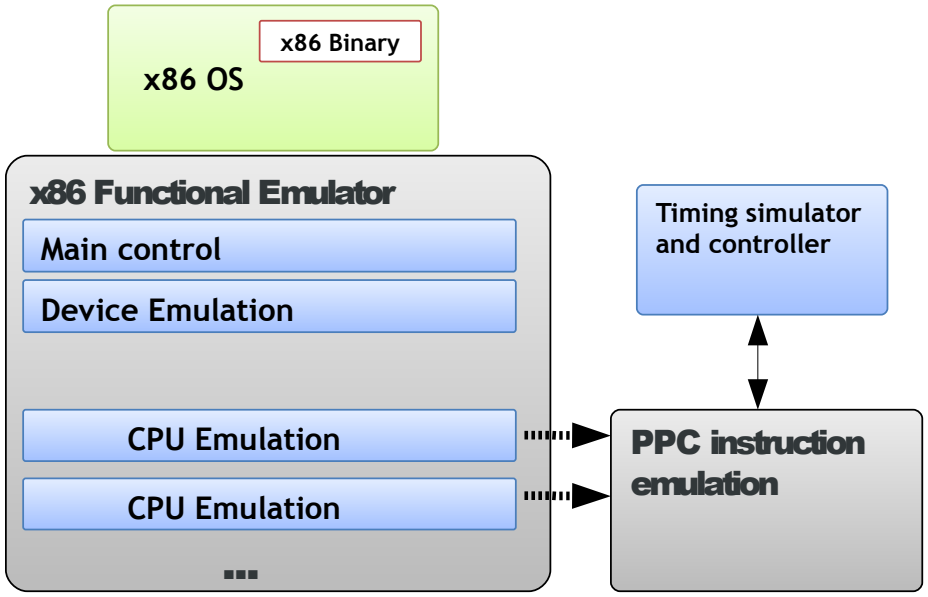


Figure 3.1: Layout of the qemu-tower simulator, including some design decisions

Operating System as the guest OS of the co-design.

The i386 ISA ensures compatibility with almost all existent modern software. This decision ensures that the resulting co-designs will have good compatibility and low migration costs. Choosing this architecture is a common design decision on virtualization scenarios, as the i386 is the predominant micro-architecture nowadays.

3.3 Guest Operating System

The simulation framework aims to be compatible with any major Operating System, and the Hardware/Software co-designs are aimed to be transparent to any matching-ISA OS.

To achieve a first-version working simulator a reduced Linux-based Operating System is chosen. Using a simple open source OS helps debugging and it fixes the minimal features of the simulation framework required for an Operating System.

3.4 Emulation Software Layer

Any Hardware/Software co-design needs an Emulation Software Layer. This project may be used to create new ESL or modify and improve existing ones. But there is the

need of some starting default ESL. An interesting and appropriate ESL may be the one developed by DARCO project [1] or the ones presented on DAISY [8] and BOA [10].

However, none of them are aimed at multicore systems, and they are not easily assembled into the presented framework. For those reasons, a generic emulation backend is chosen, to provide some basic ESL features along other needed emulation requirements of the system.

3.5 Emulation backend

The framework being presented in this project is heavily dependant on the emulation. The Emulation Software Layer is itself an instruction emulator. But there is the need of other aspects of emulation.

The execution hardware simulation should be done with functional instruction emulation. The peripheral management can be done by a full-system emulator tool. Any available System Virtual Machine software, like Bochs [13] or Xen [6], has peripheral management implemented.

There are software tools which can provide, separately, the features needed in this project. Unfortunately, those software pieces tend to need heavy modification to be used in a Hardware/Software co-design project like this one. Generally, they are not modular and are not easily assembled together. Those are the reasons why QEMU, a versatile emulator, catches our attention.

3.5.1 The QEMU project

The QEMU project [14] is a open source project which provides system emulation for x86, ARM, MIPS, PowerPC, SPARC, amongst others. It is also capable of application emulation for Linux platforms on the aforementioned architectures.

QEMU project provides a virtual device control, allowing it to emulate an Operating System in a System Virtual Machine mode. This can be called *system-mode* or also `qemu-system`, as this is the command name for this mode. This can be used to provide device emulation into the framework.

QEMU provides a different mode where it carries out application binary translation. This application emulation is an ABI level virtualization where a guest application from an arbitrary architecture is executed in top of a host Operating System. This can be called *user-mode* or also `qemu-user`, being this last term the command name for this mode.

We evaluate that QEMU is widely used, adaptable and also a constantly improving tool. Because those characteristics and its features, we decide to use it as the main functional emulator backend of our project. This allows us to benefit from its community and updates, while it covers most emulation requirement that this simulation framework requires. It has enough documentation and a big and reachable community of developers and volunteers.

3.5.1.1 Using QEMU

At first, it would seem that QEMU [14] offers everything needed in the presented framework. However, we have to take into account that QEMU is a work-in-progress open source project. Its list of achievements are huge, but not as vast as its goals. In addition, the micro architecture field is an ever changing field, and QEMU is always improving with upcoming processors and features.

QEMU is quite stable during its typical virtualization scenario, but this projects presents a custom configuration of emulation blocks. This became cumbersome in the development stages, as was proved to not be a straightforward configuration. QEMU is a big complex piece of software, and understanding its inner workings was key to the success of this project.

Understanding QEMU is not a trivial task. Many hours have been invested in this task, and many more hours have been needed to mend all the problems that arose during its development stages. The configuration and interaction between QEMU blocks is further explained in 3.6. There is also relevant and related information in the debugging stage 4.3.

3.5.1.2 Modifying QEMU

QEMU is a project that receives updates in a daily basis. Its code is complex, publicly available in a distributed revision control at `git://git.qemu-project.org/qemu.git`. There is also an official developer mailing list [15].

During this project there have been some bug corrections in addition to some missing features which were required and thus, were added into QEMU code. Those modifications to QEMU have already been proposed to the community. Some links and further technical description of the most important ones are presented in appendix A. Behind each of those modifications there is a thorough task of bug-hunting and inner-workings understanding. And, of course, we must ensure that the modifications work satisfactorily under typical scenarios.

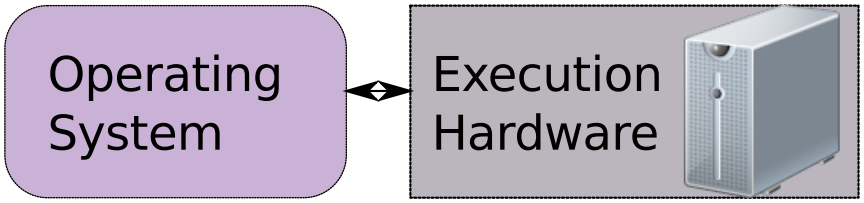


Figure 3.2: Simple configuration –no virtualization

Those contributions are needed for this project. But, in addition of the immediate use in this project, most changes have been revised and accepted by the developer community. That means that this project development has, collaterally, improved QEMU and now a huge community can benefit from it, both final users and developers. Any researcher using or modifying QEMU benefits too from those improvements. This is also a great boost for future work in this same project.

3.6 Emulation chain configuration

We have presented the different aspects of emulation, but we have to outline the emulation path of executed instructions. As we have emulation blocks for both the Emulation Software Layer and the Hardware layer, the qemu-tower presents an emulation chaining of instructions. The proposed configuration of QEMU blocks is the emulation chaining between QEMU instances. This lets us use a standard QEMU version and easy deployment of the framework, but presents some drawbacks that will be addressed.

To illustrate the emulation chaining configuration of the framework, a couple of typical scenarios will be briefly introduced prior to the final qemu-tower emulation stage configuration.

First of all, let's present a simple machine configuration without any kind of virtualization: figure 3.2. In this scenario the hardware proceeds to fetch instructions, belonging to the operating system. The operating system is executed directly by the execution hardware.

In figure 3.3 we can see a configuration with one software emulation stage. In the HW/SW co-design scenario, the software emulation stage would be the ESL. In a typical virtual machine environment, the generic *emulation software* may be an operating system with some kind of virtualization software emulating a guest system. In both the virtualization and the HW/SW co-design scenarios, the hardware is fetching instructions of a middle software layer. Moreover, this software layer is responsible of

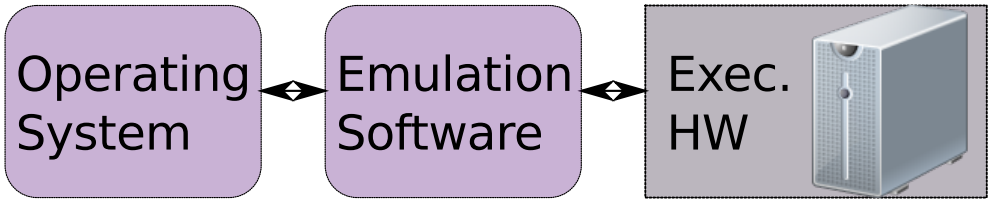


Figure 3.3: One stage emulation –typical virtualization

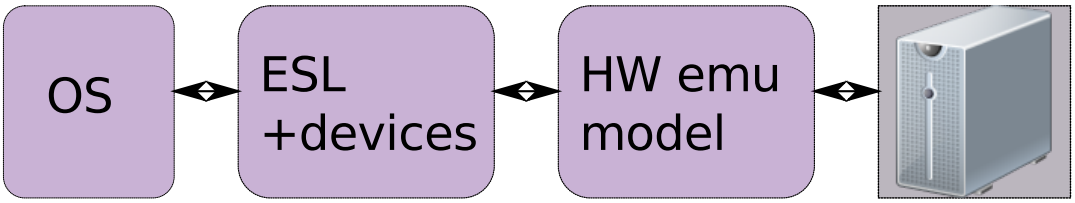


Figure 3.4: Two stage emulation –qemu-tower framework

conditioning the operating system and also providing instructions understandable by the hardware.

At last, the emulation chaining schema of the qemu-tower project is presented in figure 3.4. The execution hardware in this scenario is not relevant, is simply a physical platform where the simulation can be run and results obtained. The figure presents two stages –the emulation of instructions is done at the ESL level and then also at the *hardware emulation model* level.

For the *hardware emulation model* we use an application level QEMU command with Power PC as guest ISA. The device emulation is done by a system mode QEMU, which also is responsible of the ESL emulation. The QEMU instruction translation engine can be considered an ESL, because it is translating and behaving as such. The QEMU system bundles the ESL behavior and the device emulation. We do not consider QEMU a realistic HW/SW co-design ESL, because the requirements and design objectives of a Software Layer in a co-design are wildly different. Once a full framework is presented, a real ESL can be hooked into the framework and worked with.

Now that the two step emulation present on this project is presented, we will divide the instructions in two different types:

Hardware execution For instructions originated by the OS. This instructions are the ones managed by the ESL.

Accessory instructions For instructions originated in qemu-system, mainly related with the device emulation. Those instructions are not useful for simulation purposes.

3.6.1 Hardware execution

Most instructions coming from the Operating System needs to be executed by some hardware. In a real-life system, those instructions would be managed by the ESL, and the hardware would execute the ESL –including the translated instructions.

In the emulation framework, the first emulation step "produces" instructions, which are emulated by the hardware emulator. Those instructions would be run on top of real hardware, but we need this layer in the framework for simulation purposes. It is understandable that a framework with the objectives presented in this project needs this dual-stage emulation for those instructions.

3.6.2 Accessory instructions

There are some routines and code in the *qemu-system* which wouldn't be issued in a physical real environment. An example of those routines are the device emulation routines needed, or internal management for the Virtual CPUs control. In the proposed configuration, those instructions are also emulated through the PPC emulator.

The accessory instructions, e.g. the ones responsible for hardware model emulation, do not intrinsically need to go through a second emulation layer. In a real-life HW/SW co-design those instructions would either not exist or not be managed by the ESL. The memory access and device management would be done directly. In the presented framework architecture, all instructions go through both stages of emulation. Letting those accessory instructions run across both layers has some drawbacks, in terms of efficiency and robustness.

Despite the inconvenience, this set up requires low level of QEMU modifications. Building the framework in this fashion ensures a relatively-low complexity configuration and allows the system to use off-the-shelf components. Doing the required emulation blocks from scratch would require a massive quantity of man-hours, while using off-the-shelf components leads to useful results in timely fashion.

Once all the design decisions are discussed, the next step is the development of the framework.

CHAPTER

4

PROJECT STAGES

As the state-of-the-art available infrastructure could not fulfil the desired features for the simulation framework, the need to do a full design arose. The objective, a feature rich framework for Hardware/Software co-design multicore simulation, is an ambitious goal. The design decisions have already been discussed in previous chapter. An approximate timeline can be seen in figure 4.1. A brief discussion of this timeline follows.

Planification and tool analysis [4 months] The state-of-the-art in Hardware/Software co-design simulation frameworks is DARCO. At first, *DARCO testing* lead to an understanding of this tool, which was a good start point. However, we soon discovered it was not a good foundation given the proposed goals.

Knowing that a new framework had to be built, a *design decisions* stage started. This task overlaps with the *tool familiarization*. At the end of those tasks, a basic layout of the framework has been decided and a basic know-how has been acquired. This includes a lot of documentation reading and testing, prior to the development start.

Development stages [9 months] Those tasks are the main topic of this chapter. In the following section, we will briefly present each of those stage and later the

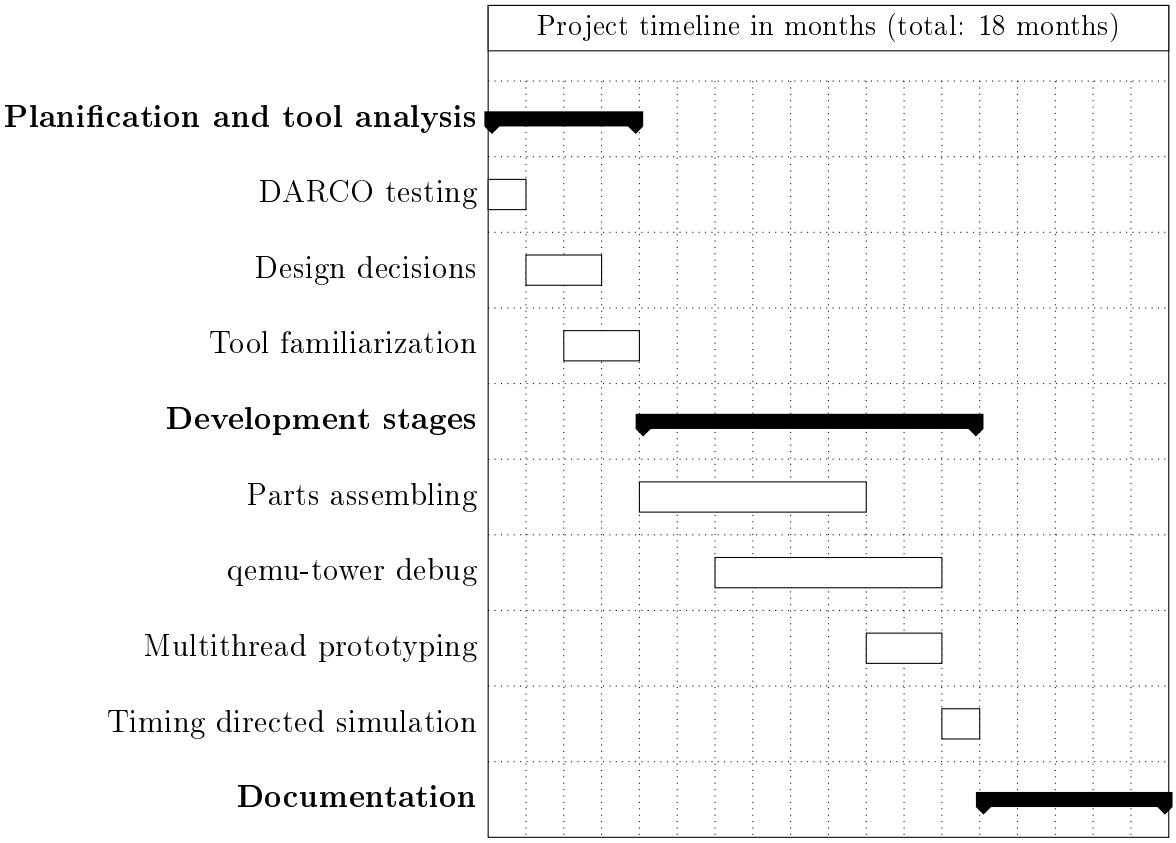


Figure 4.1: Gantt Chart of this project

explanation will be expanded.

It is worth noting that *qemu-tower debug* is a complex task. Several months are dedicated to this process, given the complexity of the presented framework and the ambitious set of goals. Correcting all the issues that the framework configuration arose is a major time-consuming task of the development.

Documentation [5 months] Once the framework is completed, there is the need of documentation closure of the task done, the modifications made and the results obtained. This same document and the presentation are some of the results of this stage.

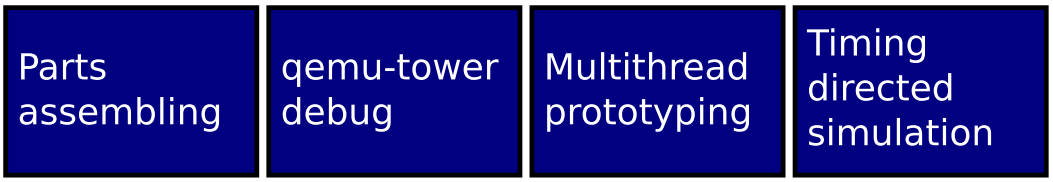


Figure 4.2: Stages of development

4.1 Development

We will present four separate stages of development. The different stages for building this project have already been presented and are shown in figure 4.2. A brief explanation follows:

Parts assembling The objective of this stage is to define the elemental blocks of which the framework will be composed. The basic layers of the project are presented, and also the way that those blocks are connected. The software used for each of those layers is also presented.

qemu-tower debug This stage was necessary in order to amend some problems which arose after the assembling stage. Those problems were due to inherent bugs and limitations in the existing QEMU project. To isolate the problems in QEMU project and successfully patch them a continuous interaction with the QEMU community was needed, and it resulted in a series of satisfactory patches accepted by the QEMU project. This stage is a technical step that involves shaping the QEMU backend to satisfy the project needs.

Multithread prototyping The goal of this stage is to imbue the multicore capabilities into the project. This is the most ambitious objective of this project: the multicore aspect of the framework. To achieve this capabilities, a multithread approach is used. Modifications must be done into the existing code. Multithread prototyping emulation routines are developed which enable the multicore emulation aspect.

Timing directed simulation The final step is ensuring the correctness of the timing behaviour. The emulated clock information and the coupling between the multicore control and the timing module are discussed in this stage.

The most time-consuming stage is the *qemu-tower debug*, and its results are mostly publicly available. Those changes are detailed in appendix A.

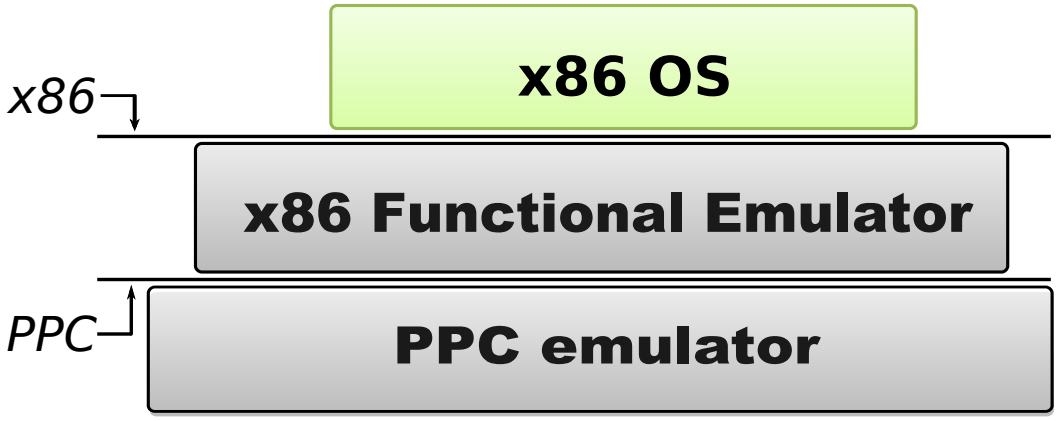


Figure 4.3: Simple layer structure of the framework

By the end of this development, a working framework is achieved. The results are discussed in the following chapter 5 and see also chapter 6 for the conclusions of this project and future work.

The following sections will present in further detail each of the development stages.

4.2 Parts assembling

The objective of this stage is to define the elemental tools of which the framework will be composed. The framework layers must correlate both the Hardware and Software aspects of the simulation. To achieve the desired fine grained emulation, a hierarchical layered structure is established. It is presented in the figure 4.3. The two bottom layers compose the qemu-tower, and are developed in this project. The PPC layer emulates the hardware and, above it, the x86 functional emulator behaves as the ESL and also contains the device emulation software.

The top layer is the unmodified operating system, the "guest OS". As stated before, qemu-tower provides an environment capable of x86 emulation. The junction point between the top layers indicates the exposed Instruction Set Architecture. The operating system "runs on" an x86 ISA which is functionally emulated by the qemu-tower. This set up mimics an operating system running in a system.

The qemu-tower is a two-layered framework: the x86 functional emulator and the PPC emulator. The required fine-grain simulation is achieved through a functional emulation of PPC instructions, which are issued by the x86 functional emulator. This means that the bottom layer, the PPC emulator, is modeling the hardware. The middle

layer is the x86 functional emulator which models the ESL. Globally those three layers mimic an OS being executed in a HW/SW co-design machine.

The layered approach is responsible for this project's codename: *qemu-tower*. From now on, we will be referring to this project as, simply, *qemu-tower*.

4.2.1 Software building process

Each layer of the framework is provided by different tool instances. QEMU provides both layers, but the execution is not straightforward, as they cannot run separately and independently.

Because the mix of different architectures, a cross-build tool[1] is used for obtaining a Power PC binary. Although this could be achieved through the use of a multi architecture environment, that solution arises even more complications in the given scenario. The cross-build QEMU instance is responsible of the x86 Functional Emulator.

A regular building of QEMU provides the Power PC hardware emulation. This regular `qemu-user` application can run the cross-build tool *x86 Functional Emulator*.

Given a sane enough cross-build environment, it is possible from a single average x86 machine to modify and build the whole simulation framework, and it is also possible to execute the simulation from the same computer. This is an advantage for researchers, as they have an easy-to-deploy simulator without the need of external or specific hardware.

4.3 qemu-tower debug

This stage was necessary in order to amend some problems which arose after the assembling stage. Although QEMU is a mature open source project, it doesn't mean that it will work flawlessly in any environment. As soon was discovered, there were hidden bugs in the software which prevented the previous tower structure from working. And on top of that, to achieve some required objectives, new features had to be included in the project. Both the architecture and essence of QEMU are untouched, but the found errors and some implementation schema are modified.

A large quantity of hours have been dedicated to this stage. Work done in this stage has mostly been reverted into QEMU community. The changes submitted are detailed in appendix A and general considerations about this modification process was explained in 3.5.1.2.

[1] A *cross-build tool* is a set of compilers and libraries which allow to build binaries for foreign architectures from a certain host Operating System. In this project, a standard x86 computer is used, and the cross-build tool allows PowerPC binaries creation.

4.3.1 Emulation chaining

This issue has been presented in 3.6. The framework must cope with the chain emulation which includes but it is not restricted to the emulation of certain *accessory instructions*. The first pitfalls prior to this stage were found when the QEMU was being run inside itself. In some point of the QEMU documentation [14], the importance of self-virtualization is stated:

> [Self-virtualization] QEMU was conceived so that ultimately it can emulate itself. Although it is not very useful, it is an important test to show the power of the emulator.

Unfortunately, QEMU is a complex piece of software. Self-virtualization may be an important test, but there is not any recent effort from QEMU development community to achieve self emulation. On top of that, the emulation inside an emulator is almost never used in any real-world applications, so there is no useful feedback for accomplishing this objective. The modifications done in this project to QEMU provides to this framework the ability to work in this fashion. These modifications are a high value byproduct of this project.

From the QEMU point of view, there were no self-emulation capabilities before the start of this development stage. That means that the command `qemu-user` was not capable of performing emulation of the binary `qemu-system`. This was due to the lack of some features in `qemu-user`, and some further incompatibilities between them. The needs of `qemu-system` were analysed and the desired features, like implementation of certain system calls and multithread implementation, were added and modified from QEMU project. As we have stated, those changes are publicly available into the official development tree and even included in QEMU official stable releases.

Once this improvement process was completed, support for basic emulation chaining was achieved. It is possible to run an Operating System through a `qemu-system` binary which is being emulated by a `qemu-user`. This configuration allows the emulation chaining to work through both layers of the framework, allowing an accurate simulation of a HW/SW co-design behavior.

4.4 Multithread prototyping

This step is one of the most important development of this project. The goal of this stage is to imbue the multicore capabilities into the project. To achieve this, a multithreaded emulation is presented.

To achieve a fine-grain simulation framework, a fine control of multiple threads is planned. Each Virtual CPU[2] is allocated and executed in an independent thread. This schema allows individual control of each core.

When starting the code development for this multithread prototyping, two main issues were revealed:

QEMU core management Multithreading is not the default QEMU VCPU management, to lead to some problems on the QEMU core management.

Thread unsafe translation QEMU internals translation is not ready for a multithread approach, therefore the lack of thread safe translation must be addressed.

Those two aspects are presented bellow.

4.4.1 QEMU core management

For system emulation mode, QEMU does a sequential management of each core[3]. A (very) simplified control hierarchy is the one showed in figure 4.4. The *emulation thread* contains a `foreach vcpu` kind of sequential management.

The proposed solution is dedicating a thread per VCPU, so we modified the control hierarchy to have those threads representing each Virtual CPU. The execution of each one of this threads can be precisely controlled and contains all the context that the PPC emulator and the timing simulator may need. The modified control hierarchy is showed in figure 4.5. In the modified schema, there are multiple threads dedicated to VCPU execution. Each of those threads (dashed blocks in the figure) relates to a VCPU executing code.

The modified *qemu-system* includes the desired multithread schema and this multithread prototyping provides the project with the desired multicore simulation capabilities.

[2]VCPU, in the QEMU software project context, refers to the software blocks responsible for the CPU emulation.

[3]QEMU is indeed able to perform real parallel simulation under the KVM kernel module. This only applies on environments where hardware virtualization is enabled and the host and target architecture match.

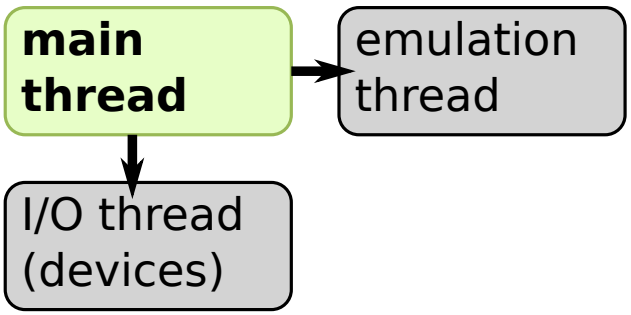


Figure 4.4: Simplified thread diagram of qemu-system

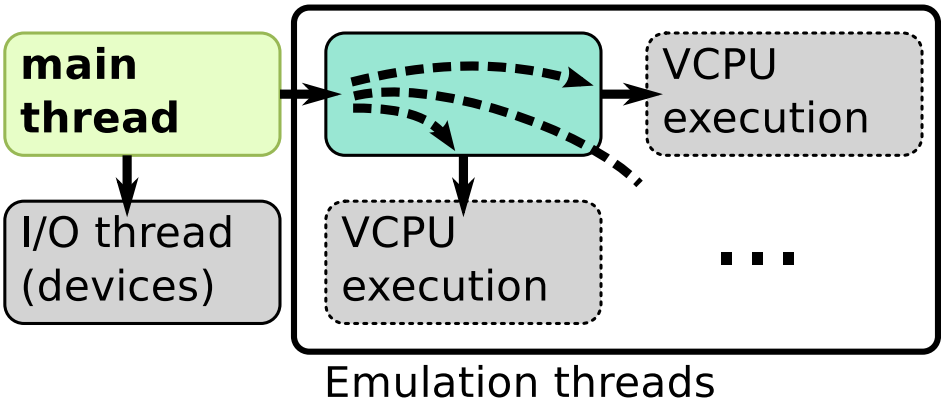


Figure 4.5: Simplified thread diagram of modified qemu-system

4.4.2 Thread unsafe translation

The binary translation of code is done, in QEMU internals, with the TCG (Tiny Code Generator, which initially was born as a backend for a C compiler, and then simplified by QEMU developers to be used as the binary translator) [16]. This library is not thread safe, which makes it harder for the multithreaded qemu-tower to work, and may not be ideal for modelling any arbitrary ESL.

Although there exist efforts towards a natively parallel QEMU [17–19], there aren't "official" patches available for latest releases of QEMU. Some comments, patches and proposals have been submitted in the developer's mailing list but no code has successfully been applied to the main branch.

This pitfall may evolve depending on QEMU development roadmaps and community efforts. Unfortunately, at this moment, qemu-tower will run TCG code in mutual exclusion to avoid those thread problems. This is not the most desirable scenario, and the aforementioned mutual exclusion fashion implies that only one thread at a time may be doing code translation.

An example scenario where this may be a problem: when simulating an ESL with parallel (multicore) code translation (not execution), the framework is not capable of fine-grain emulation of this concurrent translation.

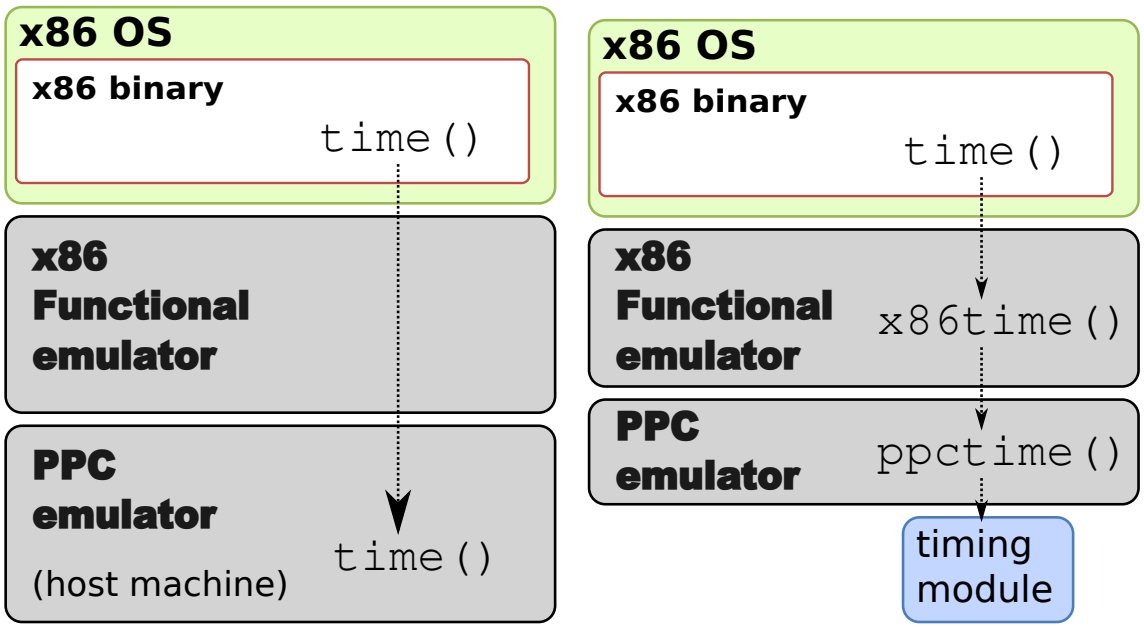
Fortunately, in a general case scenario this is not harmful because the execution flow is unaffected. Most running time of a HW/SW co-design involves execution and not code translation.

4.5 Timing directed simulation

The final step for this design is ensuring the correctness of the timing behaviour. We have stated that this project emulation will flow through a timing-directed simulator. This issue has not yet been revised in previous stages. In this chapter we address some notes on timing aspects of the framework.

By the end of this stage, the framework is ready for some basic timing-directed behaviour. Once all timing aspects are correctly addressed, a realistic hardware model may be coupled into the framework.

QEMU default time management is using the host machine clock as the timing source for everything. In a timing-directed simulation like the one presented in this project, sharing a clock for every block in the framework makes the simulation infeasible. The next subsection presents a solution for time isolation.



(a) QEMU defaults (b) Desired qemu-tower

Figure 4.6: Different behaviour of time functions

4.5.1 Time isolation

As a default, QEMU uses the host clock. Most uses of QEMU involves virtualization environments. The simplest way of dealing with clock is using the host clock and its timing information. Using a real life clock as a reference is a common requirement for a lot of networking services, and does not usually impede correct emulation. For those reasons QEMU uses this clock schema. In figure 4.6a there is a simplification of the default unmodified QEMU behaviour.

However, qemu-tower doesn't aim to provide network services, nor is interested in virtualization environments. The simulation should be as deterministic as possible, and accurate timing is needed for an accurate and realistic HW/SW co-design simulation. The diagram 4.6b presents the desired modifications which provide the ability of a realistic simulated clock. This realistic clock model would take into account the time of execution and other timing simulated aspects.

Using the qemu-tower in a default configuration QEMU (figure 4.6a would result in the following: for any time instruction issued in the guest operating system, the framework emulation would use the host clock for the result. On the other hand, in the desired qemu-tower (figure 4.6b, the behaviour would be different: the timing module is responsible for the results of time instructions. The new design allows the framework

to respond to time instructions in a consistent manner. The timing simulator uses the executed instructions as the basis for time calculations. The system becomes more deterministic, thus also enhancing repeatability.

The ideal set-up would provide a virtual clock consistent with the executed instructions. This virtual clock would be independent of any external aspects (like the simulation gear).

As stated at the beginning of this chapter, the timing related parts are used as place-holders. The simplified approach used in this development stage provides a timing based on executed instructions. Whenever a research requires a certain hardware model or any kind of special timing feature, the timing module can be developed and imbued into the framework. Obviously, this timing module is hardware- and design-dependant.

CHAPTER

5

RESULTS

The figure 5.1 shows a graphical window of the QEMU in `qemu-tower` set up. In this window, an Operating System and a four thread test application have been run. The framework is capable to execute a Operating System and a test program in it. This test was executed under a four-core configuration.

Although there is a lot of work dedicated to this project, a computer architecture simulator like the presented one is not example- friendly. The focus of this project is providing a tool for research and development, not the simulations itself, as the simulations are future applications of the framework. To demonstrate that this project provides everything that an actual application would need, two examples are shown below: *Initialization* and *Thread interleaving*. Finally, 6.5 presents some future related research.

Both examples introduced in this chapter focus on the behaviour of the framework under a multicore configuration. They are relevant because they reflect the work done during the development stages. Under the unmodified backend emulator –that is, with the QEMU project as-is– the initialization was different and thread interleaving was not possible at all. Those example are not the objective of the project, building the framework is, but they illustrate requirements that any Hardware/Software multicore co-design simulation would need. Those requirements are not fulfilled by the previous state-of-the-art.

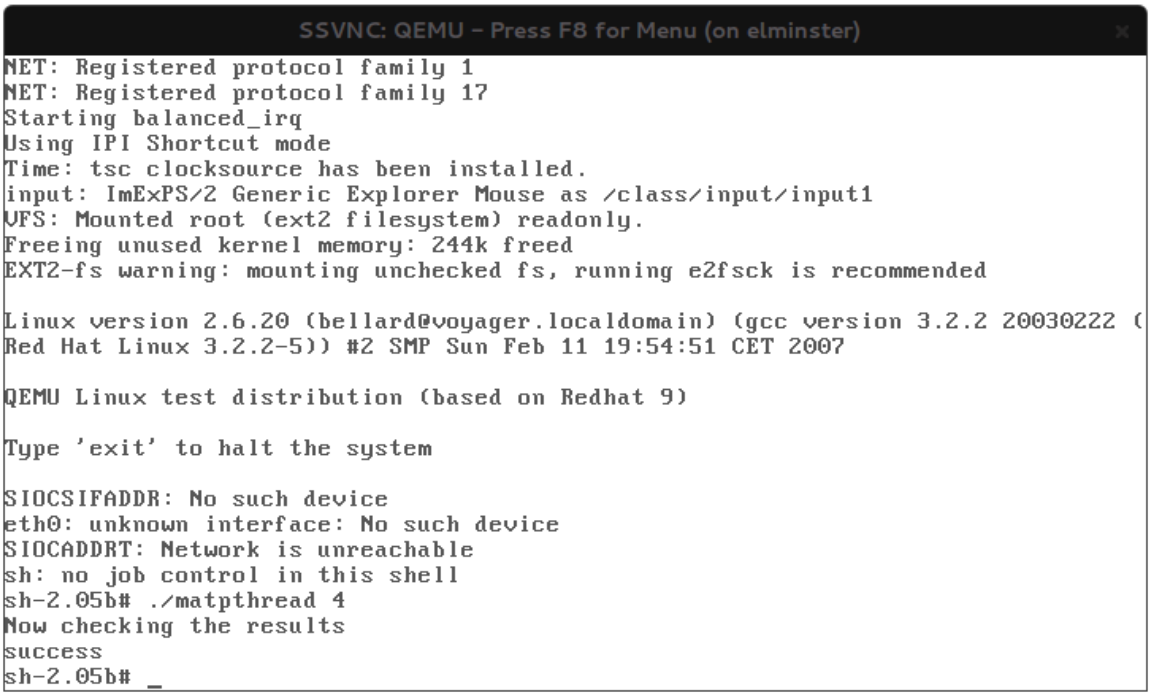


Figure 5.1: Result of the benchmark (matrix multiplication in minimal Linux system)

5.1 Initialization

The Sequence Diagram of this same example is presented in figure 5.2. For the sake of simplicity, only two threads are represented in the diagram. The original test was done with 4 threads.

The control thread starts by a *Spawning* state which starts the needed threads. The instantiation of threads is done sequentially. Once all the VCPU threads have been spawned, a new and recurrent stage of *Execution* starts. Because the multicore approach of the framework is based on a multithreaded framework, the execution of the threads can be done in parallel. Obviously there is control from the *Control thread* and numerous synchronization points between threads, but the basic idea of parallel execution remains.

This thread control allows to the system to govern each simulated processor independently. This feature will enable future research on different multithreaded alternatives, e.g. the case in which one of the threads runs with different frequency than another.

The more detailed and complex data of this example is the following verbose output:

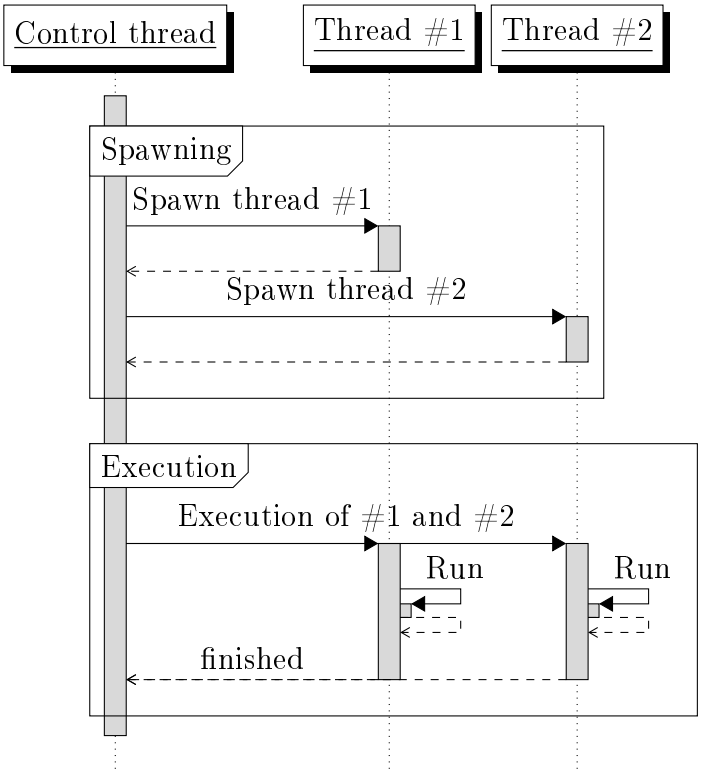


Figure 5.2: Thread initialization sequence diagram

Thread initialization verbose output

```
qemu-tower-user[0xf5206b70]: New thread (#0) about to start execution
qemu-tower-user[0xf519ab70]: New thread (#1) about to start execution
qemu-tower-user[0xf50feb70]: New thread (#2) about to start execution
qemu-tower-user[0xf508eb70]: New thread (#3) about to start execution
qemu-tower-sys: Spawning threads --initialization
qemu-tower-sys[0x45d05300]: Starting thread #0
qemu-tower-user[0xf4f9ab70]: New thread (#4) about to start execution
qemu-tower-user[0xf4f9ab70]: VCPU thread
qemu-tower-sys[0x4f8d0300]: I am spawned --number #0
qemu-tower-sys[0x45d05300]: Starting thread #1
qemu-tower-user[0xf4f26b70]: New thread (#5) about to start execution
qemu-tower-user[0xf4f26b70]: VCPU thread
qemu-tower-sys[0x500d0300]: I am spawned --number #1
qemu-tower-sys[0x45d05300]: Starting thread #2
qemu-tower-user[0xf4ebab70]: New thread (#6) about to start execution
qemu-tower-user[0xf4ebab70]: VCPU thread
qemu-tower-sys[0x508d0300]: I am spawned --number #2
```

```
qemu-tower-sys[0x45d05300]: Starting thread #3
qemu-tower-user[0xf4e4eb70]: New thread (#7) about to start execution
qemu-tower-user[0xf4e4eb70]: VCPU thread
qemu-tower-sys[0x510d0300]: I am spawned --number #3
qemu-tower-sys: Threads spawned
qemu-tower-sys: The 0 CPU should run briefly...
qemu-tower-sys: The 1 CPU should run briefly...
qemu-tower-sys[0x4f8d0300]: Starting
qemu-tower-sys: The 2 CPU should run briefly...
qemu-tower-sys[0x500d0300]: Starting
qemu-tower-sys[0x500d0300]: Finished work
qemu-tower-sys[0x508d0300]: Starting
qemu-tower-sys[0x508d0300]: Finished work
qemu-tower-sys[0x4f8d0300]: Finished work
qemu-tower-sys: The 3 CPU should run briefly...
qemu-tower-sys[0x510d0300]: Starting
qemu-tower-sys[0x510d0300]: Finished work
qemu-tower-sys: Barrier is over on controller
```

In this example there are verbose output of both the System emulation `qemu-tower-sys` and the binary translation `qemu-tower-user`. There is redundancy between debugging lines because of the layer chaining of the framework. The debugging begins with the first stage, where threads are spawned. Once everything needed has been spawned, the *Execution* stage starts and each VCPU is started. The debugging ends at the first synchronization across threads –`barrier` on `controller`.

5.2 Thread interleaving

We have previously analysed in 4.4.2 some drawbacks of a multithread configuration of QEMU. With the modified backend emulator QEMU, the multicore system is emulated in a multithreaded fashion. Independently of the emulation backend, the parallel binary translation is something Hardware/Software co-design specific, as there can be parallel designs or sequential ones. The framework infrastructure has potential to use any parallel configuration. In the presented example, the correctness of a global lock mutual exclusion configuration is shown.

To show this global lock configuration, a sequence diagram is shown in figure 5.3. In this example two VCPU threads are depicted, and their interaction with the binary translation logical function of the system is also shown.

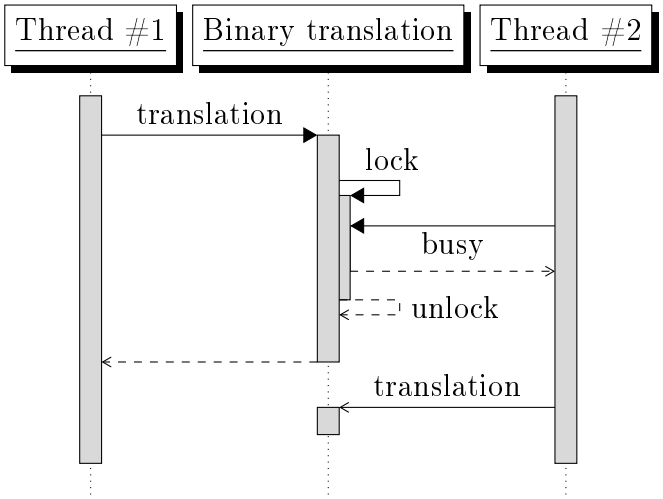


Figure 5.3: Thread interleaving sequence diagram

A binary translation starts when some thread requires it and there is no pending translation work. The binary translation then starts a lock which prevents any other thread from starting a translation. When the translation has ended, there is an unlock and the execution returns to the calling thread.

The complete debugging data of an interleaving example comes below:

Example of thread interleaving

```
[#2]: Starting thread on qemu-system
[#3]: Starting thread on qemu-system
[#3]: Starting CPU exec
[#3]: proceeding to lock
[#3]: locked
[#3]: proceeding to unlock
[#3]: unlocked
[#0]: Starting thread on qemu-system
[#2]: Starting CPU exec
[#3]: cpu_loop_exit (not unlocking)
[#3]: Finished work
[#0]: Starting CPU exec
[#2]: proceeding to lock
[#2]: locked
[#0]: proceeding to lock
[#1]: Starting thread on qemu-system
[#1]: Starting CPU exec
[#1]: interrupt point (setjmp)
```

```
[#1]: proceeding to lock
[#2]: proceeding to unlock
[#2]: unlocked
[#0]: locked
[#2]: proceeding to lock
[#0]: proceeding to unlock
[#0]: unlocked
[#1]: locked
[#1]: unlocked
[#2]: locked
[#1]: cpu_loop_exit (not unlocking)
[#0]: cpu_loop_exit (not unlocking)
[#2]: proceeding to unlock
[#1]: Finished work
[#2]: unlocked
[#0]: Finished work
[#2 keeps working ...]
```

In this debugging output example the same principle applies. The `lock-unlock` process is the same as the one presented in figure 5.3 but the output corresponds to 4 different VCPU threads. Only the VCPU thread debugging data is shown. Each VPU works and proceeds to lock whenever there is translation required. When the VCPU finishes, it issues the `cpu_loop_exit` call and exits into the synchronization stage along all the other VCPU and the control thread.

CHAPTER

6

CONCLUSIONS

The main goal was preparing an emulation block for Hardware/Software co-design multicore systems. Because the state-of-the-art of emulation tools for HW/SW co-design systems is not suited to simulate multicore capabilities, the qemu-tower fills this need by providing a feature full framework.

6.1 First pillars

To achieve a functional simulator, a combination of emulator layers was planned and QEMU was proposed as a backend emulator.

QEMU is a valuable tool, and is a well-tested off-the-shelf component. The initial planning considered that assembling a two-layer architecture capable of chain emulating instructions would be an easy way to provide a HW/SW co-design-like architecture while being capable of fine-grain emulation.

Unfortunately, adapting QEMU into the project framework has not been a straightforward process as we first thought. Debugging and adding features to the QEMU project has been an important and extensive task. This task is key, not only for the qemu-tower project itself, but to anyone using QEMU code –either as a tool or as a software suite.

The work done in the aforementioned open-source community driven project – QEMU software [14]– has already received external validation from the developer community: a series of patches have been sent and accepted, along some technical observations, to the developer mailing list. A brief extract of some of those mails and patches can be read in appendix A. Most of those patches have been approved by the main project managers and are already in the stable branch of QEMU.

The extensive work performed in the QEMU project has required a considerable amount of dedication, but has resulted in high quality improvements. Those new features and the bug corrections of QEMU are key for this project and for any other project which may follow qemu-tower framework layout. The improvements in the QEMU project benefit also to the whole user community, because some features and bugs are related not exclusively to the custom build of QEMU for the qemu-tower project, but can affect normal QEMU execution.

6.2 Viability of simulation

Qualitatively, the system is capable of simulating code from an OS successfully. The layered structure exposes both the software-provided ISA (the one that the ESL presents to the OS) and the hardware instruction set.

The complexity of the emulation layers is high. To ensure a robust enough research tool, the framework is capable to work with standard QEMU system images. This means that the desired snapshot of the simulation is acquired through a vanilla QEMU and then the framework can simulate it maintaining repeatability.

On top of the complexity of the layers, fine-grain simulation is a required step in qemu-tower roadmap. To achieve the fine-grain simulation in the framework some components need to be integrated into the simulation infrastructure. Once the fine-grain simulation works, the next step becomes fully accurate multicore simulation.

Below are described those most representative components integrated into the framework:

Timing module provides time isolation between layers and models the time flow of the simulated environment.

ESL is capable of binary instruction translation. The ESL is part of a HW/SW co-design, and its presence is basic in a HW/SW co-design emulation framework.

We integrated an initial implementation of these components for debugging purposes, but the users of this infrastructure should implement the actual components that model the specific constraints of the environment they want to simulate.

6.3 Multithread for multicore

This project has a clear top goal: capabilities for thorough multicore simulation. This goal motivated a certain layout for the framework basic blocks and some technical decisions.

One important design decision was enabling multicore simulation through the instantiation of multiple threads. Each of those threads is modelling a certain Virtual CPU. Each context of the VCPU is present in a different thread, and a fine-grain control of the emulation threads results in a fine-grain emulation of the system.

Achieving the fine-grain emulation is key for a time-directed simulation: it provides the features needed for the multicore simulation, enabling timing-directed simulation. The inner tools of the project had to be suited for this objective. In our situation, modifications to the QEMU VCPU management had to be performed.

6.4 Final state

The developed framework performs combined emulation for both the software layer and hardware layer. The emulation is designed to have fine-grain capabilities, a feature key for the multicore emulation goal.

The project includes QEMU as the main emulation tool and custom blocks, like the timing module and the adapted ESL. The QEMU official release includes a handful of features and bug corrections related to this project work, although they benefit everyone using QEMU software.

6.5 Opened research avenues

The foundations for a fine-grain simulation of HW/SW co-design suited for multicore systems are successfully established. Further projects can use this project to achieve different objectives, like:

- Provide accurate timing information by adding a custom timing module.
- Research on ESL software by adapting different ESL into the qemu-tower while adding profiling

and even a combination of the above.

Related to the ESL aspect of the framework, research on the multicore and parallel characteristics of the binary translation and optimization can be addressed. There is a whole field of research on which binary translation configuration fashion is appropriate.

The framework provides simulation for both the Software Layer and the Hardware Layer. Hardware multicore designs with special characteristics, dedicated accelerators, and such may be simulated into the framework and its result combined with ESL research.

A plausible scenario is the merge between DARCO [1] and qemu-tower. This would yield a multicore framework which would include all the ESL research done in the DARCO project and its hardware model. The research already done in the DARCO project may, in that case, be applied to multicore systems with low framework development overhead. But not only for the DARCO project: qemu-tower can provide significant improvements to the state-of-the-art existing simulation frameworks and tools by providing a foundation for the desired multicore capabilities.

APPENDIX

A

CHANGES SUBMITTED TO QEMU PROJECT

Following sections present the different patches done to the QEMU project to enable it to be used in the qemu-tower project.

Those changes include both bugs and features. The objective of those is polishing QEMU to shape it into a tool usable in a multithreaded (multiple CPU) multilayer (QEMU inside QEMU) emulation.

A.1 Preparing safe sigprocmask wrapper

The qemu-user binary has no protection on linux signals. In some programs, this may lead to problems when the emulated binary does some singal mask manipulation.

When trying to execute the qemu-tower, there were segmentation faults. Some signals used internally by qemu-user are used as well by qemu-system. This is the case of the `SIGSEGV` signal.

The solution chosen is adding sanity checks on the syscall wrappers. Before this patch, the wrapping of system calls which modified the signal mask was done directly.

Status: **Pending approval** (it works on qemu-tower)
Last submission: `Sat, 20 Oct 2012 16:15:55 +0200`
http://lists.gnu.org/archive/html/qemu-devel/2012-10/msg03638.html

A.2 register align p{read, write}64

The wrappers for the system calls `pread64` and `pwrite64` doesn't manage variable alignment. When 32 bits architectures align 64 bits variables in different ways, those system calls do not work correctly.

Currently, qemu-tower is using 32 bits binaries on both architectures x86 and PPC. The alignment of the system calls is different, so the qemu-tower failed whenever a disk device read was issued. The first symptom detected was bad snapshot recognition, because of incorrect file access read syscalls.

The first solution was a "quick & dirty" argument correction. Simultaneously, I collaborated on the mailing list and a robust patch was made.

Status: **Solved**
Last submission: `Sun, 30 Sep 2012 03:32:39 +0200`
http://lists.gnu.org/archive/html/qemu-devel/2012-10/msg00018.html

A.3 Bug on a zero comparation with sas_ss_flags

The implementation of `sigaltstack` syscall was faulty, as it had a bad comparison.

I discovered this while using the sigaltstack family functions (see A.4). The sigaltstack does not work at all under the PPC because a zero comparison is done just in the wrong way. This error may had been undetected because the typo is only under PPC architecture –on all the other architectures the comparisons are correct.

Status: **Solved**
Last submission: `Thu, 15 Mar 2012 09:52:06 +0100`
http://lists.gnu.org/archive/html/qemu-devel/2012-03/msg02947.html

A.4 New sigaltstack backend for coroutine

The QEMU development uses coroutines as a way of managing asynchronous code. As can be read in the developer mailing list:

> This patch introduces coroutines which allow code that looks synchronous but is asynchronous under the covers.
>
> A coroutine has its own stack and is therefore able to preserve state across blocking operations, which traditionally require callback functions and manual marshalling of parameters.

Full description and implementation details can be found in [22].

The technical realization of those coroutines was done through ucontext syscalls. As the date of this document, there is not any ucontext functions support on qemu-user. On top of that, ucontext system calls are not really standard.

My decision was to implement a new backend with coroutines with sigaltstack [23]. This is a POSIX standard, and GNU Portable Threads [24] uses successfully a similar set up.

The new implementation is working, in some cases is even preferred over the old one and eases the configuration of qemu-user to be able to run qemu-system.

Status: **Feature available**
Last submission: `Tue, 28 Feb 2012 12:25:48 +0100`
http://lists.gnu.org/archive/html/qemu-devel/2012-02/msg03871.html

---

BIBLIOGRAPHY

[1] Demos Pavlou, Aleksandar Brankovic, Rakesh Kumar, Maria Gregori, Kyriakos Stavrou, Enric Gibert, and Antonio Gonzalez. DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines. In *Proceedings of the 4th Workshop on Architectural and Microarchitectural Support for Binary Translation, held in conjuction with ISCA-38 (AMAS-BT)*. **2011**.

[2] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc. **2005**.

[3] Wine. **March 2013**.
URL http://www.winehq.org/

[4] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java™ Virtual Machine Specification*. Oracle Corporation. **February 2013**.
URL http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[5] Oracle Coroporation. *Oracle VM VirtualBox User Manual*. **September 2013**.
URL https://www.virtualbox.org/manual/

[6] Xen. Xen Hypervisor. **February 2013**.
URL http://xen.org

[7] Red Hat Emerging Technology Project. Kernel Based Virtual Machine. **February 2013**.
URL http://www.linux-kvm.org/page/Main_Page

[8] Kemal Ebcioğlu and Erik R. Altman. DAISY: dynamic compilation for 100% architectural compatibility. *SIGARCH Comput. Archit. News*, pp. 26–37. **May 1997**.

[9] IBM. DAISY. **March 2013**.
URL http://researcher.watson.ibm.com/researcher/view_project.php?id=2909

[10] Sumedh Sathaye, Paul Ledak, Jay Leblanc, Stephen Kosonocky, Michael Gschwind, Jason Fritts, Arthur Bright, Erik Altman, and Craig Agricola. BOA: Targeting Multi-Gigahertz with Binary Translation. *In Proc. of the 1999 Workshop on Binary Translation, IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pp. 2–11. **1999**.

[11] MAME. Multiple Arcade Machine Emulator. **March 2013**.
URL http://www.mame.net/

[12] Stephen R. Goldschmidt and John L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. *ACM SIGMETRICS conference*. **1993**.

[13] Bochs. The cross platform IA-32 emulator. **February 2013**.
URL http://bochs.sourceforge.net/

[14] Open Source Project. *QEMU*. **February 2013**.
URL http://wiki.qemu.org/Manual

[15] QEMU Project. *Developer mailing list*. **2012-13**.
URL http://lists.gnu.org/archive/html/qemu-devel/

[16] Fabrice Bellard. *Tiny Code Generator*. QEMU, tcg/README. **February 2013**.

[17] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. PQEMU: A Parallel System Emulator Based on QEMU. *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pp. 276–283. **2011**.

[18] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. *Proceedings*

of the Tenth International Symposium on Code Generation and Optimization, pp. 104–113. **2012**.

[19] Zhaoguo WANG, Ran LIU, Yufei CHEN, Xi WU, Haibo CHEN, Weihua ZHANG, and Binyu ZANG. COREMU: a scalable and portable parallel full-system emulator. *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pp. 213–222. **2011**.

[20] John L. HENNESSY and David A. PATTERSON. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, third edition. **2003**.

[21] Mike JOHNSON. *Superscalar Microprocessor Design*. Prentice-Hall Inc. **1991**.

[22] Kevin WOLF. *coroutine: introduce coroutines*. QEMU developer mailing list. **August 2011**.
URL http://lists.gnu.org/archive/html/qemu-devel/2011-08/msg00448.html

[23] Linux Programmer's Manual. *SIGALTSTACK(2)*. **September 2010**.

[24] Ralf S. ENGELSCHALL. *GNU Portable Threads*. **2006**.
URL http://www.gnu.org/software/pth/pth-manual.html