Thesis no: MECS-2014-02

Paravirtualizing OpenGL ES in Simics

Eric Nilsson

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering, Game and Software Engineering. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Eric Nilsson
E-mail: erne09@student.bth.se

External advisor:
Erik Carstensen
Intel Corporation

University advisor:
Prof. Håkan Grahn,
Research Dean of Faculty
Faculty of Computing

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Full-system simulators provide benefits to developers in terms of a more rapid development cycle; since development may begin prior to that of next-generation hardware being available. However, there is a distinct lack of graphics virtualization in industry-grade virtual platforms, leading to performance issues that may obfuscate the benefits virtual platforms otherwise have over execution on actual hardware.
Objectives. This dissertation concerns the implementation of graphics acceleration by the means of paravirtualizing OpenGL ES 2.0 in the Simics full-system simulator. Furthermore, this study illustrates the benefits and drawbacks of paravirtualized methodology, in addition to performance analysis and comparison with the Android emulator; which likewise utilize paravirtualization to accelerate simulated graphics.
Methods. In this study, we propose a solution for paravirtualized graphics using Magic Instructions; the implementation of which is subsequently described. Additionally, three benchmarks are devised to stress key points in the developed solution; comprising areas such as inter-system communication latency and bandwidth. Additionally, the solution is evaluated based on computationally intensive applications.
Results. For the purpose of this study, elapsed frame times for respective benchmarks are collected and compared with four platforms; i.e. the hardware accelerated Host machine, the paravirtualized Android emulator, the software rasterized Simics- and the paravirtualized Simics platforms.
Conclusions. This thesis establishes paravirtualization as a feasible method to achieve accelerated graphics in virtual platforms. The study shows graphics acceleration of up to 34 times of that of its software rasterized counterparts. Furthermore, the study establishes magic instructions as the primary bottleneck of communication latency in the devised solution.

Classification: E.1.1 [Software infrastructure]: Virtual machines; K.6.4 [Graphics systems and interfaces]: Graphics processors; N.1.0 [Companies]: Intel Corporation;
Keywords: Paravirtualization; Simics;

Preface

Acknowledgement

During my employment at Intel® I have had the opportunity to work with an intriguing product employed by a wide array of significant players in the software industry. I have been met by a creative work environment; being granted exchanges with ambitious colleagues in an internationally competitive trade. As such, I would like to express my sincere gratitude to Intel® and its flourishing work culture. Furthermore, I would like to thank everyone at the Intel® and Wind River Systems, Inc. offices in Stockholm, all of whom have been very welcoming during my stay.

In particular, the following persons ought be acknowledged for their contribution to the project. These persons are ordered alphabetically in accordance to their surnames[1].

- Daniel Aarno (Intel®)
- Erik Carstensen (Intel®)
- Anders Conradi (Intel®)
- Mattias Engdegård (Intel®)
- Prof. Håkan Grahn (Blekinge Institute of Technology)
- Christian Häggström (Intel®)
- Stefan Lindblad (Intel®)
- Jakob Skoglund (Wind River Systems, Inc.)
- Magnus Vesterlund (Wind River Systems, Inc.)
- Bengt Werner (Intel®)

Additionally, I would like to acknowledge the contributions of Erik Carstensen (Intel®) and Prof. Håkan Grahn (Blekinge Institute of Technology) whom have acted industry- and university advisor, respectively, throughout the course of this project, in addition to Alexander Mohlin - the acting student opponent to this thesis.

[1]Note that the order of the persons named is not indicative of the magnitude of their contribution.

Structure and Formatting

This section is presented to give the reader an idea of how the document is structured and formatted, and what to expect when reading the material presented in this thesis. Points of interest are summarized in the list below.

- The first occurrence of acronyms are written in full, e.g. GNU (GNU's Not Unix!).
- Some acronyms are not expanded, such as CPU. This is due to said acronyms having become standardized in their usage. As such, the reader is expected to be familiarized with the acronyms in question. These are still featured in the List of Acronyms, however.
- The introduction of new terminology is capitalized in its first occurrence, e.g. Semiconductor.
- The document refrains from using formatting such as italics and bold text in order to increase the readability of the document.
- Terminology and acronyms are presented in the List of Terms and List of Acronyms in the back matter of this dissertation.
- Many terms and acronyms used throughout the course of this document are hyperlinked to said compiled lists.
- Square brackets are used solely for citations.
- For the purposes of discerning complementing text from references, nestled parentheses are not distinguished using any other formatting.
- Cross-references to body matter divisions (such as chapter, section, or paragraph) are purposely non-capitalized, as to increase document readability.
- For the sake of visualization, values outside of corresponding standard deviation are not presented in graphs unless specified otherwise.

Industry Collaboration

This study was performed at the Intel® offices in Stockholm, in collaboration with Intel® and Wind River Systems, Inc. During this time, the author was employed at Intel® and performed the majority of this thesis as part of his duties there.

Accessibility

This document, along with its source code and revision history, is hosted as an open source repository which includes the entirety of the raw data collected for the study. As such, this data may be reviewed and analyzed for future studies. Furthermore, the process of releasing the benchmarking source code, under an open source license, is underway at the Intel® Open Source Center. References to said source code may be found in latter revisions of this document.

In the thesis back matter, a unique commit hash is enclosed identifying the corresponding Git commit and its changes. The Git repository, along with supplementary material produced during the study, may be found at:

www.github.com/CaterHatterPillar/dv2524

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Virtual platforms are becoming an important tool in the software industry in order to provide cost-effective Time-to-Market (TTM) gains and meet the ever-shortening product life-cycles [24][37] (see [22] for an overview of the benefits of full-system simulation). Virtual platforms deliver these TTM-benefits in two major ways. Virtual platforms enable pre-silicon development, that is; software development can begin prior to next-generation hardware being available [24]. Furthermore, virtual platforms may provide additional development tools compared to working with actual hardware. For example, some virtual platforms allow simulated systems, often known as simulation Targets, to be stopped synchronously without affecting Timing or states of the target software [39], and allow investigation of race conditions and other parallel programming issues [31][21]. Additionally, such platforms may allow intricate inspection of simulated hardware, such as memory, caches, and registers [24]. Some virtual platforms provide advanced features such as Reverse Execution [21] (the ability to run a simulation backwards) and Checkpointing [21] (functionality to save- and restore the state of a simulation). These features are useful for debugging and testing a diverse range of software; from firmware to end-user applications [21].

There are several techniques to provide fast functional virtual platforms that are running CPU-bound workloads. Typical methods include Interpretation [33] (slowest), Just-in-Time (JIT) compilation [2], and Hardware-assisted Virtualization [2][21] (fastest). Virtual platforms using these techniques can typically achieve a simulation performance in the range of 10-1000 Million Instructions Per Second (MIPS) [2], but there is recorded performance of up to 5000 MIPS [21].

The GPU has become a vital part in delivering good user experience on many types of devices ranging from wearable, hand held, and portable units to desktop computers. In contrast to CPU-bound workloads, when developing GPU kernels, seemingly insignificant additions to code may cause significant changes in performance due to massively parallelized instruction sets (see Performance Considerations by Kirk and Hwu [17] for an analysis on the volatility of GPU performance). Since GPUs operate significantly different from CPUs, they pose unique challenges to designers and developers (see Parallel Programming and Computational Thinking by Kirk and Hwu [17] for an elaboration on General

Purpose computing on Graphics Processing Units (GPGPU) methodology). The increased utilization of GPUs for general purpose workloads has extended the need for virtualization of such hardware in situations when hardware is busy, unavailable, not sufficient, or for the purposes of debugging and profiling[1]. Yet seemingly few virtual platforms support virtualization of GPUs, despite their influence on modern computing.

For the purposes of simulation, different solutions follow varying use-cases. For example, developers interested in benchmarking or driver development for next generation GPU- or CPUs may require detailed simulators that provide insight into execution engines and pipelines [31]. Albeit applicable in certain use-cases and capable of running 'toy' applications, such platforms are often orders of magnitude too slow to run commercial workloads [24]. Application developers, on the other hand, do not necessarily care for the internal workings of hardware as they typically work at a higher abstraction level, for example; utilizing a graphics Application Programming Interface (API) (such as OpenGL) that, in turn, communicate with the device driver. As such, application developers may be more interested in achieving decent simulation performance rather than a timing-accurate processor model (see [21] for an analysis of compromises in system simulation). However, due to large differences in-between CPU- and GPU-architecture, simply delegating commonly GPU-bound workloads to the CPU is rarely feasible in terms of performance.

Common approaches to achieve better simulation performance includes creating a functionally accurate model of the GPU (inducing the advantage of being able to utilize the same software (drivers, libraries, etc.) as the simulated platform), where internal details may be simplified, or using software rasterization without involving the GPU model. However, these methodologies traditionally incur heavy performance losses in comparison to GPU hardware acceleration. In order to achieve better performance, one may offload such kernels to the GPU of the system on which the simulation runs, often referred to as the simulation Host. There are a number of way to achieve such acceleration, such as relying on Peripheral Component Interconnect (PCI) Passthrough and similar technologies to grant access to the underlying host hardware from within the virtual platform [29] (see [54] for an overview on PCI passthrough), or utilizing a concept commonly referred to as 'Paravirtualization' at a higher level of abstraction (e.g., the OpenGL library).

Paravirtualization is a relatively new term originally defined as selectively modifying the virtual architecture to enhance scalability, performance, and simplicity [1]. Effectively, this means modifying the virtual machine to be similar, but not identical, to host hardware [5]. As such, one may simplify the virtual-

[1] An example of modern technology developed for said purposes is Microsoft Windows Advanced Rasterization Platform (WARP) - Microsofts latest addition to the DirectX framework (see [55]).

ization process by neglecting some hardware compatibility [38]. Implementing GPU simulation by the means of paravirtualization provides the benefits of improved simulation performance, albeit it may entail loosing some of the benefits a virtual platform can provide. For example, it may be challenging to support advanced features such as Deterministic Execution, checkpointing, and reverse execution. Additionally, paravirtualization entails modifying the simulated system, since some part of the target machine must be modified to accommodate the paravirtualized methodology. Yet, aforementioned technique is the preferred method of Google in the Android SDK when utilizing the QEMU virtual platform in order to simulate OpenGL ES.

This dissertation comprises an investigation of paravirtualization of OpenGL ES in the Simics virtual platform. The work presents an implementation of accelerated OpenGL ES 2.0 graphics by the means of paravirtualization and using Magic Instructions as a communications bridge. In addition to this, we present the development of three benchmarks stressing important attributes of the devised solution. Using the results collected from the execution of said benchmarks on a number of platforms, the solution is evaluated. Furthermore, this dissertation identifies performance bottlenecks obstructing large numbers of paravirtualized invocations for the purposes of real-time graphics. The results presented in this thesis show performance improvements of up to 34 times in relation to software rasterized counterparts.

The remainder of the report presents the objectives and question formulations adhering to the presented study. Additionally, an introductory chapter on relevant background and preceding work in the area - expanded upon by a succeeding chapter concerning the Simics full-system simulator - means to bring about many of the terms and concepts used in the subsequent material. Next, a description of the devised paravirtualized implementation is presented along with a chapter on experimental methodology. Furthermore, this thesis features an elaboration on potential threats to validity before detailing the results collected using the methods described in prior chapters. The material is then finalized by a speculative discussion chapter, analyzing the paravirtualized solution in regard to advanced functionality in the Simics full-system simulator, foregoing the conclusions which may be established from the compiled material. Finally, the dissertation is concluded by a report on recommended future work in the area.

Chapter 2
Aims, Objectives, and Research Questions

The study presented in this document consists of implementing paravirtualization of a graphics API (being OpenGL ES 2.0) in the Simics full-system simulator developed by Intel® and sold through Intel®'s subsidiary Wind River Systems, Inc. The solution devised to accelerate OpenGL adheres to that of the Android emulator, which is elaborated upon in section 3.4. As such, this study concerns investigating the performance, and the feasibility of extended benefits and advanced functionality, of paravirtualized graphics in a virtual platform. This entails investigation, analysis, and development of methods and techniques for efficient communication and execution in the Simics run-time environment. Furthermore, the study comprises analysis of the liabilities of paravirtualized technologies in regards to Simics philosophy (being high-performance determinism and repeatability [2]). Thus, the study does not exclusively concern Simics integration, but an investigation of paravirtualized libraries in virtual platforms.

For the purpose of this thesis, a paravirtualized solution for graphics acceleration in Simics is developed. Furthermore, to accommodate the analysis of benefits and drawbacks of paravirtualized graphics, a number of benchmarks are devised to stress key points in the developed solution; with the goal of locating performance bottlenecks. The benchmarks are designed to stress latency and bandwidth in target-to-host communication, in addition to computational intensity brought on by complex GPU-bound workloads.

Based on the devised solution, in coagency with the benchmarks, this study comprises an analysis of the performance of paravirtualized graphics compared to that of traditional software rasterization. The objectives of the dissertation is to evaluate the feasibility of paravirtualization as an approach to accelerate graphics in virtual platforms; along with identifying its strengths and weaknesses.

In line with previous work in the area specified in chapter 3, there has been no indication - in academic writing - of any pre-existing solution of paravirtualized graphics APIs signifying deterministic behavior; paving the way for supporting reverse execution graphics. Such functionality could simplify debugging, testing, and profiling of applications comprising some GPU-bound workload; not limited to graphics- or GPU utilization in its entirety. Entailed by these research gaps, the research questions formulated in this chapter are considered to be lacking in

the field.

As such, the study performed for the purposes of this thesis is relevant to the field of computer science by expanding upon the the knowledge of graphics acceleration in virtual platforms; in terms of facilitating debugging, testing, and profiling of software dependent on GPU graphics acceleration. By the means outlined in chapter 2, this dissertation contributes to the field of computer science by answering these questions from the perspective of graphics paravirtualization in the Simics full-system simulator. Accordingly, the key investigatory attributes and explicit research question formulations, sought to be answered by this thesis, are presented below.

1: *What constitutes a viable implementation of paravirtualized graphics?*
2: *What are the benefits and disadvantages of paravirtualized graphics?*
3: *How does paravirtualized graphics performance relate to software rasterization?*

Furthermore, the study pertains to a number of discussion topics in line with the solution's relation to the Simics full-system simulator. These discussion questions presented below.

1: *What inhibits deterministic execution of paravirtualized graphics?*
2: *What inhibits checkpointing of paravirtualized graphics?*
3: *What inhibits reverse execution of paravirtualized graphics?*

Chapter 3

Background and Related Work

System simulators are abundant and exist in corporate [9], academic [30], and open-source variations [1]. Such platforms, like Simics, have been used for a variety of purposes including, but not limited to, thermal control strategies in multicores [6], networking timing analysis [27], web server performance evaluation [35], and to simulate costly hardware financially unfeasible to researchers [4]. Furthermore, such simulators may also be used to port Operating System (OS)s to new processors [61].

For the purposes of graphics acceleration, strategies, methodologies and procedures are numerous. Based off a number of core strategies, such as device modeling, passthrough technologies, and paravirtualization, there have been numerous attempts at effective GPU virtualization; many of which require modification of both target- and host systems (such as the development of specialized passthrough drivers [19]). One such instance is presented by Hansen in his work on the Blink display system [14].

Neither are the concepts of advanced simulatory features new to GPU virtualization, as there have are multiple attempts to implement the Dvtglosscheckpointrestart Checkpoint / Restart-model in a GPU context, such as the work by Guo et al. concerning the CUDA framework [13]. Another solution which also supports the dvtglosscheckpointrestart Checkpoint / Restart-scheme is VMGL, as presented by Lagar-Cavilla et al., which by the means of paravirtualization accelerates the OpenGL 1.5 framework [19]. The groundwork produced by Lagar-Cavilla et al. showcases the potential for paravirtualized graphics since the VMGL framework, for a certain set of benchmarks, attains improvements of roughly two orders of magnitude in relation to software rasterization.

Current promising projects surrounding GPU virtualization include the Virgil3D-project [42]. As described at the project homepage[1], the project strives to create a virtual GPU which may subsequently utilize host hardware to accelerate 3D rendering. The project is currently being maintained, again according to the project's GitHub homepage, by Red Hat's Dave Airlie. Other related works include modeling GPU devices in the QEMU full-system simulator with software OpenGL ES rasterization support, as presented by Shen et al. [32].

[1] www.virgil3d.github.io

At the time of writing, graphics virtualization is no longer limited to the academic community but is also existent in the industry, as big virtualization players incorporate various graphics acceleration solutions in their products. One such example is VMware, Inc. [62].

Pursuant to the aim and objectives specified in chapter 2, this thesis pertain to the technologies and concepts described in this chapter.

3.1 OpenGL ES

OpenGL ES is an API for 3D-graphics on an assortment of embedded systems, such as mobile devices or vehicle displays [25]. The OpenGL ES set of APIs is developed by Khronos, the same consortium responsible for the development of the OpenGL APIs, which - unlike OpenGL for embedded systems - is intended for desktop graphics [36]. The OpenGL ES APIs are traditionally derived from the standard OpenGL APIs, and are thus similar in appearance albeit more limited [36]. The OpenGL ES specification is intended to adhere to the following specifications, in relation to the original OpenGL APIs [25]:

- Reduce complexity, but attempt maintain compatibility with OpenGL when possible.
- Optimize power consumption for embedded devices.
- Incorporate a set of quality specifiers into the OpenGL ES specification. This includes minimum quality specifiers for image quality into the standard; accommodating for limited screen sizes in embedded systems.

OpenGL ES 2.0 consists of two Khronos specifications; being the OpenGL ES 2.0 API specification and the OpenGL ES Shading Language Specification [25]. Being derived from the OpenGL 2.0 specification [36], OpenGL ES 2.0 supports programmable shaders and is no longer encumbered by the fixed functionality that characterized earlier versions of the OpenGL and OpenGL ES APIs [25]; making the API a modern contestant amongst a variety of applications on a range of platforms, such as the Android- and iOS operating systems [36], popular on modern smartphones. The OpenGL ES 2.0 API was selected for use, for the purpose of this study, due to its programmable shaders and reduced function set, in addition to accelerated support for the API in the Android emulator.

3.2 GPU Architecture

GPUs are massively parallel numeric computing processors often used for 3D graphics acceleration. While CPUs are designed to maximize sequential performance, GPUs are designed to maximize Floating-point Operations Per Second (FLOPS) throughput; originally required for the floating point linear algebra often needed by 3D graphics. Additionally, the continuous need for faster processing

units, slowed down by the heat dissipation issues limiting clock frequency since 2003 [17], has driven the utilization of GPUs for general purpose workloads.

Generally, CPUs are designed to optimize the execution of an individual thread. In order to facilitate sequential performance, the processing pipeline is designed for low-latency operations; including thorough caching methodologies and low-latency arithmetic units. This design philosophy is often referred to as latency-oriented design, since it strives for low-latency operations to accommodate high performance of individual threads [17].

Meanwhile, GPUs are designed to maximize the the throughput of a large number of threads; having less regard for the performance of individual threads. As such, GPUs do not prioritize caches or other low-latency optimizations like CPUs, but aim to conceal overhead induced by memory references or arithmetic by the execution of many threads, which may perform in place during arithmetic- or memory operations. Such design is often referred to as throughput-oriented design [17].

The differing design philosophies of CPU and GPU units are so fundamentally different that they are largely incompatible in terms their workloads, as explained by Kirk and Hwu [17]. As such, programs that feature few threads, CPUs with lower operation latencies may perform better than GPUs; whereas a program with a large number of threads may fully utilize the higher execution throughput of GPUs [17].

The introduction of a number of CPU-oriented optimizations for simulating GPU-bound workloads on CPUs has made it possible to simulate such kernels several times faster than traditional simulation. Although orders of magnitudes slower, such optimizations may ease the simulation of GPU workloads on CPUs, as indicative by some studies [26]. However, without hardware-assisted virtualization, the execution of GPU workloads on CPUs quickly becomes unfeasible; inducing the need of more advanced graphics simulation methodologies.

3.3 Graphics Virtualization

There are a number of ways of virtualizing GPUs in system simulators, a few of which accommodate for hardware acceleration of GPU kernels. When faced with tackling the issue of GPU virtualization, there are equally many variables to consider as there are options; the first of which is the purpose of said virtualization. The Simics architectural simulator is by all means a full-system simulator; meaning, as portrayed in chapter 4, that it may run real-software stacks without modification. However, Simics is intended to feature low level timing fidelity for the purposes of high performance, and is - as such - not a cycle-accurate simulator. In this way, and in line with the considerations for GPU virtualization, one must analyze and balance the purpose of simulation since there is not always a general best-case. As such, methodologies with varying levels of simulatory accu-

racy present themselves - from slow low-level instruction set modeling to fast high level paravirtualization of an assortment of graphics frameworks. Summaries of these methodologies are presented in this section.

GPU Modeling One may consider developing a full-fletched GPU model; that is, virtualizing the GPU Instruction Set Architecture (ISA). This methodology may be appropriate for the purposes of low-level development close to GPU hardware. For example, one might imagine the scenario of driver development for next-generation GPUs.

However, the development of GPU models, similar to that of common architectural model development for the Simics full-system simulator, incurs a number of flaws. The first of these flaws encompasses estimated development costs reaching unsustainable levels, due to GPU hardware often being poorly documented [19] on the contrary to CPU architectures. Furthermore, the modeling of massively parallelized GPU technology on CPUs induce high costs rendering the methodology less preferable for development requiring high application speed.

PCI Passthrough Furthermore, one ought to examine the benefits of PCI passthrough; allowing virtual systems first-hand access to host machine devices [54]. The direct contact with host system devices accommodated by methodologies such as PCI passthrough enable fully-fledged hardware accelerated graphics. Yet the methodology suffers from several disadvantages, such as requiring dedicated hardware, causing the host system to lose access to devices during the course of simulation. In terms of GPU virtualization, this would induce the necessity of the host machine featuring multiple graphics cards. Additionally, and mayhaps the biggest flaw of passthrough methodologies, is the requirement of modifying the simulation target to utilize host hardware; effectively restricting what systems may be simulated. This restriction encompasses, inter alia, the utilization of corresponding device drivers to the host system, rendering the methodology inflexible in terms of GPU virtualization diversity. As such, and in line with a paravirtualized approach, PCI passthrough requires modification of the target system - in addition to configuration of the simulation host.

Soft Modeling As an alternative to precise modeling of GPU devices, one might analyze the feasibility of high-speed software rasterization. Albeit not up to hardware accelerated speeds, some results indicate an increased feasibility of high-speed software rasterization in modern graphics frameworks (see [26]), where traditional software rasterization is accelerated using thread pooling optimizations and Single Instruction, Multiple Data (SIMD) technologies [55]; all for the purposes of optimizing execution for CPU-, rather than GPU-, execution. As such, one may avoid some of the overhead induced by simulating GPU workload on CPUs, which is traditionally not fit for purpose. One might speculate

that using such technologies in collaboration with hardware-assisted virtualization might bring software rasterization up to competitive speeds fit for some simulatory development purposes, replacing the need for more sophisticated virtualization techniques. However, without native execution speeds or used with complex GPU workloads, the technique may fall short.

Paravirtualization At a higher level of abstraction, there is the option of virtualization by paravirtualization. By selectively modifying target systems, one may modify the inner workings of system attributes and add functionality such as graphics hardware support. For the purposes of graphics acceleration, such a system attribute may be a graphics library or a kernel driver.

Often, paravirtualized methodologies incurs the benefits of host hardware acceleration of some graphics framework, and is implemented at a relatively high abstraction level (see figure 5.2). Inherent from its higher abstraction, paravirtualization may be relatively cost-effective to implement in comparison to alternatives such as GPU modeling. Additionally, virtualizing at the graphics library software level circumvents the need for users to re-link or modify the application they wish accelerated. Furthermore, the serialization of framework invocations by the means of fast communications channels may accommodate for significant performance improvements when compared to that of networking solutions (see section 5.5).

However, despite the possibility for significant performance improvements, graphics virtualization by the means of paravirtualization is not without inherent flaws. In particular, a paravirtualized graphics library may be expensive to maintain as frameworks evolve and specifications change. Additionally, the means of paravirtualization requires the target system to be modified; albeit not necessarily being a substantial defect as a paravirtualized framework may still accelerate unmodified target applications utilizing the library. In this way, paravirtualization may be considered to be a decent leveling of the benefits and drawbacks of the various virtualization methodologies presented in this section.

3.4 QEMU

QEMU[2] is an open-source virtual platform described as a full system emulator [7] and a high-speed functional simulator [32] (see [1] for an overview of QEMU). It supports simulation of several common system architectures and hardware devices and can, like Simics, save and restore the state of a simulation [7]. As such, QEMU may, like Simics, run unmodified target software such as OSs, drivers, and other applications. The platform is widely used in academia, and is the subject of several articles and reports cited throughout this document, such as

[2] 'Quick Emulator'.

the graphics acceleration described by Lagar-Cavilla et al. [19], and the work by Guo et al. [13]. Additionally, QEMU powers the Android emulator, which helps mobile developers bring about software for the Android OS.

The Android emulator is described as a virtual mobile device emulator [48]. Included in the Android SDK, it supports virtualization of an assortment of mobile hardware configurations. In the presence of the Android 4.0.3 release, the Android SDK was updated to make use of hardware-assisted x86 virtualization; significantly increasing the performance of CPU-bound workloads for x86 systems [44]. In addition to this, Google implemented hardware support for the OpenGL ES 1.1- and 2.0 frameworks; granting developers utilizing said frameworks hardware acceleration of graphics [44]. Google's solution (see [49]), consists of a paravirtualized implementation which circumvents the simulation by forwarding its OpenGL ES invocations to the host system by using networking sockets or directly via the simulator program[3]. As of Android 4.4, the Android emulator uses QEMU to simulate ARM and x86[4] devices aiding those wishing to develop software for mobile units.

3.5 Magic Instructions

Sometimes during system simulation, there may be reasons as to why one would like to escape the simulation and resume execution in the real world. Such a scenario would be a debugging breakpoint, to share data in-between target and host systems, or for any reason modify the simulation state. There are a number of ways to communicate with the outside world (including the host machine) from within the simulation, such as by networking means or specially devised kernel drivers, but few are as instant as the - arguably - legitimately coined 'magic instruction'.

The magic instruction is a concept used to denote a `nop`-type instruction, meaning an instruction that would have no effect if run on the target architecture (such as `xchg ebx, ebx`[5] on the x86-architecture), which - when executed on the simulated hardware in a virtual platform - invokes a callback-method in the simulation host [21]. An advantage of this methodology is an often negligible invocation cost, as the context switch is often instant from the perspective of the target system [28]. Furthermore, being a greatly desirable attribute, magic instructions require no modification of the target system. Another advantage of the magic instruction paradigm is that the system invoking such an instruction may, without complications, run outside of a simulation - as this would simply

[3]Note, however, that there is no software rasterized solution for running OpenGL ES 2.0 in the Android emulator.

[4]By using images devised by Intel®, the Android emulator may run the Android OS on x86 simulated hardware (see section 6.1).

[5]'Swap contents in registers `ebx` and `ebx`'.

result in regular `nop`-behavior. In effect, implementation of magic instructions requires replacing one- or more instructions in the target instruction set; thereby making the magic instruction platform-dependent. However, the solution is often designed to only respond to magic instructions wherein a certain magic number, sometimes called a 'leaf number' [28], is present in an arbitrary processor register.

3.6 Virtual Time

In terms of system simulation, time often becomes abstract; since it is not necessarily the same for an observer outside of the simulation as that of an observer from the inside. The variance in virtual time, as compared to that of real-world time, is called 'simulation slowdown' and may reach orders of magnitude faster than that of real-world time, or likewise orders of magnitude slower.

The concepts of real-world and virtual time are particularly important when considering performance measurements. When attempting to establish some sort of measurement in a full-system simulator, such as Simics, one must contemplate what type of time is relevant to the study being performed. For graphics acceleration of real-time applications, it is likely that the real-world wall clock is the primary point of reference (see section 7.1 for an elaboration on how time measurement is performed for the sake of this study). However, there are cases in which virtual time is worthwhile to profile, such as the performance of virtual system drivers.

Chapter 4

Simics

Simics is a full-system simulator developed by Intel® and sold through Intel®s subsidiary Wind River Systems, Inc. Simics was originally developed by the simulation group at the Swedish Institute of Computer Science (SICS)[1]; the members of which founded Virtutech[2] and commercially launched the product in 1998 [23].

As an architectural simulator, Simics primary client group is software- and systems developers that produce an assortment of software for complex systems involving software and hardware interaction [2]. As such, key attributes of Simics are scalability, repeatability, and high-performance simulation. For these purposes, the simulator utilizes hardware-assisted virtualization, and other performance boosting technologies such as Hypersimulation [21]. Simics also features a number of advanced functionalities, adhering to the deterministic nature of the simulator, such as checkpointing (see section Checkpointing) and reverse execution (see section Reverse Execution) [21].

The ability to simulate the entirety of an unmodified software stack has led to Simics being used to simulate a variety of systems including, but not limited to, single-processor embedded boards, multiprocessor servers, and heterogeneous telecom clusters [2]. Current employers of the Simics full-system simulator include IBM [18], NASA [63][57], and Intel® [34]. Other past and current employers of the simulator include Sun Microsystems, Inc., Ericsson, and Hewlett-Packard Company [23], in addition to Cisco Systems, Inc., Freescale Semiconductor, Inc., GE Aviation, Honeywell International, Inc., Lockheed Martin, Nortel Networks Corporation and Northrop Grumman Corporation [56]. Additionally, the simulator has a strong academic tradition; being known to operate in over 300 universities throughout the world [35].

4.1 Deterministic Execution

'Deterministic execution' commonly refers to the execution of Deterministic Algorithms; meaning that a certain function, given a definite input, will produce a

[1] This being the first instance of an academic group running an unmodified OS in an entirely simulated environment

[2] Virtutech was acquired by Intel® in 2010 [56].

decisive output - throughout the process in which the system passes through a distinct set of states (see [10] for an overview on deterministic and non-deterministic algorithms). Some sources have voiced concerns over the consequences, in terms of debugging, of non-deterministic behavior caused by concurrent software [20][16]. Some even go as far as to argue that determinism is a prerequisite for effective debugging and testing [11][39]. As such, deterministic behavior may be seen as a valuable attribute in a virtual platform (see [8] for an overview on the value of determinism in software development).

In Simics, 'deterministic execution' denotes the entire target system as a deterministic algorithm, wherein all instructions are executed in a deterministic manner on the simulated hardware [2]. This means that the simulated system (e.g., an OS), presuming the same input, will allocate the same memory space in virtual memories, receive the same number of interrupts in sequence, and even inhabit the same registers in virtual CPUs [2] (see [45] for a brief rundown of deterministic execution in Simics). As such, given an arbitrary number of instructions, the simulation state may be recreated indiscriminately down to the level of the instruction set and corresponding cycles. Thus, throughout this document and in relation to the Simics full-system simulator, deterministic execution refers to the deterministic state transition of a target system; as described in this section.

4.2 Checkpointing

The state of a computer system may be defined as the entirety of its stored information, or memory, at a given time [15]. It may be beneficial to store such states in a 'dvtglosscheckpointrestartCheckpoint / Restart' scheme, as suggested by Jiang et al. in regards to CUDA kernels [40]. In this way, developers may save- and restore the state of a system which can reduce overhead of restarting computationally heavy applications from scratch [2].

In Simics, 'checkpointing' refers to the functionality to save the complete state of a simulation into a portable format. When applied, this format is known as a Checkpoint. This ability not only saves time in terms of program initialization and debugging [24], but may also ease testing and collaboration in-between developers as checkpoints may be distributed and shared [2] (see [47] for an overview on checkpointing in Simics). As such, throughout this document and in relation to the Simics full-system simulator, checkpointing denotes this functionality.

4.3 Reverse Execution

'Reverse execution' provides software developers with the ability to return, often from portable checkpoints, to previous states of execution [3]. This may be useful

when debugging, profiling, or testing as difficult-to-reach system states may be stored and returned to, effectively bypassing program initialization and other hindrances [24].

In Simics, 'reverse execution' denotes said ability - spanning the entirety of the simulated system [21]. As such, simulated systems may be run in reverse; including virtualized hardware device states, disk contents and CPU registers [2] (see [46] for an overview on reverse execution in Simics); whilst still maintaining determinism in the simulated system. As such, throughout this document and in relation to the Simics full-system simulator, reverse execution refers to the functionality described in this section.

Chapter 5
Proposed Solution and Implementation

The implementation constitutes the realization of the paravirtualized technology described in this document. From the implementation, several concepts and phrases may be recognized and related to the Android Hardware OpenGL ES emulation design overview (see [49]). In accordance to the Android emulator, paravirtualized graphics acceleration is achieved by the means of three overall components; being the Target System Libraries, the Host System Libraries, and the communications channel aptly named the Simics Pipe. These components, along with elaboration on the methodologies accommodating them, are described in this chapter (see figure 5.2 for an overview of the components and their relationships).

5.1 OpenGL ABI Generation

For the purposes of ensuring scalability of the solution during development, a set of scripts automating the generation of library source code is used to compile the majority of the target- and host system libraries. As such, a large amount of the OpenGL function definitions encoded and decoded by the components described in sections 5.2 and 5.3 are produced by this tool. The functionality is implemented in a set of `Python` scripts that, from a number of specification/configuration files detailing function signatures and argument attributes, generate both headers and source files in `C`; thus collocating the target OpenGL and EGL Application Binary Interface (ABI)s, along with the corresponding host decoding libraries. The tool generates all but the methods that need special treatment (due to state saving) and may thus generate methods with return values and inout arguments. See figure 5.1 for an overview of the code generation process.

5.2 Target System Libraries

The target system libraries are compiled from `C` and `C++`, and comprise implementations of the EGL- and OpenGL APIs. Due to the tight coupling in-between OpenGL and the platform windowing system, the solution must also accelerate

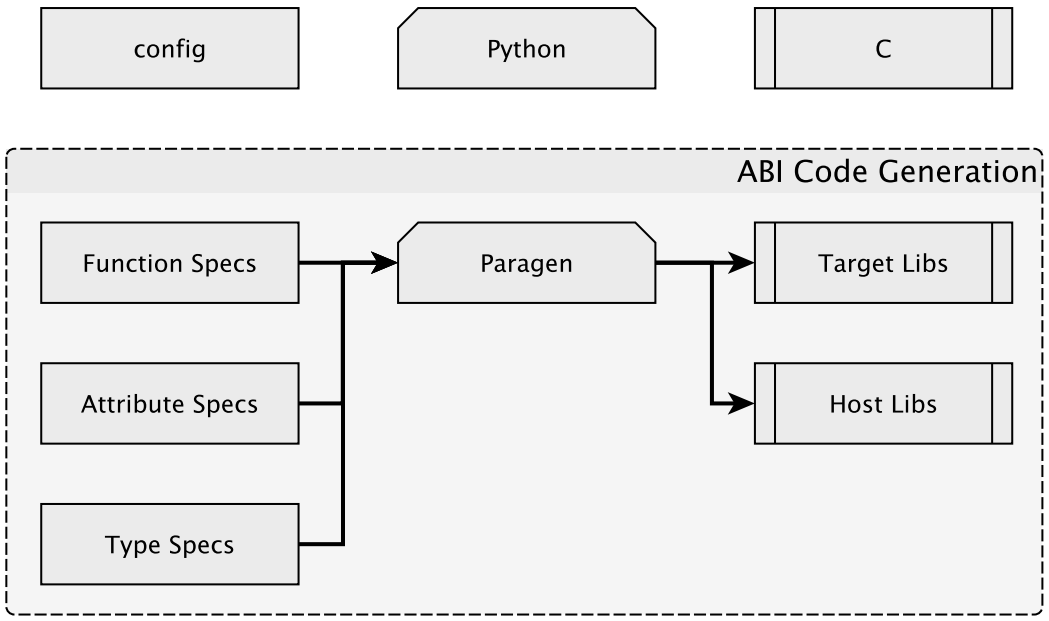

Figure 5.1: Visualization of the ABI code generating process performed to compile the target- and host system libraries.

the Khronos EGL API - the interface between OpenGL and the underlying platform windowing system (see section 5.4 for an elaboration on the full extent of EGL interaction). As may be derived from the name, the target system libraries run on the simulation target system and replace the existing Khronos libraries.

As such, the target systems libraries implement the EGL- and OpenGL ES 2.0 APIs and lures whatever application it is being linked to that it is, in fact, the expected platform libraries. Given that the target system libraries adheres to the OpenGL headers defined in the system, the application is naïve in terms of it's paravirtualized status. The interplay with the original OpenGL ES headers also results in the solution adhering to the platform-dependent type definition, flags, and constants; as originally defined by Khronos. However, instead of communicating with the platform windowing system (in terms of EGL) and the graphics device (in terms of OpenGL) - and instructing said device in coagency with the user; the target system libraries rather serialize the given command stream and forwards it to the simulation host. However, the transmission of the command stream is not necessarily performed at once, or in the designated order due to the formation of the OpenGL ES 2.0 framework. This complex of problems involve uncertainties of the proportion of argument data, as size is not necessarily specified by the user. As such, certain serialization may have to be delayed until further information surrounding the argument dimensions have been relayed to the OpenGL library. Furthermore, a subset of the OpenGL state need be

maintained by the target system libraries. These attributes are comprised by, inter alia, bound vertex- and index element buffers, in addition to properties of OpenGL vertex attributes. Such states must be kept in the target system libraries due to the asynchronous nature of serialization of OpenGL invocations in the paravirtualized solution.

The serialization described in the above paragraph is thus formatted and encoded in accordance to a certain data format, which is kept as minimal as possible throughout execution. This encoding includes packing variable length data types, such as 8-bit characters, 16-bit fields, or 64-bit integer values, into fixed length structures, so that the host system may interpret these values independently of how corresponding types are defined on that unrelated platform[1].

5.3 Host System Libraries

In collaboration with the target system libraries, the host system libraries - running on the host system - subsequently decodes and interprets the received byte stream. Said decoding involves unpacking data from fixed length storage into variable-size types that the OpenGL- and EGL libraries expect. Furthermore, and again similarly to the target system libraries, due to design inherent in the OpenGL ES 2.0 framework, the host system libraries need maintain some data; in particular - vertex attributes. Such data is buffered in the host system libraries until drawn in a later, and separate, OpenGL invocation[2]. When the requested OpenGL invocation has been performed, any return- or in-out values are returned to the target system using the Simics Pipe (see section 5.5). As with the the target system libraries, the receiving end of an OpenGL method definitions in the host system libraries are likewise generated to a large degree. The host system libraries are written in `C` and `C++`.

5.4 Windowing Systems

Due to variations in the creation and maintenance of windows on different platforms (i.e., Fedora 19 and Android), incurred by said platforms utilizing different interfaces for the purpose, the window to which OpenGL renders is kept on the simulation host. The problem may be summarized as the dilemma of the target system libraries having to communicate with the correct window (located in the simulation host), yet having the target system window reporting successful initialization. After all, it would be problematic if the OpenGL-utilizing application

[1] It should be noted, however, that the solution assumes a little endian architecture and IEEE 754 standard for floating point representation. If the host system would not conform to these prerequisites, the solution would have to be complemented with additional support.

[2] A possible optimization would be to cache said data, to avoid the need to transmit unmodified vertices multiple times, despite so being specified by the user.

would have to be modified in order to be paravirtualized. Effectively, this means that it would be desirable to maintain the native functionality of the target system EGL library. However, in order to swap the backbuffers to which OpenGL renders (in the window present in the host system), one must use EGL methods. In this way, the problem comprises a conflict in-between wanting to keep the functionality of the native EGL library, yet modify a small subset of it. This issue is overcome by the use of symbol overriding[3], which allows the target system libraries to overload a function, serialize and forward the invocation to the host system, locate the next occurrence of the symbol in the symbol table (being the original native EGL function definition), and invoke the original function. As such, the target system EGL library does not replace the native target EGL library, as with the OpenGL library, but rather overloads some of its definition. This gives the effect of a successfully created window, not having returned any errors in the window creation and maintenance - yet having the application actually communicating with a different window (present in the host system). As such, the target system libraries are effectively performing a man-in-the-middle attack on the target EGL library.

In order to run the benchmarks described in section 6.2, the target system libraries need simply overload the `eglSwapBuffers` function, but the solution could easily be extended to listen in on requested window dimensions and other window attributes; effectively circumventing the need to heed any differences in-between platform-specific windowing systems.

5.5 Simics Pipe

The so named 'Simics Pipe' constitutes the target-to-host communications channel in Simics. As such, it is responsible for transferring a serialized command stream - or byte stream - to- and from the simulation host using magic instructions. The role of the Simics Pipe is comprised of the allocation and maintenance of page locked target memory. Furthermore, the Simics Pipe utilizes a magic instruction to relay the starting address of said memory space to the simulation host; where the responsibilities of the Simics Pipe end.

There are a numerous methodologies to communicate with the outside world from inside of the simulation, such as utilizing UNIX sockets or other networking technologies. However, due to the inherent performance demands brought on by serialization of real-time graphics invocations, the methodology of magic instructions were selected for use for the purpose of this study. Magic instructions allow for fast, in the pretext of the simulation target - almost instant - escape from the simulation context, and may carry a limited amount of information with it from inside the simulation out into the real world. However, other - albeit slower,

[3]Utilizing `LD_PRELOAD` on the Linux platform.

methodologies may be fast enough to for use in graphics paravirtualization; the Android emulator, in addition to magic instructions, supports TCP/IP communication[4].

In order to perform a magic instruction, the Simics Pipe uses `GCC` Extended Asm to directly inject `__volatile__`-flagged (as to prevent the compiler from reordering memory accesses during optimization) `assembly` instructions into `C` code, in which - in addition to `C++` - the Simics Pipe is written. During said instruction, one may simply write an arbitrary 64-bit address to any register fit for the purpose (the number- and size of processor registers being the data-sharing bottleneck of the magic instruction paradigm). Not being the first time magic instructions have been used for the purposes of hardware acceleration [21], the Simics Pipe utilizes such methodology, namely the `CPUID` x86 instruction, to carry a lone memory address (being the address of the serialized command stream) in the target `rbx`[5] register. The execution of the injected instruction in a simulated processor, in line with the magic instruction paradigm, invokes a callback method in the host side of the Simics Pipe; having effectively escaped the simulation and paused the simulation state. Knowing the occurrence of a magic instruction with the corresponding leaf number, one may assume the existence of a target memory address in the simulated target CPU register `rbx`. Note that, at this point, the retrieved address is in the format of a target system virtual address; the destination of which is unknown to the simulation host. As such, this address need be translated in order to retrieve the package contents said memory address points to. See section 5.6 for more information on how the address is interpreted and how the entirety of the intended sequence of bytes is retrieved from target system primary memory.

5.6 Page Table Traversal

As outlined in section 5.5, target-/host memory sharing in the Simics Pipe is performed by the means of exposing a target virtual address to the simulation host. Said virtual address is, understandably so, only valid in the simulated system and bears no relevance in the real world; outside of the fact that the data to which the virtual address refers - in the target system - is, in turn, hidden behind layers of abstraction somewhere in host system memory.

Using Simics to utilize the Memory Management Unit (MMU) of the target system, one may translate a target virtual address (or 'virtualized virtual address') to a (target) physical device address; to-/from which - again, using Simics - we may access an arbitrary number of bytes in physical memory. However, due

[4]It may be of interest to know that the Simics OpenGL ES paravirtualization technology, during development, was accommodated by a TCP/IP client-/server model.

[5]Accordingly, if the simulation target should correspond to a 32-bit x86 system, said register would translate to the `ebx` register.

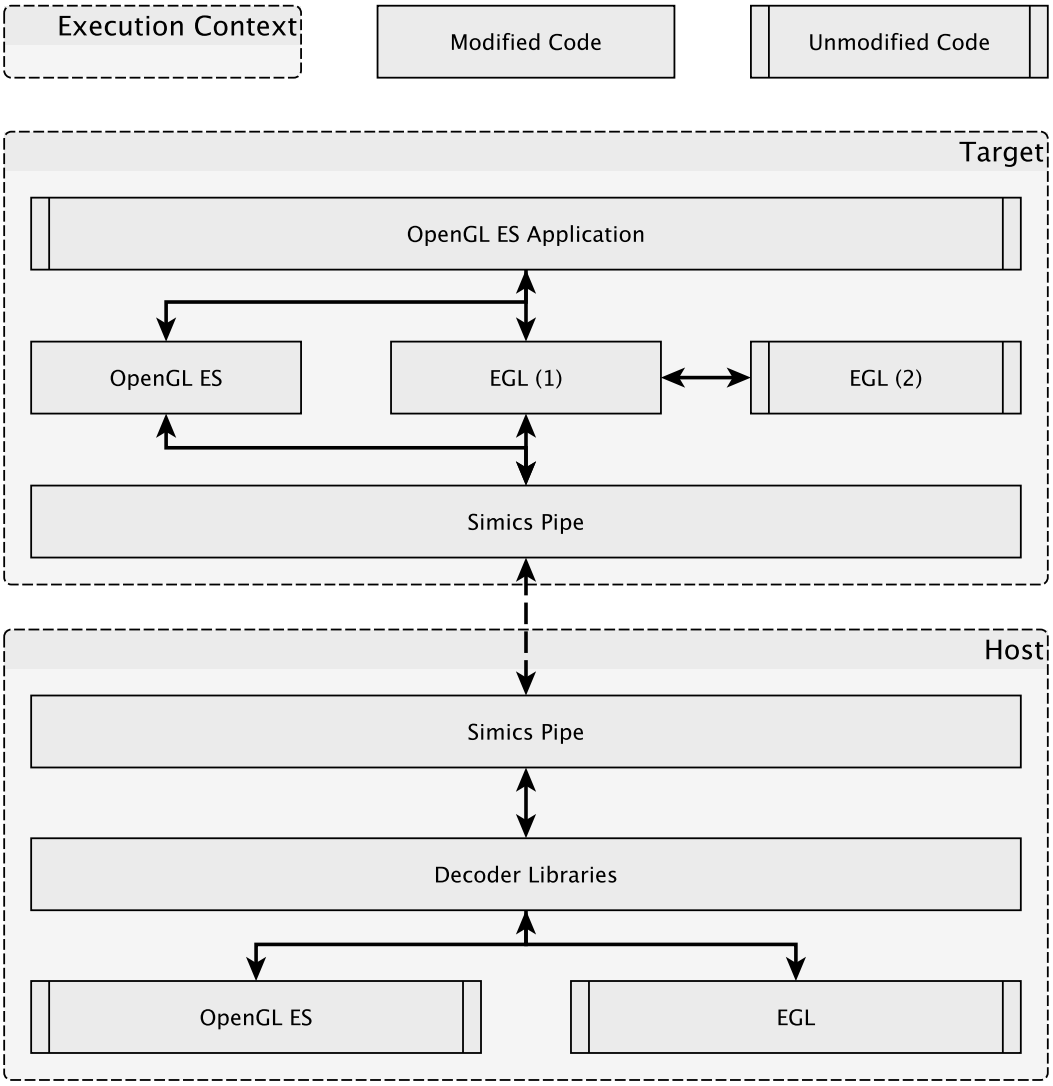

Figure 5.2: Overview of the implementation accomodating paravirtualized graphics in Simics.

to the complexity induced by circumventing the abstraction of virtual memory, there is no guarantee that the memory page to which the exposed physical address refers has not been swapped out of primary memory. In order to solve this, using some OSs (e.g., Fedora 19[6]), one may 'lock' pages in order to prevent them from being swapped to disk. This is the methodology used to ensure that the physical memory space, referred to by the virtual address sent to the simulation host, is still existent in target primary memory when the simulation state has been paused[7]. Furthermore, and again induced by the unorthodox circumvention of the virtual memory paradigm, it is probable that the multiple memory pages making out particularly large serialized command streams, such as the transmission of vertex or texture data, are not consecutively aligned in physical memory, although guaranteed to be continuous blocks in terms of virtual memory. As such, the physical addresses of memory pages must be continuously retrieved and translated on a per-page basis; effectively 'traversing' the virtual memory table (see figure 5.3). This can be done in a trivial manner by simply iterating the original virtual address with the *target* page size, in our case, 4096 bytes. Naturally, said process must be performed regardless of data being read or written to-/from the intended - physical - memory space.

[6]Using the `MAP_LOCKED` flag during invocation of the Linux `mmap`-system call; alternatively using the specific `mlock`-system call.

[7]Other methods to achieve this include repeatedly 'polling' the corresponding memory pages in the target system before inducing the simulation context switch to the host system.

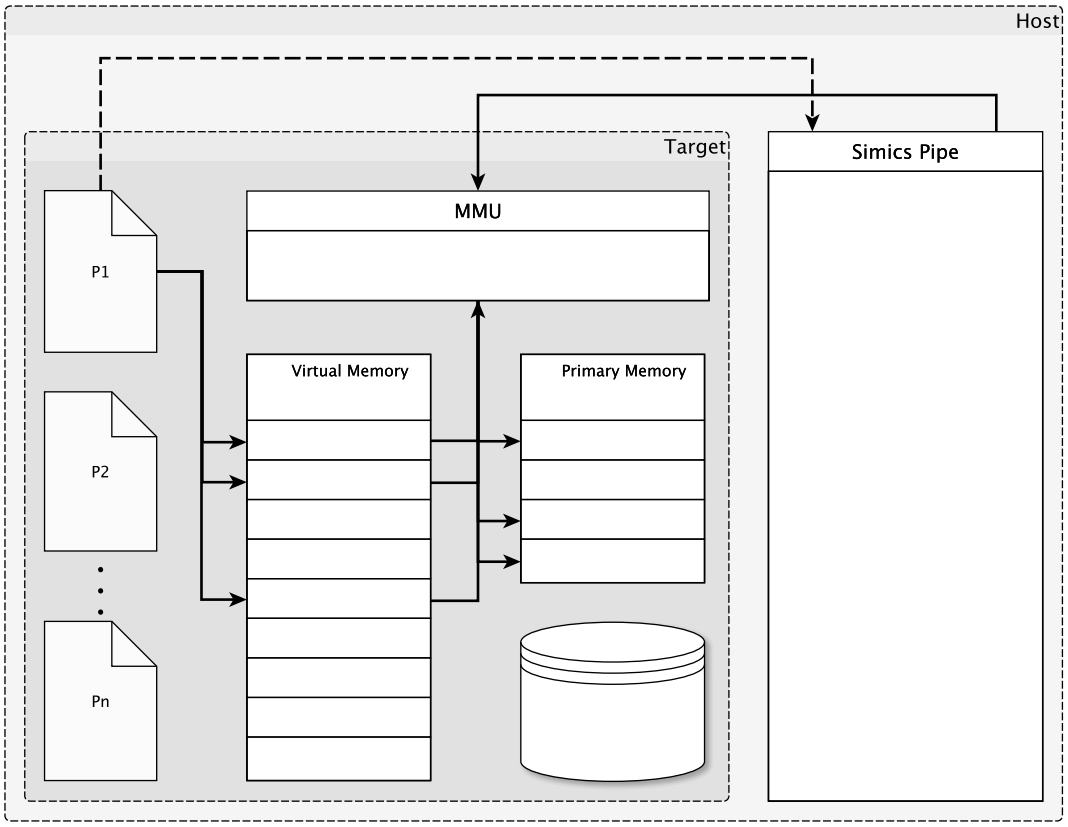

Figure 5.3: Memory translation overview. The OpenGL process hands a virtual memory address, pointing somewhere in the target system *primary* memory, to the paravirtualized solution - which inquiries the target system MMU to retrieve designated bytestream directly from target physical memory.

Chapter 6

Experimental Methodology

From this chapter onward, the term 'experiment' is used to refer to the experimental methodology put in place to answer the question formulations phrased in chapter 2. The author would like to emphasize that the terminology is simply used to denote a method of experimentation to answer, that is verify or refute the validity of the hypotheses that may be intrinsically derived from-, the research questions outlined in chapter 2. As such, 'experiment' does not refer to controlled-, natural-, or field experiment methodologies, or indeed any other specific method for experimentation unless specified otherwise. The author reserves the right to use the following terminology interchangeably: study, case study, and experiment.

As such, the terminology used to describe the experimental methodology produced for the purpose of this study, henceforth referred to as 'the experiment', is not meant to be indicative of the method quality, or in any way indirectly describe said experimental methodology. Below, the methods used to perform the experiment are presented.

6.1 Platform Configuration

The experiment devised for the purpose of this dissertation is performed in software rasterized- and paravirtualized Simics platforms, respectively. Furthermore, the experiment profiles the performance of the Android emulator to relate the devised solution to pre-existing technologies, in addition to execution on the hardware accelerated host platform as a reference case. The following paragraphs outline the configuration of these target platforms, in terms of host configuration, simulated hardware, and the software configuration of the simulated systems.

Host Configuration The system upon which the experiment is performed is an x86-compatible Haswell Intel® processor configured to run the Fedora 19 Linux distribution. Said OS was selected for use due to Fedora 19 being the primary platform used for development of the Simics full-system simulator at the Intel® offices in Stockholm - at which the solution described in this document has been developed.

Target Hardware Configuration The Simics hardware configuration utilized for the purpose of this experiment simulates an Intel® Core™ i7; the same processor series to that of the actual hardware of the simulation host system. Furthermore, Simics execution, both the software rasterized and paravirtualized platforms, utilize Kernel-based Virtual Machine (KVM) for the simulation target to run natively on the host hardware. In order to accommodate for similar acceleration in QEMU, Android is configured to run on the Intel® Android x86 system image (see [52]), circumventing the need to interpret ARM instructions [60]. Furthermore, the solution also utilize KVM in order to provide similar native execution on the host hardware[1].

Target Software Configuration In line with Fedora 19 being the OS in use when performing the reference benchmark experiments on the host platform (see paragraph Host Configuration), it may be of value to have the simulated target machine configured to run with the same OS. As such, the platform OSs used for the purpose of this study are as follows:

- Hardware accelerated Fedora 19 host system
- Software rasterized[2] Fedora 19 Simics target system
- Paravirtualized Fedora 19 Simics target system
- Paravirtualized Android 4.4[3] QEMU target system

6.2 Benchmarks

Throughout the course of the pilot study, no existing OpenGL ES 2.0 benchmark (featuring cross-platform profiling support for Android and X11 Linux) was deemed appropriate for for the purposes of this experiment. As such, a number of benchmarks have been devised on-site for the purposes of stress-testing the paravirtualized technology described in this document. The benchmarks, numbering three in total, are intended to stress suspected bottlenecks in the implementation; corresponding to a large number of relatively insignificant OpenGL ES invocations, computationally intensive GPU kernels, and passing of large data such as textures or models. Since the benchmarks are required to run in both Linux- and Android platforms, the benchmarking suite utilize Java Native Interface (JNI) to invoke OpenGL ES from the same `C` code base independent of platform. Effectively, this entails having the benchmarks produced using `Java`, `C`, and `GLSL`, collectively. Furthermore, all benchmarks are configured to run at roughly 16 ms, which would correspond to roughly 60 Frames Per Second (FPS), when hardware

[1] Had the solution been running on the Windows OS, the solution would have utilized Intel® Hardware Accelerated Execution Manager (HAXM) (see [53]) to accommodate for said native execution [51].

[2] Using the Mesa OpenGL software rasterization implementation.

[3] Android KitKat - API level 19.

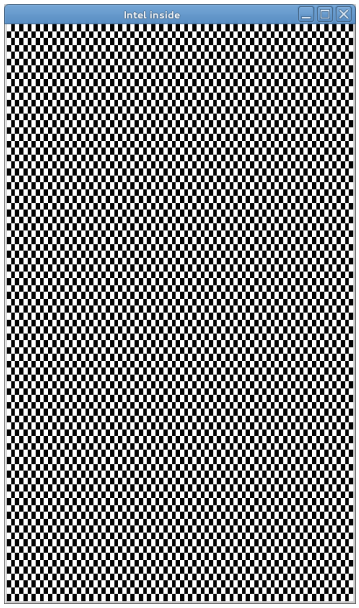

Figure 6.1: Chess.

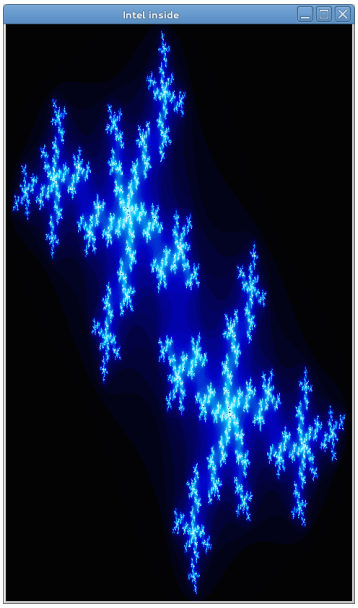

Figure 6.2: Julia.

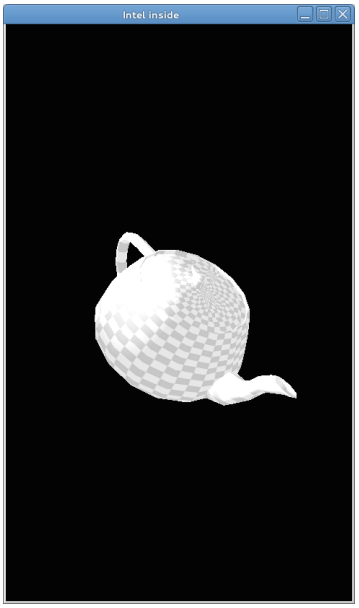

Figure 6.3: Phong.

accelerated on the host system. The benchmarks are devised in this way in order to reflect the expected load of a modern real-time interactive application. As such, the purpose of developed benchmarks is to be representative of typical scenarios induced by modern graphics applications whilst utilizing a graphics framework such as OpenGL. When run during the experiment, each benchmark instance measures the elapsed time of 1000 frames which makes up the data subsequently analyzed. Frame captures of the benchmarks are presented in figures 6.1, 6.2, and 6.3.

Benchmark: Chess The 'Chess' benchmark is developed for the purposes of stressing the latency in-between target- and host systems. It is so named because of the chess-like tileset the graphics kernel produces. The benchmark is designed to perform a multitude of OpenGL ES 2.0 library invocations per frame; in which each invocation is relatively lightweight in execution and carry a small amount of data argument-wise. In the Chess benchmark, this is achieved by rendering a grid of colored (black or white, in order to adhere to the chess paradigm) rectangles where each tile is represented by four two-dimensional vertices in screen-space, in addition to six indices outlining the rectangular shape. Since the vertices are already transformed into screen-space, the graphics kernel need perform no additional transformation, adhering to the desired lightweight behavior of each kernel invocation. Additionally, the tileset vertices and indices are pre-loaded into OpenGL vertex- and index element buffers, so that a lone buffer identifier may be carried over in-place of the heavier vertex set load. Each tile is then individually drawn to the backbuffer, rendering the chess-like appearance of the benchmark.

Effectively, this means that, for each tile, the benchmark need only bind a vertex- and an index element buffer, set the corresponding tile color, and lastly invoke the rendering of said tile.

For each frame rendered, depending on the number of drawn tiles, the solution will perform a large number of magic instructions. This induces a high utilization of the Simics Pipe, which is intended to stress magic instruction overhead (section 8.2). The repeated invocation of lesser draw calls is representative of common usage of drawing a multitude of shapes with OpenGL, such as a user interface. Additionally, the number of tiles being computed is easily modifiable; rendering the benchmark scalable for the purposes of the experiment described in this document. As such, said benchmark is considered suitable for the purpose of representing a large number of graphics invocations using OpenGL ES 2.0.

Benchmark: Julia The 'Julia' benchmark is developed for the purposes of stressing computational intensity in software-rasterized and paravirtualized platforms. It is so named due to the kernel calculating the Julia fractal; the texturing and frame-wise seeding of which gives the benchmark it's distinct look. The benchmark is designed to perform a lone computationally intensive graphics kernel invocation, which will stress the computational prowess of the profiled platform. The case is selected for use as the computation of a fractal is trivially scalable in terms of complexity, by modifying the number of iterations the fractal algorithm performs, and is thus considered suitable for profiling of computationally intensive graphics kernels.

Benchmark: Phong The 'Phong' benchmark is developed for the purposes of stressing the throughput, or bandwidth - if adhering to the networking paradigm, in-between target- and host systems. The Phong benchmark is comprised of a rotating model, being the Newell teapot, which is textured and subsequently shaded by a single point light using the Phong shading model; thus giving the benchmark it's name.

The rasterization and shading of a model with a given, large, texture is representative of three-dimensional graphics commonly rendered with graphics frameworks such as OpenGL ES 2.0. As such, said benchmark is suitable for the purposes of representing the usage of big data (being models, textures, etc.). For the purposes of stressing the bandwidth of target- and host communication, the Phong benchmark is easily scalable in terms of resizing the large texture in question. Vertices and indices of the model do not utilize OpenGL vertex- and-/or index element buffers, thus forcing the solution to transmit the data every frame. Additionally, in order to further stress the throughput of the profiled platform, texture data is updated every frame.

Benchmark	Key Fig.	Halved Fig.	Reference Fig.	Double Fig.
Chess	No. Tiles	60×60	84×84	118×118
Julia	No. Iterations	225	450	900
Phong	Texture Resolution	1448×1448	2048×2048	2896×2896

Table 6.1: Input data used for benchmark variations for each benchmark.

Benchmark	Halved Fig.	Reference Fig.	Double Fig.
Chess	$32400 + 3$	$63504 + 3$	$125316 + 3$
Julia	16	16	16
Phong	17	17	17

Table 6.2: Number of magic instructions performed per-frame for each benchmark input data variation.

6.3 Input Data Variation

In order to detect anything but linear potential of scaling in software rasterized- and paravirtualized Simics platforms, the benchmarks are configured to run in three separate instances, respectively. These benchmark versions differ in terms of input data; where said input is a changed variable that could potentially worsen benchmark performance (e.g., larger texture). The purpose of these 'input data variations' is to detect any unexpected results in execution; and thus identify any performance complexity issues in software rasterized- and paravirtualized Simics platforms. For consistency, each benchmark variation - of which there are two; resulting in three unique experiments per benchmark - have been produced to halve- and subsequently double the corresponding input data. The described input data, for each benchmark, is presented in table 6.1. Consequently, in table 6.2, the per-frame number of magic instructions induced by these input data variations are presented, differing only for the Chess benchmark.

Note that for each tile rendered in the Chess benchmark, the solution will perform 9 magic instructions; entailing a total of 32400[4], 63504[5], and 125316[6] magic instructions, respectively, in addition to a minor number per-frame.

[4] $9 \times 60 \times 60 = 32400$.
[5] $9 \times 84 \times 84 = 63504$.
[6] $9 \times 118 \times 118 = 125316$.

Chapter 7

Threats to Validity

7.1 Platform Profiling

In the presence of the complexities caused by the occurrence of virtual time, profiling of elapsed time in virtual platforms sometimes dictate special measures. This is due to the fact that profiling of elapsed time outside of the simulation - that is, time in the context of the observer - may be more relevant than profiling of virtual time. This is often the case if the simulation has outside dependencies of some sort. Naturally, such is the case in terms of real-time interaction and rendering - as such application is observed from outside of the simulation.

In a Simics simulation, as described by section 3.6, time increments differently than in the context of the host machine. As such, in order to appropriately measure the elapsed time of a benchmark frame, the profiling must take place outside of the simulation. As expanded upon in section 3.5, there are a number of ways to escape- and pause the state of a simulation. Mayhaps the most intelligible is to listen in on activity passing through a target serial port; being a traditional front-end to the machine. In this way, the simulator is instructed to listen in on, and set a simulation breakpoint at the occurrence of, a specialized sequence of bytes being written (via a serial console) to a Universal Asynchronous Reciever/Transmitter (UART) hardware serial port. In Linux, the interface to such a port may be accessed by any of the `/dev/ttyS*` file descriptors. This process is visualized in figure 7.1.

When using serial ports in this manner, one may introduce a profiling cost. This may be induced by - amongst other possible factors - the file descriptors, to which data is being written, not immediately sending said byte sequence via the system UART. In order to improve responsivity in time critical profiling of elapsed time, we ensure said serial console is flushed to by the means of performing the system call `tcdrain`. Furthermore, in order to ensure the accuracy of performed time profiling, the overhead cost of said method has been measured[1]. Collected profiling overhead data is presented in table 7.1.

From the data presented in table 7.1, we may deduce that the profiling overhead cost may occasionally be volatile. However, with a standard deviation of

[1]This has been performed by the means of invoking profiling start and stop consecutively.

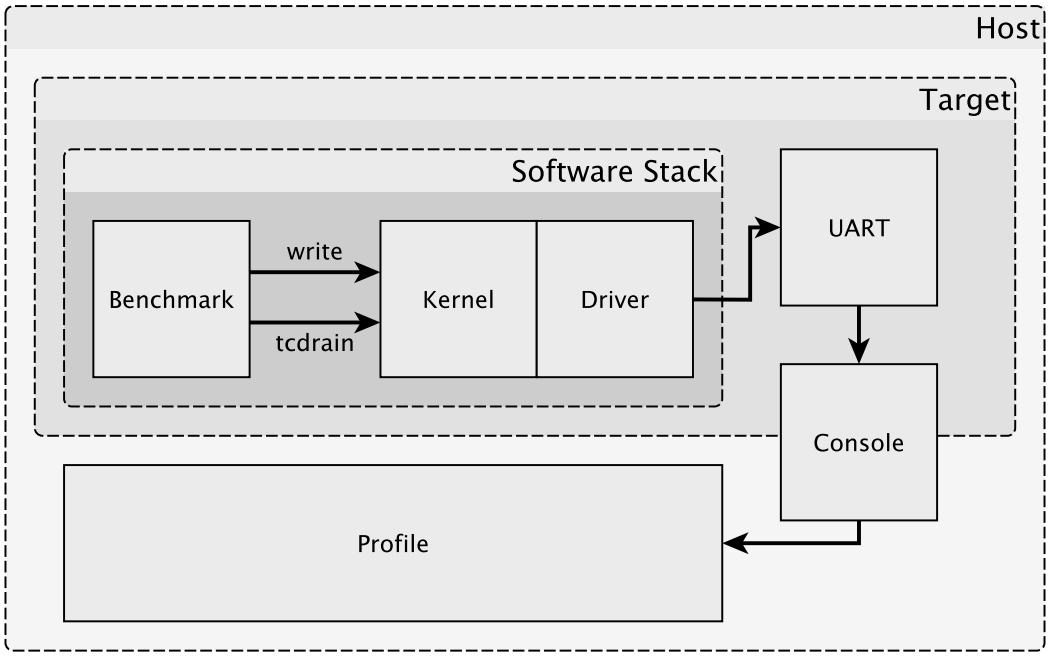

Figure 7.1: Overview of the frametime profiling process for a Simics simulation.

Min	Max	Std	Avg
0.62	32.50	1.54	1.56

Table 7.1: Collected results of the profiling overhead cost in milliseconds.

1.54 ms, slightly below that of the average, such large deviation is to be quite rare. If not specified otherwise, all results collected from the Simics platforms are subtracted with the average of this profiling cost; being 1.56 ms. Utilizing said methodology - when the simulator detects the chosen bit-pattern, it may start- or stop platform timing. This is the method used to measure the elapsed time of a frame during any benchmark running inside of a Simics simulation. As such, this method is used for the software rasterized- and paravirtualized Simics platforms.

Likewise, in a QEMU simulation, consideration must be taken to virtual time. However, in the QEMU-derived Android emulator, the system clock appears synchronized to that of the simulation host [50][41]. The theory is supported by findings presented in [58] and has been confirmed with tests profiling elapsed time before performing any experiments. It is probable that the Android emulator has been implemented in this way to accommodate for audio- and video playback in simulated Android applications. As such, in order to profile elapsed wall clock time when performing benchmarks test in the Android emulator, one may simply measure time elapsed inside of the simulation in accordance to the system clock. This is the methodology used to profile the QEMU platform.

7.2 Benchmark Variations

As visualized in figure 7.2, the Phong benchmark exhibits erratic variations in its performance when software rasterized in the Simics platform. From this visualization we may establish that the performance of the Phong benchmark varies greatly when software rasterized in Simics, independent of the Phong benchmark texture size.

It is uncertain as to what is causing this behavior. However, there may be cause to believe that texture mapping a model such as the one in the Phong benchmark, with a non-mipmapped texture of such a large texel count, may induce severe cache-misses due to the volatile texture mapping. Such a scenario may occur due to the large texture, in coagency with the frame-wise rotating model, causing the texture mapping outcome to be likely to differ on a per-frame basis. In consideration to the sheer size of the textures used for the experiments, it is likely that such eccentric memory referencing may account for the volatile performance of the Phong benchmark in the software rasterized Simics platform. Albeit not necessarily applicable to CPU cache behavior; see [12] for an elaboration on texture mapping and GPU cache methodologies in addition to an analysis on the performance implications of GPU cache behavior.

In accordance to the erratic benchmark behavior that has been established in this section, the Phong benchmark results - although having value to this study in terms of profiling memory bandwidth scalability - are to be considered less reliable than its more predictable benchmark peers.

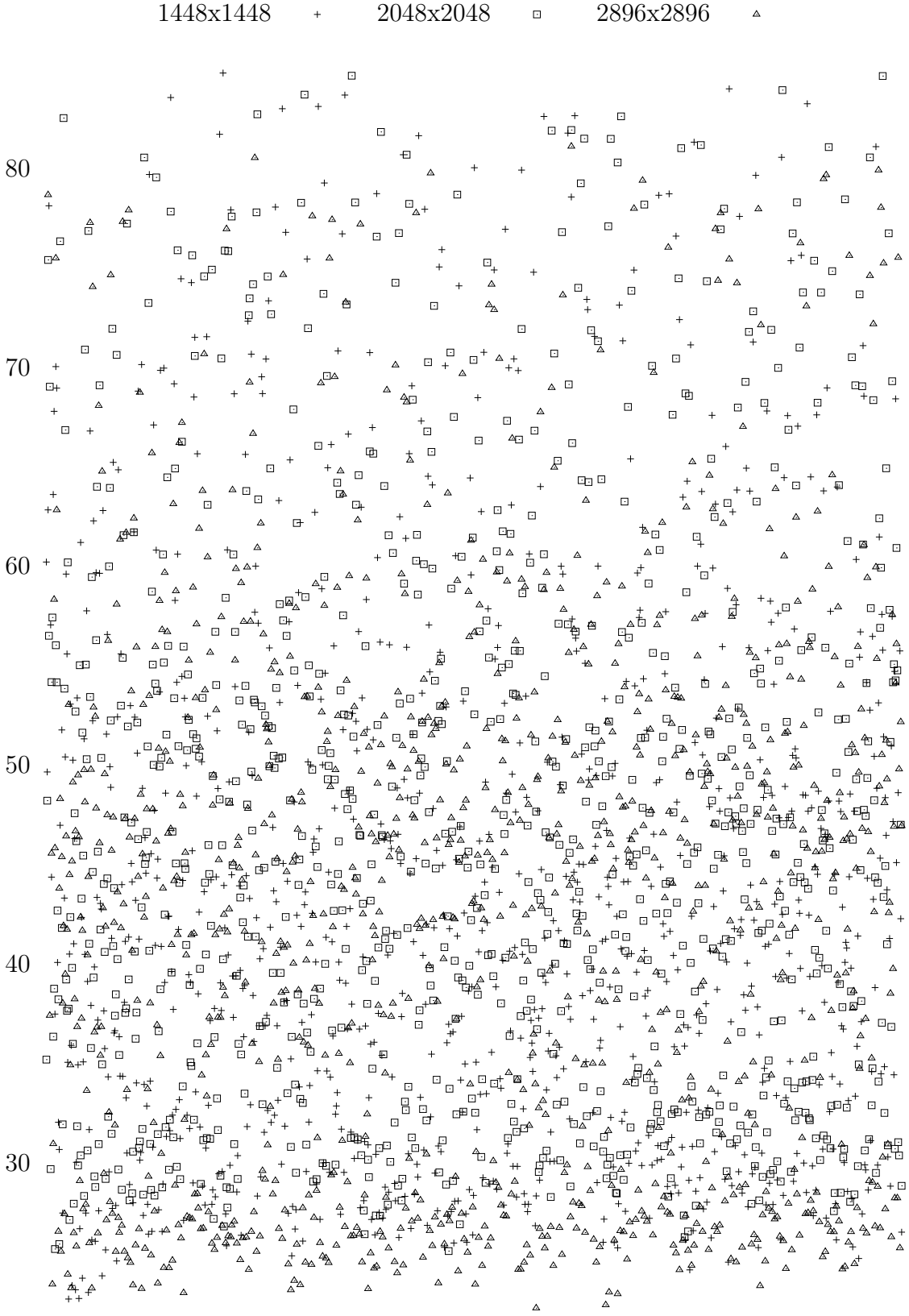

Figure 7.2: Scatterplot portraying frametime variation in milliseconds for the Phong benchmark. Each entry represents a frametime sample of wherein the three symbols represent different texture dimensions.

Chapter 8

Results

In this chapter, the results collected from having applied the described experiment methodology onto the devised solution are presented. As such, the results gathered from execution on the host is presented in figure 8.1, the results compiled from execution in the Android emulator presented in figure 8.2, and the results accumulated from software rasterized- and paravirtualized execution in Simics are presented in figures 8.3, 8.4, and 8.5. Note that the data compiled from software rasterized- and paravirtualized Simics platforms include additional data from having performed the experiment using the input data variations, as described in section 6.3.

In this chapter, the results are compiled into histograms; visualizing elapsed time in milliseconds to sample density. As such, the Y axis showcase sample density; although the axis keys have been removed as they bear little relevance to the outcomes presented in this material. The histograms each feature 100 bins; into which a 1000 samples, for each experiment performed, are rounded into. For the purposes of good visualization methodology, values outside of the standard deviation[1] are not featured in the figures presented in this section. In order to accommodate for the, however few, samples outside of said limits the figures are all complemented with key ratio tables (see tables 8.1, 8.2, 8.3, and 8.4).

8.1 Benchmark Results

Based on the reference profiling presented in figure 8.1, we may conclude that the benchmarks, when hardware accelerated on the host system, perform with concentrated density; not being much scattered across the graph except a few irregular extremities in terms of maximum frame times (see table 8.1). This is supported by the standard deviation presented in said table. Furthermore, we may conclude that the Phong demo features two distinct peaks in density distribution - about 1 ms in-between. There may be cause to believe that this is caused, or partly caused, by frame-wise rotation of the teapot - rotation featured in the benchmark (see section 6.2) - inducing some fluctuation into its execution

[1] That is; values above that of the $mean + std$ and values under that of $mean - std$.

Benchmark	Elapsed time (milliseconds)			
	Min	Max	Std	Avg
Chess	11.10	62.02	2.85	12.15
Julia	14.65	117.52	5.96	16.42
Phong	7.61	59.90	3.64	9.41

Table 8.1: Benchmarking results whilst hardware accelerated on the simulation host system.

Benchmark	Elapsed time (milliseconds)			
	Min	Max	Std	Avg
Chess	20.91	60.30	5.68	30.96
Julia	13.25	88.48	6.22	24.64
Phong	2.16	38.22	4.58	18.13

Table 8.2: Benchmarking results whilst paravirtualized in the QEMU-derived Android emulator.

(see figure 8.5). However, this behavior is, strangely so, not apparent whilst paravirtualized in the QEMU derived Android emulator, although this may be a visual artifact due to the resolution of the graph (see figure 8.2). See section 7.2 for an elaboration on divergence in the Phong benchmark. Additionally, and in accordance to tables 8.1 and 8.2, one may observe relatively high recorded maximum frame times in relation to compiled maximum- and average values, yet featuring - in relation to divergent maximum, relatively low standard deviations.

The remainder of this chapter will present a benchmark-Wise analysis of the data compiled from executing the experiment on the software rasterized- and paravirtualized Simics platforms; said platforms being the subject of this study. For the sake of brevity, these analyses are segmented into paragraphs for each benchmark. These paragraphs are presented below.

Chess From the data visualized in figure 8.3, we may observe that the Chess benchmark, when executed in the software rasterized Simics platform, has a relatively broad distribution of its sample density, yet the distribution often seems evenly distributed around a single point. The right-hand side of the graph, although also showcasing the impaired performance of the corresponding (paravirtualized) platform, visualizes a decrease in the distribution of the sample density. This is supported by the data presented in table 8.4.

Based on the data summarized in table 8.3 (whilst software rasterized in Simics) and comparing said data to that of table 8.4 (whilst paravirtualized in Simics), we may observe that the software rasterized solution outperforms it's paravirtualized counterpart; not only in the base experiment, but in all of its input data variations. The only redeeming attributes the paravirtualized solution

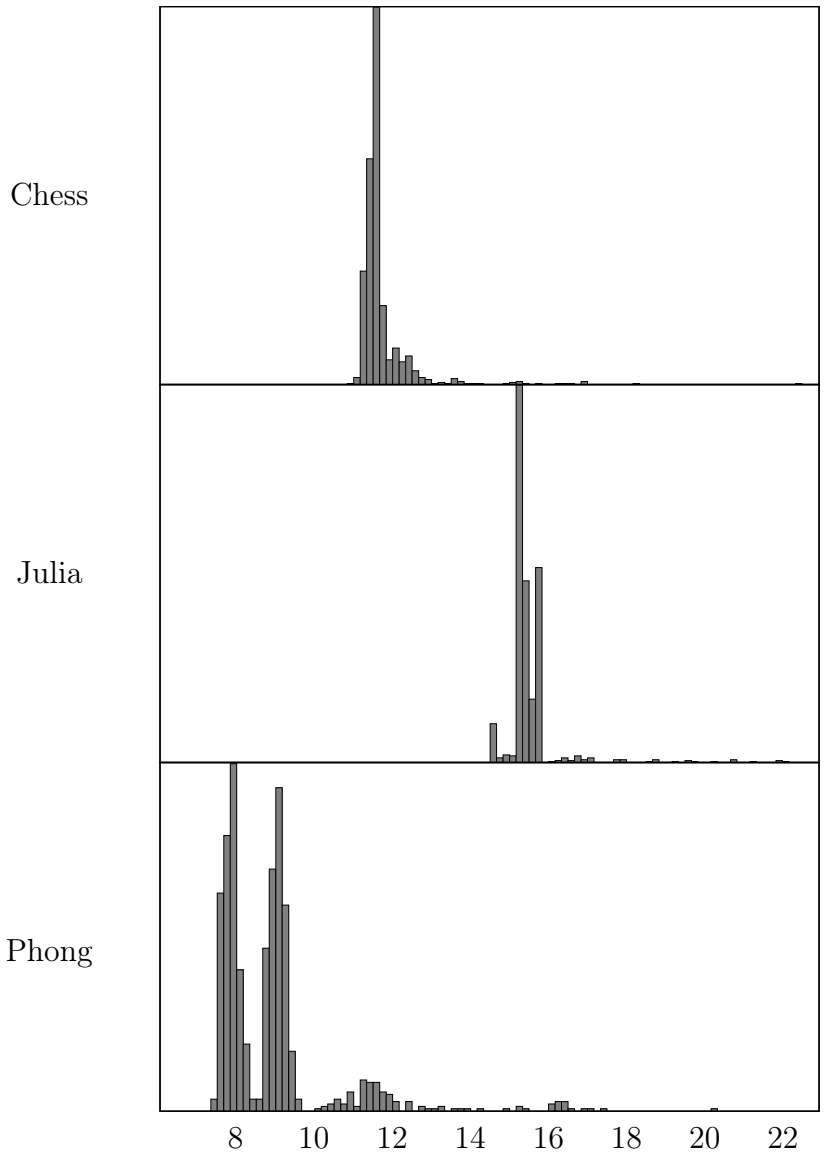

Figure 8.1: Histogram depicting benchmark elapsed frametimes in milliseconds and the density distribution of 1000 frames whilst hardware accelerated on the simulation host. The Y axis thus depict sample density. Its axis keys have been removed as they bear no relevance to the outcomes presented in this document.

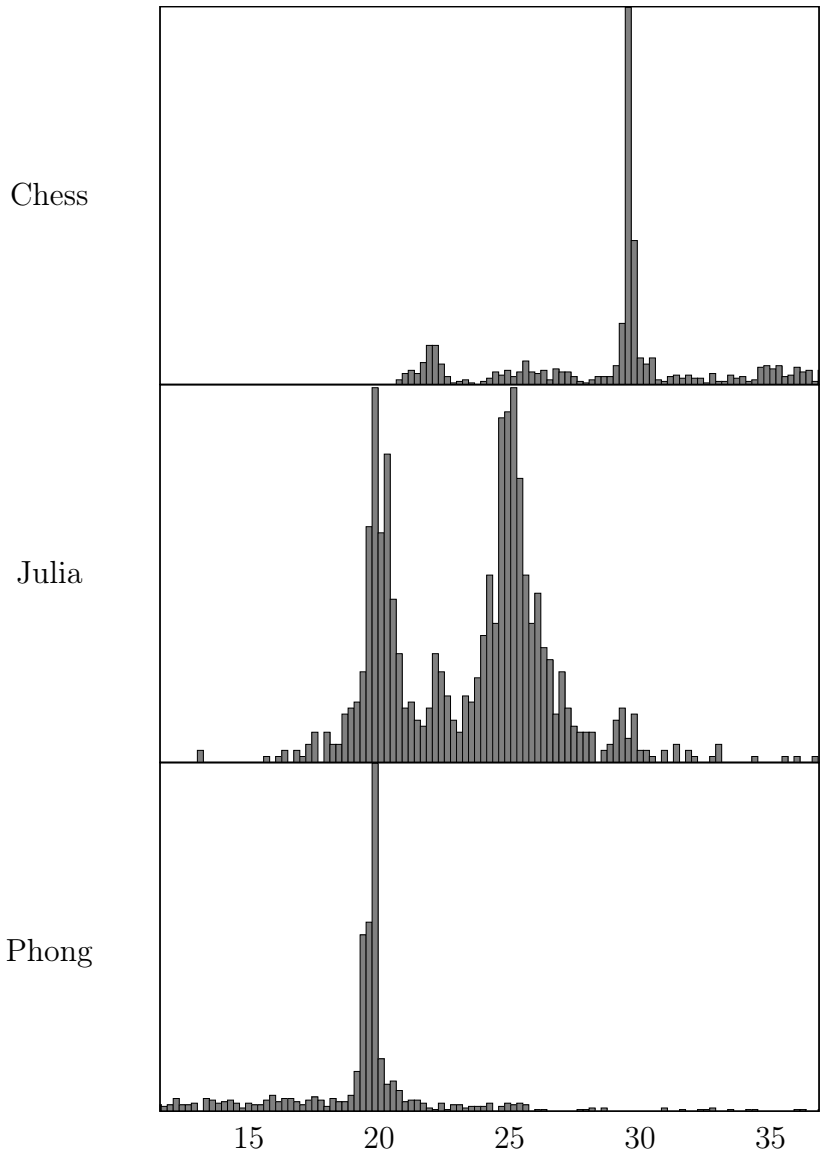

Figure 8.2: Histogram depicting benchmark elapsed frametimes in milliseconds and the density distribution of 1000 frames whilst paravirtualized in QEMU. The Y axis thus depict sample density. Its axis keys have been removed as they bear no relevance to the outcomes presented in this document.

Benchmark	Key	Elapsed time (milliseconds)			
		Min	Max	Std	Avg
Chess	60 × 60 tiles	76.88	439.27	19.48	114.41
	84 × 84 tiles	167.36	402.87	9.38	192.08
	118 × 118 tiles	238.86	701.16	17.63	259.54
Julia	225 iterations	397.82	2183.99	83.45	461.64
	450 iterations	744.91	2662.67	62.88	776.41
	900 iterations	1338.40	2669.19	113.87	1415.23
Phong	1448 × 1448 texels	17.06	926.13	34.26	40.43
	2048 × 2048 texels	17.83	545.43	21.21	35.04
	2896 × 2896 texels	28.39	729.39	38.57	66.48

Table 8.3: Benchmarking results whilst software rasterized in the Simics full-system simulator.

Benchmark	Key	Elapsed time (milliseconds)			
		Min	Max	Std	Avg
Chess	60 × 60 tiles	185.19	362.23	11.19	201.85
	84 × 84 tiles	309.08	401.26	15.51	336.83
	118 × 118 tiles	612.77	719.14	14.12	650.67
Julia	225 iterations	21.89	47.65	3.82	26.56
	450 iterations	38.79	101.56	10.01	49.83
	900 iterations	34.74	156.35	8.55	41.81
Phong	1448 × 1448 texels	15.20	54.19	7.95	23.05
	2048 × 2048 texels	26.53	125.31	6.38	36.09
	2896 × 2896 texels	50.35	154.17	9.80	64.38

Table 8.4: Benchmarking results whilst paravirtualized in the Simics full-system simulator.

brings to the table, as elaborated upon in the above paragraph, is a decrease in the standard deviation of the benchmark profiling. When comparing these results to the uncompromised hardware accelerated counterpart on the host machine (see figure 8.1), we may observe - albeit considerably less prominent - an adherence to the single-peak behavior in the distribution of the sample density.

The purpose of the Chess benchmark is to locate any bottlenecks related to the number of paravirtualized library invocations (see section 6.2), which was predicted during the pilot study performed for the sake of this experiment. As such, as presented in section 8.2 in combination with the shaping of the Chess benchmark as presented in section 6.2, there is cause to believe that the prediction of a probable bottleneck in the target- to host communication latency has been confirmed; arguably identifying the weakness of graphics paravirtualization in the Simics full-system simulator. The conclusions that may be drawn from these further stresses analysis into what is the root cause for the target- to host latency for a multitude of paravirtualized method invocations. Results related to this matter are presented in section 8.2. Indeed, if proceeding from the findings of section 8.2, magic instruction overhead accounts for the majority of the elapsed average when paravirtualized in Simics.

Julia In figure 8.4, we may observe double- to triple peak behavior in the distribution of the sample density; both in software rasterized- and paravirtualized platforms. Albeit the hardware accelerated host profiling (see figure 8.1) may, however minor, suggest such a pattern; it is by all means not significant. We may observe similar behavior in the distribution of the sample density when profiling the same benchmark whilst paravirtualized in the QEMU-derived Android emulator (see figure 8.2). What causes this behavior is unclear, as frame-to-frame branching in the fractal algorithm is minor and ought not cause such a variance.

The Julia benchmark is incorporated into the experiment to establish how the paravirtualized solution performed under computational stress (see section 6.2). Using this benchmark, a performance weakness in the software rasterized Simics platform has been identified, with frame times well above the two second mark (see table 8.3); with the corresponding maximum frame time in the paravirtualized Simics platform measuring up to to a mere 156.35 ms. With the Julia benchmark, as visualized in figure 8.4, we have showcased radical performance improvements for computationally intensive graphics kernels and - in turn - identified the capabilities of graphics paravirtualization in the Simics full-system simulator.

Phong *DISCLAIMER: The Phong benchmark signals strange behavior when software rasterized in Simics, and is thus considered less reliable (see section 7.2).*

The Phong benchmark is incorporated into this study for the purposes of an-

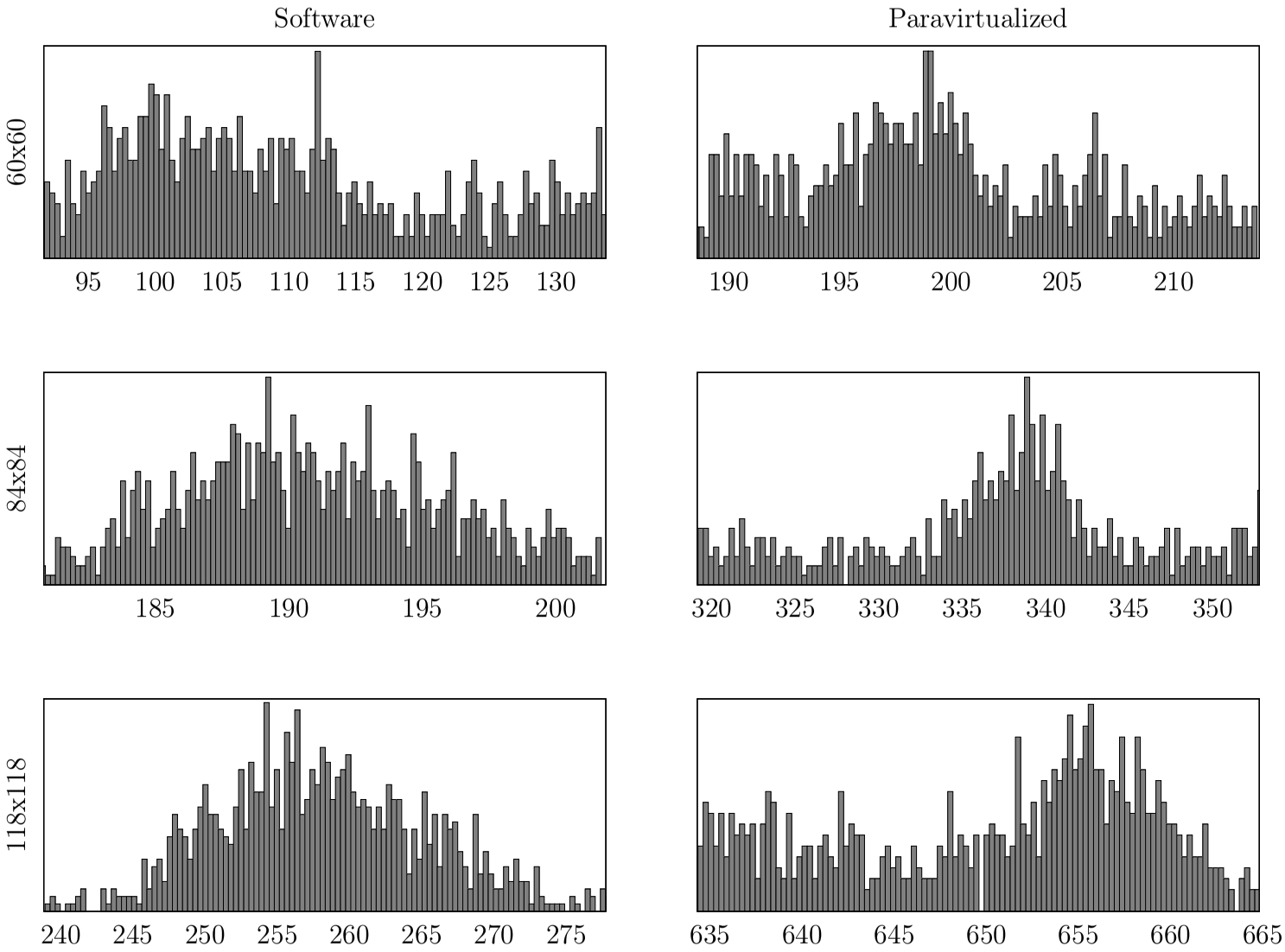

Figure 8.3: Histograms depicting benchmark elapsed frametimes in milliseconds and the density distribution of 1000 frames for the Chess benchmark key figure variations whilst software rasterized- and paravirtualized in Simics. The Y axis thus depict sample density. Its axis keys have been removed as they bear no relevance to the outcomes presented in this document.

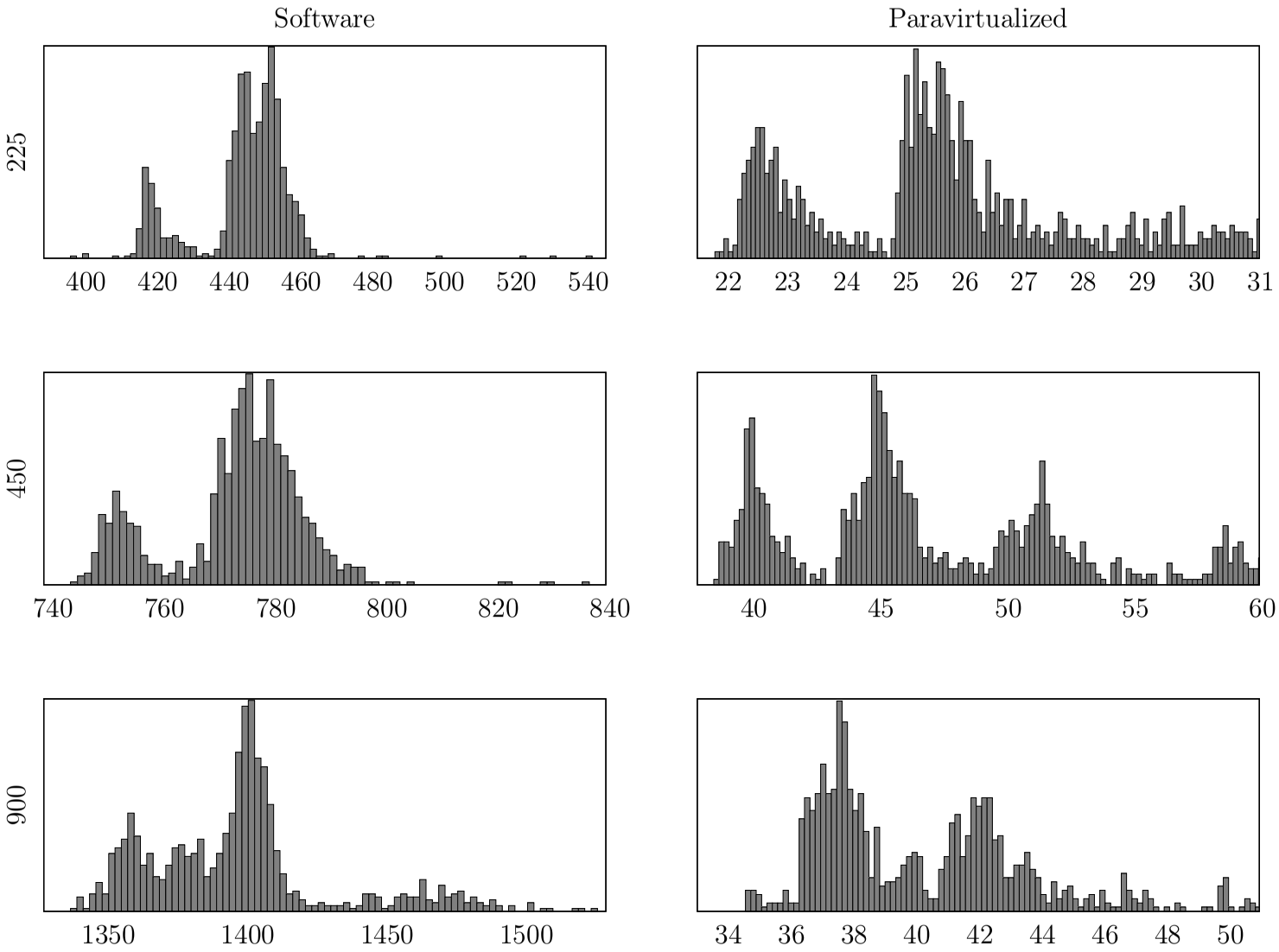

Figure 8.4: Histograms depicting benchmark elapsed frametimes in milliseconds and the density distribution of 1000 frames for the Julia benchmark key figure variations whilst software rasterized- and paravirtualized in Simics. The Y axis thus depict sample density. Its axis keys have been removed as they bear no relevance to the outcomes presented in this document.

alyzing the performance of stressed bandwidth in target-to-host communication; featuring relaying and rendering of a large texture to the simulation host. The graphs presented in figure 8.5 display erratic distribution of the sample density in both software rasterized- and paravirtualized Simics platforms. Suspicions as to why this is the case are presented in section 7.2.

By analyzing the data in presented in tables 8.3 and 8.4, it is clear that the Phong benchmark, in terms of average frame times, performs only marginally better or competitively to it's software rasterized equivalent. However, and mayhaps more interestingly so, executing the benchmark in the paravirtualized Simics environment results in major improvements in terms of frame time maximum and standard deviation; software rasterized samples topping in at 926.13 ms where the corresponding paravirtualized maximum is but 54.19 ms. Furthermore, one might argue that parts of the distribution align with the double-peak density distribution showcased by the benchmark when hardware accelerated on the simulation host (see figure 8.1).

It is unclear what causes the compiled paravirtualized Phong benchmark to perform only marginally better or competitively to its the software rasterized counterpart. The effect could be attributed to a bottleneck in the memory page traversal, as described in section 5.6, signifying a possible weakness in - in addition to the communication latency as established in paragraph Chess - the memory bandwidth of the Simics Pipe (see section 5.5).

In line with the recorded competitive average- and minimum frame times yet major improvements in frame time maximum and standard deviation; one might also argue that the relatively lightweight benchmark is misrepresentative in that its software rasterized execution is below that of the overhead induced by paravirtualization. However, due to the deviances described in section 7.2, this will not be elaborated upon further for the remainder of this dissertation.

8.2 Magic Instruction Overhead

In Simics, magic instructions incur a context switch when exiting the simulation and beginning execution in the real world. This affects the performance by forcing the simulation to no longer be executed in native mode; inhibiting the simulatory performance improvements given by hardware-assisted virtualization. Additionally, this also entail Simics no longer being able to utilize JIT compilation to speed up execution; having to rely on regular code interpretation. As such, in great numbers, magic instructions may potentially affect performance.

In line with the effect that this overhead may have on software such as benchmark Chess, two tests have been performed to establish the overhead magic instructions may induce. These tests have been run in two instances; one in which each magic instruction invocation is profiled separately using the methodology described in section 7.1, and another in which batches of magic instruction invo-

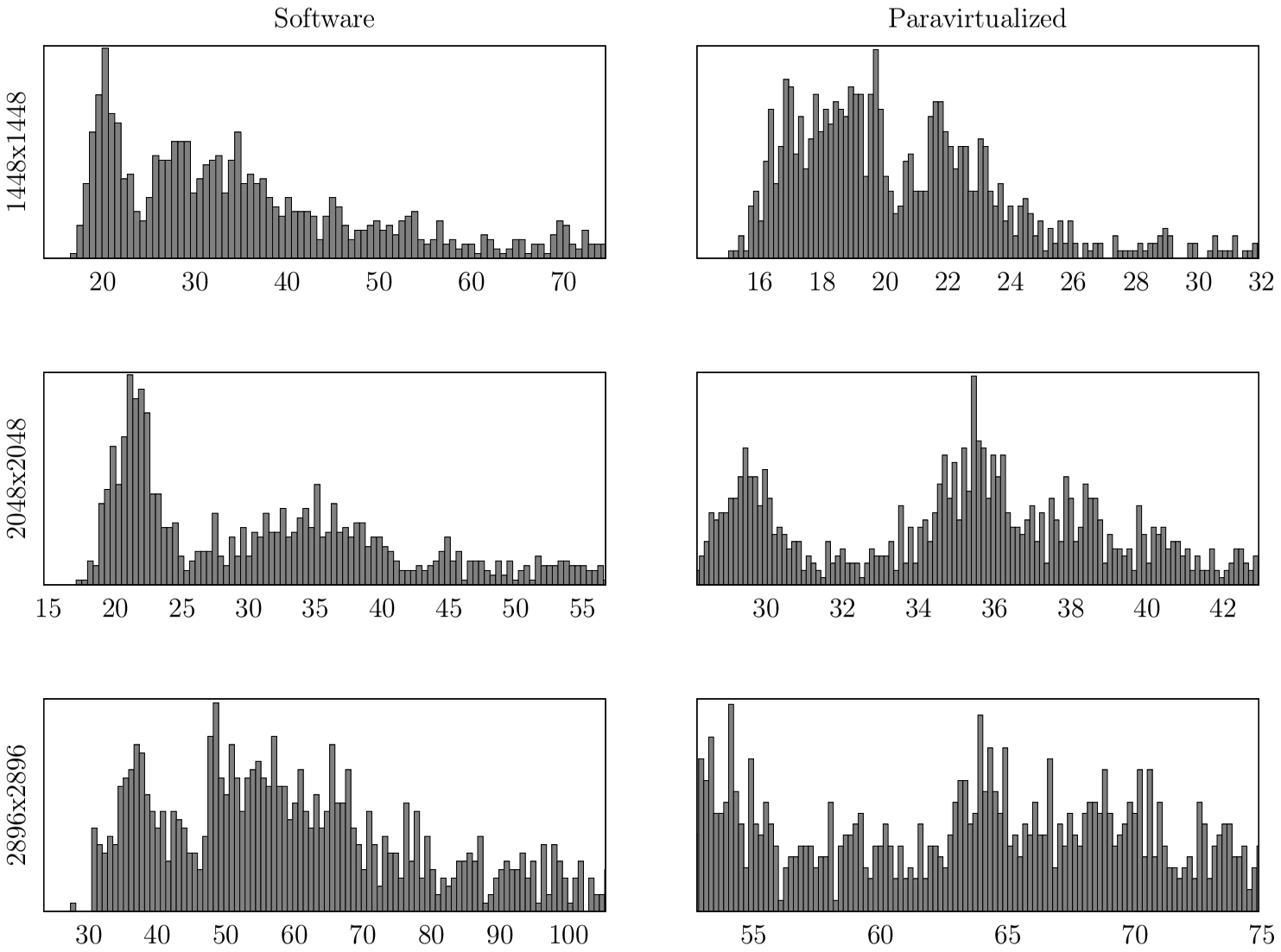

Figure 8.5: Histograms depicting benchmark elapsed frametimes in milliseconds and the density distribution of 1000 frames for the Phong benchmark key figure variations whilst software rasterized- and paravirtualized in Simics. The Y axis thus depict sample density. Its axis keys have been removed as they bear no relevance to the outcomes presented in this document.

```
int sercon = open("/dev/ttyS0",O_RDWR|O_NOCTTY);
```

```
for(unsigned i=0;i<1000;i++){
    write(sercon,"start",5);
    tcdrain(sercon);

    magic_instruction();

    write(sercon,"stop",4);
    tcdrain(sercon);
}
```

Figure 8.6: Profiling of magic instructions on a per-invocation basis.

```
for(unsigned i=0;i<1000;i++){
    write(sercon,"start",5);
    tcdrain(sercon);
    for(unsigned j=0;j<1000;j++){
        magic_instruction();
    }
    write(sercon,"stop",4);
    tcdrain(sercon);
}
```

Figure 8.7: Profiling of magic instructions in batches.

Min	Max	Std	Avg
-0.64	35.73	1.93	0.58

Table 8.5: Profiling results for individual magic instructions. Presented in milliseconds and modified in accordance to the profiling overhead established in section 7.1.

cations are measured to reduce influence of profiling overhead (see figures 8.6 and 8.7). This is due to profiling overhead, described in section 7.1, having influenced the original profilation results too greatly. As such, magic instructions has to be batched in order to be profiled correctly. Below, the results of both of these tests are presented.

Individual profiling The per-invocation profiling of 1000 magic instructions is presented in figure 8.8 as a histogram. All samples compiled in said test have been subtracted with the established average performance, as described in section 7.1. In line with this modification of the source data, due to the established profiling average overhead of 1.56 having relatively much influence on the data, the visualization features only ten histogram bins - in coagency with the approximate data. As such, any offset values below that of zero are placed in the first bin. Furthermore, any values outside that of the standard deviation are not visualized in this figure.

From this data, it is apparent that roughly 45% of profiled samples fall in the interval of 0-0.5 milliseconds, a relatively high measurement compared to that of the batch profiling (see paragraph Batch profiling). The figure is complemented with the analyzed data presented in table 8.5

Batch profiling In order to complement the data presented in paragraph Individual profiling, corresponding measurements for batch magic instruction invo-

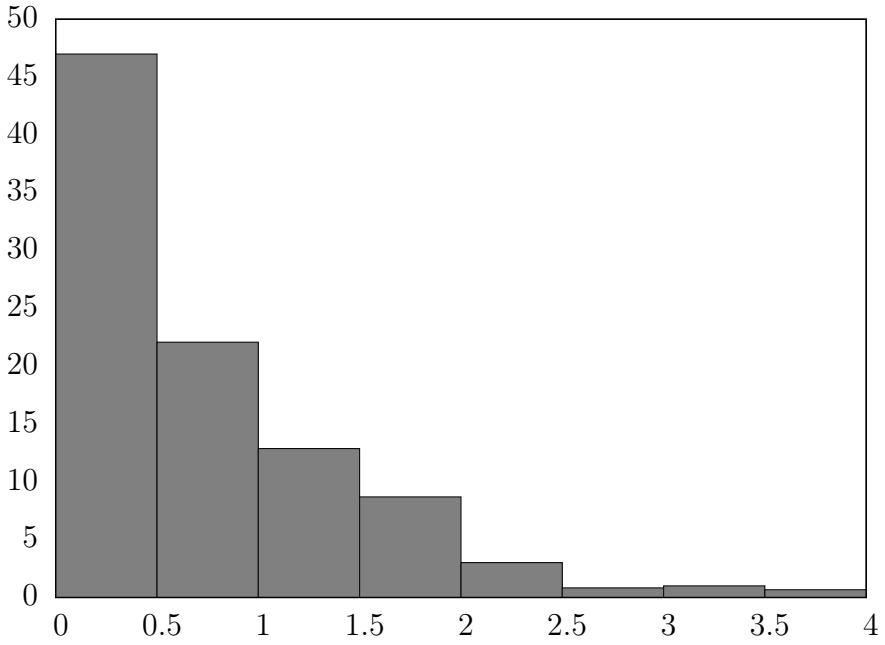

Figure 8.8: Histogram depicting approximative per-invocation profilation and percentage of density distribution of 1000 magic instructions in milliseconds.

Min	Max	Std	Avg
2.07	16.68	1.31	3.20

Table 8.6: Profiling results for batched magic instructions. Presented in milliseconds and modified in accordance to the profiling overhead established in section 7.1.

cations is collected in accordance to the pseudo code presented in table 8.6. In this way, we wanted to circumvent any obscurities induced by profiling overhead (see section 7.1).

From the data presented in paragraphs Individual profiling and Batch profiling, we may deduce that the profiling overhead induced when measuring the elapsed time of individual magic instructions causes misrepresentation in the actual elapsed time of said magic instructions. This is clear when analyzing the batch-wise data presented in table 8.6, where the cost of a magic instruction invocation is significantly lower. This may be caused by the system call `tcdrain`, which waits until all output written to the referenced serial console has been transmitted. One may speculate that said system call may be the cause for eccentric behavior in the profiling.

As such, the per-invocation profiling elaborated upon in paragraph Individual profiling is rendered less reliable due to the eccentric profiled behavior. From this we may conclude that the execution of 1000 magic instructions is expected to

induce an average overhead of roughly 3.20 ms, accounting for profiling errors as presented in section 7.1[2]. Due to the volatility of the profiling presented in paragraph Individual profiling, the author will refrain from using these results for any definite conclusions.

Thus, the established overhead cost for 1000 magic instructions is summarized in table 8.6.

8.3 Platform Comparison

In the sections above, results have been presented indicating performance gains, and potential for gains, in Simics platforms by the means of accelerating graphics using paravirtualization. However, in accordance to figure 8.2 and table 8.2, the platform on which the these results have been produced also utilizing paravirtualized methodology, there is much potential for improvement. The benchmarks, when executed in the paravirtualized Android emulator, exhibit better performance in each test performed for the purposes of this dissertation; most notably outperforming the software rasterized Simics platform for the Chess benchmark. The Chess benchmark, incurring such an overhead when paravirtualized in Simics (see paragraph Chess) due to overhead induced by magic instructions (see section 8.2), performs but roughly two times worse than its hardware accelerated counterpart at 30.96 milliseconds average (see tables 8.2 and 8.1). These results indicate potential for improvement in the target-to-host communications in Simics. In fact, considering the large similarities in the paravirtualized methodology for graphics acceleration these two platforms share, the reference performance of the QEMU-derived Android emulator may be considered a goal for a potential productification of paravirtualized graphics in the Simics full-system simulator.

These comparisons suggest that the recorded performance for the benchmarks, devised for the purpose of this study, is not necessarily representative for paravirtualized graphics in general. Furthermore, the comparison to the QEMU-derived Android emulator indicates that the shaping of the magic instruction-utilizing Simics Pipe - as established being the bottleneck during execution of the paravirtualized Chess benchmark (see section 8.2) - may have potential of improvement of approximately one order of magnitude (see tables 8.2 and 8.4).

[2]Note that this may be regarded as an optimum case, since batch execution of magic instructions may induce an executional pattern which could lead to improved results. However, this is just speculation.

Chapter 9

Discussion

For the purposes of productification of a paravirtualized graphics acceleration solution, as presented in this document, necessary improvements include endianness consideration in the communication in-between target- and host libraries[1], in addition to floating point format considerations[2]. If paravirtualization were to be used for accelerating graphics in the Simics full-system simulator, one would have to support such target- and host architectures as it is not all uncommon that these platforms differ. After all, clients often wish to simulate other platforms than those they currently possess.

Another cross-platform issue that comes to mind, when interfering with the target physical memory from outside simulation, is that is memory page locking. Such functionality are sometimes limited in operating systems; albeit entirely controlled by the user in Linux derived systems. Target system platform differences such as these may incur performance hindrances in that the amount of memory that may be locked could be limited; forcing the solution to perform its bytestream transmission in several instances of magic instructions. Furthermore, page locking functionality may not be accessible by the user whatsoever, suggesting further studies into how a paravirtualized solution for graphics acceleration may perform using other methodologies for trans-simulation communication; such as TCP/IP-networking.

In section 3.3, potentially costly maintenance of updated graphics frameworks is mentioned as drawback of paravirtualization as a methodology to achieve graphics acceleration. Later, in section 5.1, it is mentioned that the solution described in this document utilizes software to partly generate paravirtualized graphics ABIs where possible. In order to streamline the maintenance of a productified paravirtualized solution, such framework generation could be automated further by utilizing software such as `SWIG` or `SIL` to retrieve function signatures; no longer requiring developers to describe function headers by the means of configuration files (as described in section 5.1).

Furthermore, in line with the Simics attributes described in chapter 3, a certain set of behavior must be attained in order to align with Simics philosophy.

[1] Assumed to be of little endian order.

[2] Assumed to be that of IEEE 754.

Having analyzed the methodology of paravirtualization as a means to achieve graphics acceleration in the Simics full-system simulator, and making the assumptions presented in paragraphs Deterministic Execution, Checkpointing, and Reverse Execution, there is no obvious obstacle as to why advanced functionality such as deterministic execution, checkpointing, or reverse execution could not be integrated with paravirtualized graphics acceleration; nor is there any reason to believe that such attributes should severely impair performance of such a solution. Analyses and recommendations surrounding further study of said attributes, in terms of graphics acceleration by the means of paravirtualization, are presented below.

Deterministic Execution Following an API specification such as OpenGL ES, it naturally follows that there may be implementational differences in-between vendor drivers. Such variations are commonly insignificant and, although present, do not affect the end-user. Below, presuming the occurrence of such inconsistencies, a number of scenarios are presented in order to explain how this may affect the desirable traits of the Simics full-system simulator whilst simulating OpenGL ES.

In line with driver inconsistencies, there may be cause to believe that rounding of pixel values may vary dependent on host driver vendor. Typically, such variations are far from noticeable and would not offset simulation timing. Neither would this inconsequential execution affect the local deterministic trait during simulation, as the driver in itself is coherent in its execution. In Simics, determinism and repeatability are key values, meaning that simulation execution must not differ - independent on the host system. For example, one may pose the scenario of Simics users, using different driver vendors on their respective host machines, wishing to share checkpoints in which an application utilizing OpenGL ES is being executed. In this manner, posing that the execution paths of said application treads on functionality that is not intricately defined by the OpenGL ES specification, the output may differ. Note that, presuming the output (usually a buffer to-be presented on-screen) is not calculated upon, this does not affect the timing of the users' simulations. As such, this scenario maintains the determinism and repeatability traits of Simics.

However, if this varying output is calculated upon (a plausible scenario would be the compression of a graphics output screenshot, such as the ones presented in section 6.2) the timing of simulation is influenced which may propagate - effectively causing a state-change in the simulated CPU. In this manner, the deterministic trait of Simics would be broken, as certain instructions no longer correspond to their previously timing-accurate cycles.

More often than not, the scenario described in the above paragraph will not be an issue to deterministic execution in Simics, as the framebuffer usually only acts as means to present data to the user. In this way, the attribute of deterministic

execution would not be scathed by integration of paravirtualized graphics in the Simics full-system simulator presuming the preconditions described above are not compromised. As such, the incorporation of deterministic execution into the paravirtualized methodology as described in this document is possible through the set of assumptions, as described in this paragraph.

Checkpointing The incorporation of dvtglosscheckpointrestartCheckpoint / Restart schemes in the paravirtualized solution described in chapter 5 would require an efficient way of saving- and storing the OpenGL state, as present in the host system. The possibility of incorporating such functionality has, as mentioned in chapter 3, been studied and performed in the QEMU full-system simulator by Lagar-Cavilla et al. in their work on VMGL [19]. Such state retrieval, in the case of OpenGL ES 2.0, may be achieved by the `glGet`-set of OpenGL library methods[3]. Using these methods, one ought be able to retrieve the entirety of the OpenGL state variables present in the host system (see [59] for more information on the `glGet`-family of methods). Having retrieved these variables, one might subsequently store the changes in OpenGL states in some data structure; thus avoiding the need to store the entirety of these states every frame. Had the subject framework of paravirtualization been earlier versions of OpenGL, it might have been possible to use the push- and pop-methodology existent in those frameworks for the solution to know beforehand what states to save. However, this is just speculation. As such, albeit circumstantial, there ought be no reason as to why the introduction of checkpoint functionality should pose a problem to a paravirtualized graphics framework.

Reverse Execution In the solution devised for the purpose of this study, the paravirtualized solution renders to a window present in the host system (see chapter 5). Effectively, this means that the framebuffer is present in host memory. As such, Simics reverse execution functionality would not include said memory space; leading to the framebuffer presented in in the host system not to display in reverse in coagency with the reverse executing target system.

For a productification of a paravirtualized graphics framework for the Simics full-system simulator, it is probable that (unlike an academic study into the subject) it would be preferable if the would instead be located inside of the target system. If so would be the case, the framebuffer would be located in target memory, rendering said framebuffer eligible for reverse execution since such memory is stored in checkpoints. As such, assuming the graphics framebuffer being present in target system memory, reverse execution would be supported natively by Simics, and pose no hindrance to the functionality of reverse execution

[3]Note that such fit-for-purpose methods may not necessarily be available in other graphics frameworks, but since this dissertation focuses on OpenGL ES 2.0, such dilemmas will not be elaborated upon any further.

to paravirtualized libraries.

9.1 Hardware-assisted Virtualization

For the purposes of the experiment conducted, hardware-assisted virtualization have been utilized to accelerate performance of x86 instructions. Hardware-assisted virtualization reduces much of the overhead incurred by simulating hardware similar to that of the simulation host; thus stressing the capabilities of the paravirtualized acceleration devised for this study.

However, there are times when utilization of hardware-assisted virtualization is rendered impossible. Such examples include simulating systems other than that of the host machine, such as ARM architectures, or when a user wishes to set breakpoints in the software stack. At such times, the simulation is forced to execute without the benefits of native execution; often slowing down the simulation by several orders of magnitude. It may be argued that hardware assisted graphics acceleration may become particularly influential in such situations, as CPU-bound methodologies, such as software rasterization commonly used in Simics, may incur large performance issues that does not scale well with interpreted instruction sets. As such, it may be argued that the full capabilities of paravirtualized graphics acceleration, as compared to software rasterization, is not necessarily demonstrated in an environment utilizing hardware-assisted virtualization; a scenario lessening the impact of CPU-bound workloads.

Chapter 10

Conclusion

In chapter 8, we established strengths and weaknesses of paravirtualized graphics in the Simics full-system simulator; most notably the bottleneck introduced by the overhead of magic instructions in the Chess benchmark, which - in accordance to the findings presented in section 8.2 - made out the majority of elapsed frame time. As such, the methodology of creating benchmarks for the purpose of identifying such bottlenecks has confirmed original suspicions. In this way, the study has identified performance bottlenecks in great numbers of paravirtualized library functions when utilizing magic instruction technologies. However, in regard to performance improvements presented in chapter 8, the performed experiments establishes the competence of magic instructions for fast target-to-host communications of arbitrary size, including large data, in paravirtualized real-time graphics.

Furthermore, compiled results have showcased radical improvements for computationally intensive graphics kernels, as demonstrated by the Julia fractal benchmark, compared to its software rasterized Simics counterpart. As such, this study has, in addition to identifying the bottleneck induced by software rasterization for computationally intensive graphics kernels, accelerated graphics by up to 34 times; reducing frame time from that of 1415.23 ms to the real-time feasible count of 41.81 ms. As speculated upon in chapter 8, there is cause to believe that the paravirtualized solution described in this document may accelerate the Julia benchmark, as compared with software rasterized Simics, to that of two orders of magnitude. In this way, based off the results presented in chapter 8, the experiment has identified the potential of using paravirtualization for the means of accelerating graphics to that of real-time performance; testimonial to the results presented by Lagar-Cavilla et al. in their work on using paravirtualization to accelerate graphics (see [19]).

Additionally, beyond that of accelerated graphics, the results gathered for the purpose of this study indicates performance improvements in terms of maximum frame times (both in Chess and Phong benchmarks); leading to that of significantly improved standard deviation. In line with stable frame rates being prerequisites for real-time applications, this further indicates, in coagency with better frame times as portrayed by the Julia benchmark, the feasibility of uti-

lizing paravirtualized methodologies for the purposes of accelerating graphics in virtual platforms.

As to summarize; this thesis has demonstrated performance improvements by accelerating graphics using paravirtualization. By this addition, the benefits of graphics virtualization has been identified as performance improvements of up to 34 times; speculating in possible performance improvements of up to two orders of magnitude. The bottlenecks the solution devised for the purpose of this experiment have been identified as magic instruction overhead. As such, a possible drawback of graphics paravirtualization has been identified as a weakness to large amounts of framework invocations.

In chapter Discussion, an analysis of the prerequisites of advanced functionality such as deterministic execution, checkpointing, and reverse execution is presented, along with the conclusion that such integration ought be possible presuming a number of assumptions. Thus, this dissertation claims paravirtualization as a successful formula for graphics acceleration in virtual platforms; with no obvious obstacles obstructing future Simics advanced functionality.

As to conclude; for the purposes of this dissertation, a solution for graphics acceleration has been implemented in the Simics full-system simulator by the means of paravirtualization (see chapter 5). The end-result is a solution which may generate libraries imitating the EGL- and OpenGL ES 2.0 libraries. By the means of preloading, the solution may effectively overload and spy in on an applications EGL utilization with a target window; without inhibiting said exchange - allowing unmodified OpenGL applications to be accelerated from within the simulation target. Said solution communicates by the means of low-latency magic instructions, and there is no apparent limit as to how much memory may be shared[1] (see section 5.5). As such, throughout this document, several of the issues pertaining to graphics acceleration via paravirtualization, including - but not limited to, target-to-host memory sharing, have been tackled, studied, and presented.

For the purposes of the experiment performed for the sake of thesis, three benchmarks have been developed with the distinct purpose of profiling bottlenecks and potential weaknesses and strengths of graphics acceleration by the means of paravirtualization in the Simics full-system simulator (see section 6.2). Said benchmarks have been performed on the host system, the QEMU-derived Android emulator, and in software rasterized- and paravirtualized Simics platforms. Furthermore, the benchmarks (created specifically to identify issues related to memory latency, memory bandwidth, and computational complexity in the paravirtualized solution) have contributed to our understanding of the difficulties facing paravirtualized graphics acceleration.

[1] In Linux, there is a limit as to how much memory a user-space application may lock. However, this limit may be set to appropriate limits by the user beforehand, alternatively running the application as a super-user; circumventing said limit.

In chapter 8, we have presented an analysis of the results compiled throughout the performed experiments; along with an investigation into the scalability of the graphics acceleration for the tested benchmarks; for software rasterized- and paravirtualized Simics platforms. we have compiled and presented an analysis on the benefits and drawbacks of paravirtualization as a means to achieve graphics acceleration in virtual platforms; backed by hard data produced by a number of benchmarks stressing key points in the solution with the purpose of identifying both strengths and weaknesses in the discussed methodology. Accordingly, this dissertation has established the feasibility of using paravirtualization to accelerate graphics in virtual platforms to that of real-time qualities.

Additionally, the collected results have been compared with another platform using similar methodologies to accelerate the same graphics framework. Based off of these results, chapter 8 presents an analysis comparing the two platforms; being the QEMU-derived Android emulator and the paravirtualized Simics solution. From this analysis, we have established points of improvement in the paravirtualized solution developed for the Simics simulator. Furthermore, and based on the performance boasted by the Android emulator's paravirtualized graphics acceleration when stressed by the Chess benchmark, we have predicted possible improvements in the Simics Pipe (see section 5.5) communications link of up to one order of magnitude.

As such, in coagency with the results compiled in chapter 8, collected by the means presented in chapter 6, and based on the solution portrayed in chapter 5, pertaining to the idea of real-time graphics in detailed full-system simulators, this dissertation suggests utilizing high-level paravirtualization to accelerate graphics-, and as means to overcome accessibility bottlenecks, in virtual platforms.

Chapter 11

Future Work

The solution devised for the purpose of this study may be advanced in a number of ways in order to support a higher number of platforms, automation in ABI generation, an array of performance improvements, and general enhancements to make the paravirtualized solution more flexible during maintenance. Additionally, in consideration to incorporation of a graphics acceleration solution into the Simics full-system simulator, the solution ought be improved in terms of cross-platform capabilities. Below, recommendations for future study are presented.

For the purposes of improved solution performance, the author would recommend investigation into the possibility of command serialization batching; that is, the functionality to queue framework invocations and send the command stream to the host system. By batching framework invocations rather than serializing individual invocations, one might drastically reduce the number of magic instructions performed during paravirtualization. Considering magic instructions having been identified as a bottleneck for the solution devised for the purposes of this study, such an optimization may improve solution performance. However, to accommodate batch-jobs of OpenGL ES 2.0 function invocations, one would have to ensure that data to which pointer arguments refer is not modified before being transmitted to the simulation host. Alternatively, the solution could copy all argument data when invoked; only to transfer said data at a later stage.

In order to strengthen the validity of this thesis, the author would suggest complementary study into similar conditions where the target solution does not utilize hardware-assisted virtualization. Without hardware-assisted acceleration to boost the performance of CPU-bound workloads, the benefits of paravirtualized methodologies may be emphasized.

For further studies into paravirtualization as a means to accommodate graphics acceleration in virtual platforms, the author would like to suggest complementary benchmarking of accelerated graphics performance. This is due to the benchmarks presented for the purpose of this study having effectively been mini-benchmarks; stressing particular suspected bottlenecks in the solution. In line with chapter 8 and 10 having concluded paravirtualization as feasible for the means of graphics acceleration in virtual platforms, the methodology ought be profiled further with more verbose benchmarks in order to more effectively estab-

lish the magnitude of performance gains a paravirtualized solution may achieve.

Furthermore, and for the purposes of complementing this dissertation in particular, the author would like to suggest additional tests stressing target-to-host communications. Preferably, said tests would stress the communication by other means than profiling the sampling of a large texture, since such a test may cause volatile performance in the reference material (being software rasterized Simics), possibly due to cache misses (see section 7.2). When performing such a test, it may be of value to profile the overhead induced by the memory table traversal described in section 5.6.

In addition to the complementary studies suggested in the above paragraphs, certain advanced functionality may be viable for further study; being API Extensions and Safety Critical Considerations. The concepts of these scenarios are presented below.

API Extensions The essence of system simulation is often to simulate a system other than that of the simulation host. As of such, one could pose the scenario of a Linux host system simulating a machine running some variation of the Windows OS. In this case, it is possible that user applications in the simulation utilize the DirectX framework to render graphics; whereas the host machine only features the OpenGL libraries.

Earlier this year, Valve Corporation released software capable of converting DirectX 9.0c-code to that of OpenGL [43]. Albeit limited in its capabilities, the functionality of translating in-between one, platform-specific, framework to that of a cross-platform framework may be practical. If such a solution could be implemented to translate- and execute some target program on-the-fly in a virtual machine, in which the host system lacks the capability to interpret the original framework, this may extend the area-of-application for full-system simulation. As such, further studies into the feasibility of similar functionality ought be considered.

Safety Critical Considerations Paravirtualization of the OpenGL ES 2.0 library induces the need to maintain various OpenGL state variables in target- and host systems (see chapter 5). The effective overload of a certain framework opens up the possibility of modifying the behavior of that library in certain ways; including how methods are invoked, in what order, and pre-/post-hooks of certain actions.

This ability may be exploited to bend the capabilities of that library to the needs of a certain user; for example, a safety critical version of a paravirtualized library. Such a library could, for example, extend the capabilities of the OpenGL error state, keeping multiple such instances in memory to prevent the state from being overwritten by the user; effectively making the framework more transparent. Another extension, and mayhaps a more usable one, could ensure that the

OpenGL state is consistent with that as requested by the user; for example, disabling any vertex attribute that the user has not explicitly enabled. As such, further studies into the feasibility of such functionality may be considered.

List of Terms

Checkpointing	See section 4.2.
Deterministic Algorithm	An algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states.
Deterministic Execution	See section 4.1.
EGL	EGL is an interface between Khronos rendering APIs (such as OpenGL, OpenGL ES or OpenVG) and the underlying native platform windowing system.
Fedora (operating system)	An operating system based on the Linux kernel, developed by the community-supported Fedora Project and owned by Red Hat.
Full-system simulation	Full-system simulation denotes an architecture simulator that simulates an electronic system at such a level of detail that complete software stacks from real systems can run on the simulator without any modification.
Hardware-assisted Virtualization	A platform virtualization approach that enables efficient full virtualization using help from hardware capabilities, primarily from host processors. Also known as accelerated virtualization.
Hypersimulation	Technologies making a simulated processor skip through, what would effectively be, idle time rather than executing `nop`- type instructions cycle by cycle.

IEEE 754	IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).
Khronos Group	The Khronos Group is a member-funded industry consortium focused on the creation of open standard APIs on a wide variety of platforms.
Magic Instruction	See section 3.5.
Memory Page Locking	Some OSs may force memory pages not to be swapped to secondary storage, referred to as 'pinned', 'locked', 'fixed', or 'wired' memory pages.
Paravirtualization	A virtualization technique that presents a software interface to virtual machines that is similar, but not identical to that of the underlying hardware.
PCI passthrough	PCI passthrough may allow a simulation target exclusive control of physical PCI devices, such as GPUs, audio controllers, and USB controllers.
QEMU	A free and open-source hosted hypervisor that performs hardware virtualization.
Reverse Execution	See section 4.3.
Simulation Host	A system on which a virtual platform is run.
Simulation Target	A simulated system.
Simulation Timing	Time intervals when state changes in a simulated system occur.
The Simics full-system simulator	See chapter 4.
X Window System	A windowing system for bitmap displays, common on UNIX-like operating systems.

List of Acronyms

ABI	Application Binary Interface.
API	Application Programming Interface.
CPU	Central Processing Unit.
FLOPS	Floating-point Operations Per Second.
FPS	Frames Per Second.
GPGPU	General Purpose computing on Graphics Processing Units.
GPU	Graphics Processing Unit.
HAXM	Hardware Accelerated Execution Manager.
ISA	Instruction Set Architecture.
JIT	Just-in-Time.
JNI	Java Native Interface.
KVM	Kernel-based Virtual Machine.
MIPS	Million Instructions Per Second.
MMU	Memory Management Unit.
OS	Operating System.
PCI	Peripheral Component Interconnect.
SICS	Swedish Institute of Computer Science.
SIMD	Single Instruction, Multiple Data.
TTM	Time-to-Market.

UART	Universal Asynchronous Reciever/Transmitter.
WARP	Windows Advanced Rasterization Platform.

Bibliography

[1] M. S. A. Whitaker and S. D. Gribble., "Denali: Lightweight virtual machines for distributed and networked applications," Feb. 2002.

[2] D. Aarno and J. Engblom, "Simics* overview," *Intel Technology Journal*, vol. 17, no. 2, pp. 8–31, Dec. 2013. [Online]. Available: http://intel.ly/1cY2bM7

[3] T. Akgul and V. J. Mooney III, "Assembly instruction level reverse execution for debugging," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, no. 2, pp. 149–198, Apr. 2004. [Online]. Available: http://doi.acm.org/10.1145/1018210.1018211

[4] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin, "Simulating a $2m commercial server on a $2k pc," *Computer*, vol. 36, no. 2, pp. 50–57, Feb. 2003. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1178046

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462

[6] A. Bartolini, M. Cacciari, A. Tilli, L. Benini, and M. Gries, "A virtual platform environment for exploring power, thermal and reliability management control strategies in high-performance multicores," in *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*, ser. GLSVLSI '10. New York, NY, USA: ACM, 2010, pp. 311–316. [Online]. Available: http://doi.acm.org/10.1145/1785481.1785553

[7] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[8] T. Bergan, J. Devietti, N. Hunt, and L. Ceze, *The Deterministic Execution Hammer: How Well Does it Actually Pound Nails?*, 2011.

[9] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, "Mambo: A full system simulator for the powerpc architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 8–12, Mar. 2004. [Online]. Available:

http://doi.acm.org/10.1145/1054907.1054910

[10] J. Cohen, "Non-deterministic algorithms," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 79–94, Jun. 1979. [Online]. Available: http://doi.acm.org/10.1145/356770.356773

[11] J. Devietti, "Deterministic execution for arbitrary multithreaded programs," Ph.D. dissertation, Seattle, WA, USA, 2012, aAI3552763.

[12] M. Doggett, "Texture caches," *IEEE Micro*, vol. 32, no. 3, pp. 136–141, May 2012. [Online]. Available: http://dx.doi.org/10.1109/MM.2012.44

[13] X. Guo, H. Jiang, and K.-C. Li, "A checkpoint/restart scheme for cuda applications with complex memory hierarchy," in *Proceedings of the 2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, ser. SNPD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 247–252. [Online]. Available: http://dx.doi.org/10.1109/SNPD.2013.5

[14] J. G. Hansen, "Blink: Advanced display multiplexing for virtualized applications," in *Proceedings of the 17th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, ser. NOSSDAV '07, 2007, pp. 14–20.

[15] D. Harris and S. Harris, *Digital Design and Computer Architecture*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[16] G. J. Holzmann, "Spin model checking - reliable design of concurrent software," *Dr. Dobb's Journal*, no. 1, pp. 92–97, Oct. 1997.

[17] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[18] S. Koerner, A. Kohler, J. Babinsky, H. Pape, F. Eickhoff, S. Kriese, and H. Elfering, "Ibm system z10 firmware simulation," *IBM J. Res. Dev.*, vol. 53, no. 1, pp. 131–142, Jan. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1850618.1850630

[19] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara, "Vmm-independent graphics acceleration," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 33–43. [Online]. Available: http://doi.acm.org/10.1145/1254810.1254816

[20] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.180

[21] R. Leupers and O. Temam, *Processor and System-on-Chip Simulation*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[22] P. S. Magnusson, "Full-system simulation: Escape from reality." *Electronic Design*, vol. 52, no. 21, p. 18, 2004. [Online]. Available: http://bit.ly/1c57R8N

[23] ——, "Foreword: Simics*—the early years," *Intel Technology Journal*, vol. 17, no. 2, p. 7, Dec. 2013. [Online]. Available: http://intel.ly/1cY2bM7

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/2.982916

[25] A. Munshi, D. Ginsburg, and D. Shreiner, *OpenGL(R) ES 2.0 Programming Guide*, 1st ed. Addison-Wesley Professional, 2008.

[26] E. Nilsson and M. Fredriksson., "Performance variations in emulated gpgpu kernels," Jan. 2014. [Online]. Available: http://bit.ly/1n0EchA

[27] A. Ortiz, J. Ortega, A. F. Díaz, P. Cascón, and A. Prieto, "Protocol offload analysis by simulation," *J. Syst. Archit.*, vol. 55, no. 1, pp. 25–42, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2008.07.005

[28] G. Rechistov, "Simics* on shared computing clusters: The practical experience of integration and scalability," *Intel Technology Journal*, vol. 17, no. 2, pp. 124–135, Dec. 2013. [Online]. Available: http://intel.ly/1cY2bM7

[29] N. Regola and J.-C. Ducom, "Recommendations for virtualization technologies in high performance computing," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 409–416. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2010.71

[30] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The simos approach," *IEEE Parallel Distrib. Technol.*, vol. 3, no. 4, pp. 34–43, Dec. 1995. [Online]. Available: http://dx.doi.org/10.1109/88.473612

[31] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parsc: Synchronous parallel systemc simulation on multi-core host architectures," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 241–246. [Online]. Available: http://doi.acm.org/10.1145/1878961.1879005

[32] S.-T. Shen, S.-Y. Lee, and C.-H. Chen, "Full system simulation with qemu: An approach to multi-view 3d gpu design," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, May 2010, pp. 3877–3880.

[33] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May 2005. [Online]. Available: http://dx.doi.org/10.1109/MC.2005.173

[34] A. Veselyi and J. Ayers, "Early hardware register validation with simics*," *Intel Technology Journal*, vol. 17, no. 2, pp. 100–111, Dec. 2013. [Online]. Available: http://intel.ly/1cY2bM7

[35] F. J. Villa, M. E. Acacio, and J. M. García, "Evaluating ia-32 web servers through simics: A practical experience," *J. Syst. Archit.*, vol. 51, no. 4, pp. 251–264, Apr. 2005. [Online]. Available:

http://dx.doi.org/10.1016/j.sysarc.2004.09.003

[36] R. S. Wright, N. Haemel, G. Sellers, and B. Lipchak, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 5th ed. Addison-Wesley Professional, 2010.

[37] J. J. Yi and D. J. Lilja, "Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations," *IEEE Trans. Comput.*, vol. 55, no. 3, pp. 268–280, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1109/TC.2006.44

[38] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Paravirtualization for hpc systems," in *Proceedings of the 2006 International Conference on Frontiers of High Performance Computing and Networking*, ser. ISPA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 474–486. [Online]. Available: http://dx.doi.org/10.1007/11942634_49

[39] T. Yu, W. Srisa-an, and G. Rothermel, "Simtester: A controllable and observable testing framework for embedded systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: ACM, 2012, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151034

[40] Y. Zhang, X. Guo, H. Jiang, and K.-C. Li, "A checkpoint/restart scheme for cuda applications with complex memory hierarchy," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, July 2013, pp. 247–252.

Web References

[41] "Under the hood of the android emulator," Available: http://bit.ly/UmaWtA, accessed: 08-06-2014.

[42] D. Airlie, "Introducing virgil - 3d virtual gpu for qemu," Available: http://bit.ly/1oACNCd, jul 2013, accessed: 17-05-2014.

[43] V. Corporation, "Direct3d to opengl abstraction layer," Available: http://bit.ly/1gHfq82, 2014, accessed: 19-05-2014.

[44] X. Ducrohet and R. Meier, "A faster emulator with better hardware support," Available: http://bit.ly/Lv2tPA, April 2012, accessed: 04-02-2014.

[45] J. Engblom, "Determinism, simics, and flying piggies," Available: http://bit.ly/1bmWkMb, November 2012, accessed: 04-02-2014.

[46] ——, "Back to reverse execution," Available: http://bit.ly/1nN9pH5, June 2013, accessed: 04-02-2014.

[47] ——, "Collaborating using simics recording checkpoints," Available: http://bit.ly/1nN23Dn, May 2013, accessed: 04-02-2014.

[48] Google, "Using the emulator," Available: http://bit.ly/1bqCh2G, 2013, accessed: 04-02-2014.

[49] ——, "Android hardware opengles emulation design overview," Available: http://bit.ly/1fkKZUc, apr 2014, accessed: 27-04-2014.

[50] ——, "qemu-timer.h," Available: http://bit.ly/1icXSfc, 2014, accessed: 08-06-2014.

[51] G. Hofemeier, "Android* - performance results for android emulators - with and without intel haxm," Available: http://intel.ly/1giSbRu, December 2013, accessed: 13-05-2014.

[52] Intel, "Android* 4.4 (kitkat) x86 emulator system image," Available: http://intel.ly/1iI77TJ, 2013, accessed: 13-05-2014.

[53] ——, "Intel® hardware accelerated execution manager," Available: http://intel.ly/1jnoj0C, nov 2013, accessed: 13-05-2014.

[54] T. M. Jones, "Linux virtualization and pci passthrough," Available: http://ibm.co/1eTEnpm, October 2009, accessed: 05-02-2014.

[55] Microsoft, "Windows advanced rasterization platform (warp) guide," Available: http://bit.ly/19hiwrZ, 2013, accessed: 04-02-2014.

[56] J. Miller, "Wind river to add virtutech simics products to comprehensive embedded software portfolio," Available: http://bit.ly/1g0jbj2, February 2010, accessed: 04-05-2014.

[57] NASA, "Independent test capability (itc)," Available: http://1.usa.gov/1bHVIAQ, January 2014, accessed: 10-02-2014.

[58] S. Overflow, "Is the emulator clock synced to the real system clock?" Available: http://bit.ly/1kWhTqa, April 2011, accessed: 08-06-2014.

[59] I. Silicon Graphics, "Opengl es 2.0 reference pages," Available: http://bit.ly/1vskz87, 2006, accessed: 19-05-2014.

[60] C. Stylianou, "Speeding up the android* emulator on intel® architecture," Available: http://intel.ly/1jntJJ7, November 2013, accessed: 13-05-2014.

[61] I. The NetBSD Foundation, "About netbsd/amd64," Available: http://bit.ly/1lM0HHK, apr 2014, accessed: 17-05-2014.

[62] I. VMware, "Experimental support for direct3d," Available: http://bit.ly/S3nyo4, may 2014, accessed: 17-05-2014.

[63] I. Wind River Systems, "Nasa meets satellite project testing and verification goals with wind river simics," Available: http://bit.ly/1eltFqU, 2014, accessed: 10-02-2014.

Colophon

This dissertation is typeset in Times New Roman using the LaTeX- and BibTeX typesetting systems by Leslie Lamport, originally devised from Donald Knuth's TeX, and has been written using, amongst other environments, the Sublime Text- and the Atom text editor. Furthermore, the environment utilizes packages by, inter alia, Brent Longborough to extract Git metadata.

Bibliographies are structured in accordance to the IEEE sorted style and entries are, when available, retrieved and formatted in accordance to the ACM digital library.

Graphs are compiled using GNUplot and figures have been devised using yEd.

The thesis is revision controlled using Git and hosted on GitHub; whilst being built using Python.

This document was compiled in Release configuration 2014-06-26 and last edited by Eric Nilsson 2014-06-26. Corresponding revision may be identified by commit hash `52037bc`.

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57