Cycle-accurate Benchmarking of JavaScript Programs

Anders Handler

Kongens Lyngby 2012
IMM-MSC-2012-20

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

JavaScript is the language for building web applications. To meet the demand for a good user experience, when using todays fast growing web applications, JavaScript engines needs to be highly optimized. In order to optimize the JavaScript engines, analysis of the consumed execution time is essential. To gather execution time information, this thesis provides a framework to benchmark JavaScript applications as accurately as possible. The timings are collected at instruction level using full-system simulation. As the user experience is depended on the wall clock execution time, the entire software stack is simulated. The high accuracy is obtained by using the cycle accurate x86 64-bit instruction set simulator gem5. The simulated system consists of a Gentoo Linux system with WebKit and JavaScriptCore for the JavaScript execution. Several extensions were made to gem5 in order to gather the detailed timing information.

Acknowledgement

I would like to thank Lars F. Bonnichsen, Laust Brock-Nannestad, Jesper H. Hansen and Niklas Q. Nielsen for their support and our interesting discussions in the office. I am truly grateful for the reviews done by Daniel L. Winther and Rasmus Handler, who both supplied me with valuable feedback.

I would also like to thank all the people at the gem5 mailing list for fast and helpful feedback, special thanks goes to Ali Saidi and Gabriel M. Black.

Finally I would like to thank my helpful advisors Christian W. Probst and Sven Karlsson.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the increasing size and complexity of web applications, there is a rising demand for faster JavaScript execution. To achieve higher JavaScript execution performance, the JavaScript engines need optimization. In order to reveal what is slowing the JavaScript execution, detailed information about the entire system is needed. To aid JavaScript engine development this thesis builds a benchmarking framework to analyze the low level behavior of JavaScript execution, by saving the instruction traces of the execution. This requires configuring a full-system simulator and building the entire software stack for the simulated system consisting of operating system, browser, JavaScript engine, and the JavaScript to execute.

During the last decade websites with little or no dynamic content have evolved into large web applications, featuring email clients, social networking applications and full office suites. With the capabilities of web applications the trend is moving from local applications to web applications, featuring advantages such as easy application maintenance, cross platform support and access from any computer connected to the Internet. These large web applications heavily rely on the execution of JavaScript, as all client side behavior is done using JavaScript. Therefore to ensure good performance, the underlying system should be able to execute JavaScript programs efficiently. However as web applications continue to grow in complexity and features, getting good performance becomes harder and harder. Poor performance will lead to un-

responsive applications and flaky animations. This will lead to a bad user experience which eventually will decrease the popularity of your application or even make it unusable.

Even with the latest revision of the HTML standard, the HTML5 [29] standard, JavaScript continues to play an essential role for building web applications. Where the HTML standard earlier made space for using different scripting languages, revision 5 has made the type declaration optional, having JavaScript as the default choice. So even with the new HTML elements and the extensions to CSS, JavaScript will inevitable be the dominant language of future web application. And not only will JavaScript be running client side, running JavaScript server side is currently gaining popularity through platforms like NodeJS [21].

Also Microsoft has with their announcement of their next operating system, Windows 8, upgraded the JavaScript language to a first class member in their Visual Studio development suite [20]. Supporting the claim that JavaScript will be the dominant language of future web applications.

With these rising demands to run large JavaScript applications, there is a need for faster JavaScript execution. In order to optimize the performance of the JavaScript execution, optimizations to the JavaScript engine have to be made. To reveal, wherever changes to the engine are improving performance, benchmarks have been made. These benchmarks are often provided by the creators of the JavaScript engines themselves and it can be argued how the performance reflects the performance users experience [23]. What matters to users is the total time it takes to execute the code.

Different techniques exist to profile different layers of the software stack. Regular software is typically instrumented and profiled using a profiler to reveal the time consuming parts of the application. JavaScript programs can be profiled using tools such as the Firebug plugin for Firefox, or using the Web Inspector in WebKit based browsers such as Chromium or Safari. However profiling in the individual layers of the software stack does not yield the entire truth about the execution, namely the total time it takes to execute the requested operation.

To benchmark a system to reveal the total execution time, the entire software stack will have to be taken into account. This differs from normal practice of benchmarking programs, where only the program itself is tested. This is of course due to the fact that under normal conditions a programmer can only optimize the program itself and not the entire system it runs on. But the overwhelming complexity of the entire software stack should not be a bad excuse not to analyze it. Questioning the complexity of the software stack has

been done before. Already back in 1992 researchers asked themselves “where have all the cycles gone?” [2]. Because a deep understanding of how things are working is the key to create better systems, wherever it should be faster, simpler or more secure.

1.1 Problem Description

In order to improve the execution of JavaScript programs, we need an accurate measure of the execution. However a normal computer cannot measure these timings itself, as it would require running additional code to do the measurements. By using a simulator, the exact trace of the execution can be captured. The timings can also be measured accurately instead of using approximate sampling methods. To measure all parameters influencing the execution, the entire software stack of a computer should be simulated. Using a full system simulator, an exact measurement of the JavaScript execution can be saved.

The measurements of the JavaScript execution should include timings and instruction count. It should be possible to save complete traces of the execution of a JavaScript program with timing of the individual instructions. In order to only save the relevant information it should be possible to control the trace output. From the trace output we should be able to extract mappings to the source code, from which the instructions are made.

This project strives to build a framework for easy measurements of JavaScript programs at instruction level. The framework should consist of a full system simulator and a system to simulate, where the JavaScript programs to analyze can be executed. The simulated system should resemble the computer of a regular web user using the latest software stack, consisting of operating system, web browser and JavaScript engine. In order to extract the required measurements of the simulator, it should be modified or extended accordingly.

1.2 Report Structure

This report is structured as follows. First an overview of the framework infrastructure is given, describing the different components needed to run a simulation such as the Linux kernel and a disk image. The chapter also describes how the components are made to contain the desired features. Chapter 3 goes into detail with the gem5 simulator and how it is configured and extended in

order meet the requirements of the framework described above. This includes extracting data from the memory of the simulated system, controlling when to save the traces and how to modify the trace output. In chapter 4 we show the capabilities of the framework, by tracing a simple JavaScript program. In the performance chapter we run benchmarks with four differently configured simulated CPU models to compare their performance with the performance of a physical CPU. Chapter 6 suggests future work, both in regard to extending the simulator, and for improving the framework build here.

1.3 Time Plan

To successfully execute this project, a time plan was followed. As the time plan was made on paper, it has been reconstructed in figure 1.1. Since the time plan has changed over time, figure 1.1 shows both the time plan made in the beginning of the project (original) and the plan that has been adjusted during the project in order to fit the challenges met (actual). The initial plan contained both a part for setting up the framework and a part for using it. Setting up the framework required more effort than expected, which resulted in the changes to the time plan. In both plans the dates denote deadlines.

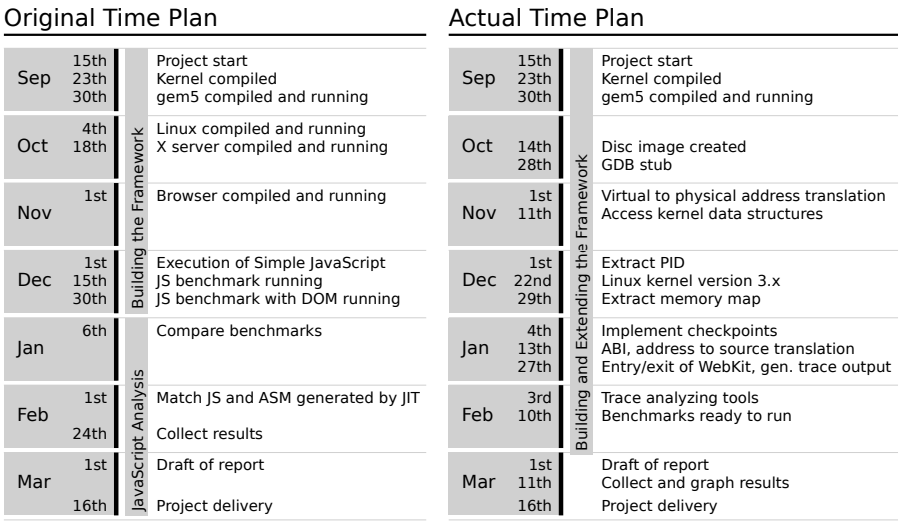

Figure 1.1: Original and actual time plan for the project.

1.4 Notation

Throughout this thesis the following special conventions and words will be used.

We are working with the concept of a simulated system, thus the word *host* refers to the computer where the simulation is actually computed. To distinguish simulated hardware from non-simulated hardware, the term *physical hardware* is used to emphasize that we are talking about non-simulated hardware.

Please note that some names, like *gem5*, does not convey to standard conventions of using capital starting letter of proper nouns. However as the different style is part of the brand, the same style will be used here. Other applications like *WebKit* makes use of camel case, which is also considered part of the brand and thus also used here.

This thesis contains a lot of references to code, which, for clarity, it written in `typewriter` font. Beside references to code, references to paths of files and common Linux commands have also been written using typewriter font.

Chapter 2

Infrastructure

This chapter describes the infrastructure of the framework, which includes the choice of simulator, the structure of the chosen simulator and how the simulated system was build.

2.1 Choice of Simulator

To build a JavaScript program benchmarking framework with the characteristics of being cycle accurate, we need a cycle accurate simulator and a system to simulate. As the goal of this project is to mimic the current platform of a regular web user, the simulated system should be based on mainstream technology.

The most widespread architecture of modern computers are x86, where most computers are 64-bit compatible, although many computers are still deployed with a 32-bit operating system. As this project is improving the technology of tomorrow, only x86 64-bit will be considered here.

The operating system Linux has been chosen, as it is the most widespread open source operating system.

There exists several full system simulators, but limiting the search to cycle accurate x86 64-bit simulators greatly reduces the number. The most well known is Simics. Simics is a commercial simulator, which supports a wide range of processor architectures beside x86 (32 and 64-bit), it also supports ARM, MIPS, Power PC and SPARC. Simics was one of the first full system simulators on the market [17] and it is still actively developed. Because Simics is a commercial simulator, license constraint prevented us from using this simulator.

PLTsim [31] is another cycle accurate simulator supporting the x86 architecture in both 32-bit and 64-bit. However PLTsim does not have any active development anymore, the latest news on their website is from the 17th of August 2009, and there is almost no activity in their mailing list. It was considered a high risk to use such an unsupported simulator, if we later ran into problems caused by bugs or lack of features.

The gem5 simulator is an open source full system simulator supporting multiple platforms, including x86 64-bit. The gem5 simulator is actively developed, as commits have been submitted to their repository recently, and there is frequent activity in their mailing list. The gem5 simulator was chosen for this project. The features of gem5 is further explained in the section 2.2.

As simulating a full system is extremely slow other approaches exist. A promising project by HP and AMD called COTSon [3] makes detailed simulation of x86 hardware. The timings of caches etc. can be assumed to be very accurate, because AMD, as a x86 64-bit CPU manufacturer and author of the 64-bit specification, has all possible reasons to make it a good resembling of physical hardware. However to gain performance, cycle accuracy was traded for speed, leaving the high accuracy timings behind. Depending on the accuracy needed, COTSon could be an interesting project, but will not be used here.

2.2 The gem5 Simulator

The gem5 simulator is a merge of the two projects M5 [4] and GEMS [18]. The M5 project was started at University of Michigan as a full system simulator to simulate large networked systems and explore designs of network I/O.

The Multifacet General Execution-driven Multiprocessor Simulator (GEMS) Toolset was started at University of Wisconsin. GEMS features a timing simulator of a multiprocessor memory system called Ruby, which was originally

run together with the commercial Simics simulator. Ruby implements a domain specific language called SLICC (Specification Language for Implementing Cache Coherence) which is used to model advanced cache coherency protocols.

The merge of M5 and GEMS into gem5, took the best assets from the two projects, taking the functional full system simulator from M5 and the memory modeling capabilities of GEMS. The simulator is still under heavy development and the concept of stable builds is not used yet, thus the latest development sources are used. The simulator does use a review system in order to ensure that commits do compile and successfully passes a small number of tests before they are added to the repository.

The gem5 simulator can run as a full system simulator, but also in another mode called system emulation mode (SE). In SE mode only the user space program is simulated and all system calls are handled by the simulator. Because the SE mode integrates with the Linux operation system, gem5 provides some extra Linux specific features such as loading the symbol table of the kernel.

The gem5 simulator supports simulation of multiple platforms. The platforms supported are Alpha, ARM, MIPS, PowerPC, SPARC and x86 64-bit. However the completeness of the implementations varies between platforms. The schematic presentation of the current status of the different implementations are shown in [11].

To build the framework, we need the simulator, a configuration of it and a system to simulate. In figure 2.1 a high level overview is shown of gem5 and the input it takes, the output it gives and the methods for interacting with it. To run the simulator it takes a Linux kernel, a disk image and a configuration file.

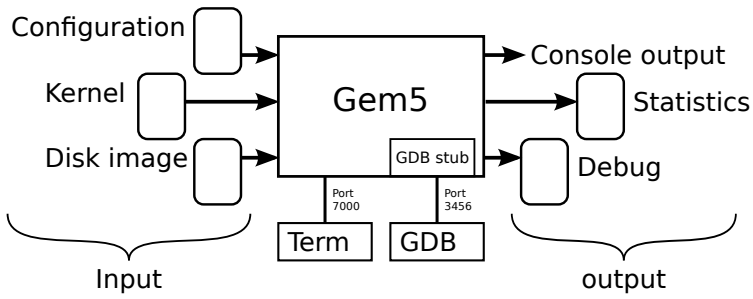

Figure 2.1: Overview of gem5.

The output of the simulator is console output, a statistics file and debug information dumped to a file. The statistics are collected during the simulation and dumped to a file in the end of the simulation. The debug output can be controlled using flags, which is further explained in section 3.7.3.

Beside having the input and output, we can access the simulated system while it is running by connecting a terminal or GDB[1] over specified network interfaces. Attaching GDB can be used to debug the simulated system. That means stopping the simulation, read and write registers and memory etc. The use and implementation of the GDB connection stub is further discussed in section 3.8.

2.3 Simulated System

The simulated system is a Linux system. The Linux distribution chosen for this project is Gentoo Linux, because of its flexible structure and the fact that everything can easily be compiled from source, which is a great advantage when debugging symbols are needed. For the base system Gentoo and Ubuntu distributions are recommended by the gem5 developers [9], for kernels, they propose fetching it directly frem `kernel.org`.

As mentioned above the gem5 simulator has special features for Linux systems, thus to simplify booting the Linux kernel is loaded into memory independently of the Linux disk. This makes it easy to change the Linux kernel used. So to use the simulator, a Linux kernel is needed and a Linux disk image.

2.3.1 The Linux Kernel

The first key component needed for the simulation is a compiled Linux kernel. The gem5 wiki [10] provides four different configurations of the Linux kernel for x86 64-bit. The Linux kernel configurations are for the following versions: 2.6.22-9, 2.6.22-9 (SMP), 2.6.25-1 and 2.6.28-4. Since the goal of this project is to provide the most recent software stack, newer versions of the Linux kernel is preferable. The supplied kernel configurations have been used as template for configuring newer versions of the Linux kernel.

[1] GDB: The GNU Project Debugger, a popular debugger and the standard debugger on many UNIX systems.

For other architectures like Alpha and ARM, gem5 supplies a patch queue. Meaning that the latest kernel is just added a number of patches which, if the patches can be applied, will create a working kernel. This feature might also be available for x86 64-bit in the future. But for now, we must build custom kernel configurations based on the ones provided.

The Linux kernel version used here is version 2.6.32-29. This version is still maintained by kernel developers at the time of writing. An even newer kernel would be preferred, but Linux kernel versions newer than version 2.6.32 uses the segment registers differently, causing kernel panics. Serious effort has been put into making the newer kernel work such as a fixing bad handling of segment registers [5], but more work is needed before a kernel newer than version 2.6.32 will work properly.

The configuration for kernel version 2.6.32-29 was made using the default kernel configuration[2] as a template. Then it was compared to the kernel configuration from version 2.6.28-4, which was known to work in the simulator. This was tedious work, because of the large size of the configurations, which range from 2500 to 3000 lines of options. None of this could be done automatically, as some configuration variables have changed name, some have disappeared and new ones have emerged.

The difference between the configurations was used to find devices not enabled by default. For instance the only disk interface currently supported is IDE. Support for IDE is no longer compiled into the Linux kernel as default, so it has to be enabled. Specifically the Intel PIIX IDE controller is used by the simulator. The output of the simulated system uses a legacy PTY device, which is not in the kernel by default either. There is still development going on in the gem5 project to support more devices, but for now using the IDE and PTY devices is the most stable solution.

In order to access the information about a program by reading from memory while the simulated system is frozen, swapping was disabled to avoid having data swapped out. This would introduce another layer of complexity, which is unnecessary when we can avoid it, by disabling swapping. If the machine however runs out of memory it will crash, but again it can be avoided by assigning enough memory to the simulated machine. In theory we could add almost an infinite amount of memory to the machine since it is just simulated, and if the amount of used memory in the simulated system exceeds the amount of physical memory of the host system, the host would just use swapping. However several problems has been reported for memory amounts larger than the amount of physical memory on the host and when

[2]Default kernel configuration is generated using "`make defconfig`".

the host machine starts swapping, performance would decrease significantly. Thus we used a reasonable large amount of memory of 1 GB, which is large enough for the software we use to run, but still less than the amount of memory on the host.

Finally gem5 supports loading kernels with debug symbols. This means that when the kernel is loaded the symbols are stripped from the executable and stored in a separate data structure. Then the executable are loaded into the memory of the simulated system and booted. The symbols can then later be used output more useful information about the simulated system.

2.3.2 The Disk Image

The simulated system is given by a Linux kernel and a disk image. Instead of doing the actual booting of the disk image, the simulator is started by loading the kernel into memory and then giving the simulator the entry point of the kernel. In this manner there is no need to fit a boot loader like GNU Grub or LILO onto the disk image, which would require configuration of the master boot record and might as well require an additional partition. Loading the kernel directly into memory also decreases the boot time.

Disk images used in gem5 are stored in raw image file. A raw image file contains the exact file system of a normal disk, just placed in a fixed size file instead of taking up an entire physical disk. This of course requires that the disk is smaller than the physical disk it is stored on. The raw disk image format is the most basic format of disk images, many other formats exists which often uses thin provisioning, i.e. they only save the data actually stored on disk in the image and not the entire allocated space. The use of a thin provisioning disk image format has been discussed in the gem5 mailing list [28], but it is still under development. The disk image file can be inspected using tools like `fdisk`, to extract information such as partition table and disk geometry. If you run out of disk space, the disk can be extended, and the file system on the image can be extended, if the file system supports extension.

The disk image is created by allocating a file with zeros and mounts the file as a loopback device. Then the loopback device can be partitioned using `fdisk`. When the disk is partitioned, the partition can be mounted as a loopback device and files can be placed onto the image. In figure 2.2 the disk layout is shown. In order to make the disk image work in the simulator, the disk has to convey to the standards of a regular physical disk as the BIOS will use the disk geometry. This implies that the disks geometry, that is the number of

cylinders, heads and sectors, has to be defined. The sector size has to be the value of 512 bytes as this is currently fixed in gem5[3].

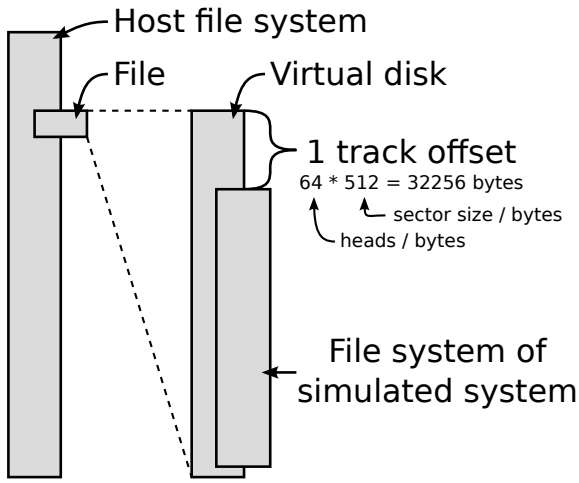

Figure 2.2: The disk layout.

The partition on a x86 system is found 1 track in, leaving the first track for the boot loader, thus if the system has 63 sectors and each sector is 512 bytes the offset on the disk will be 32256 bytes as shown in figure 2.2.

2.3.3 Installing Software on the Disk Image

With the mounted partition, we need to install the Linux base system onto it. The Gentoo Linux base system can be installed using the "stage-3" package. Stage 3 refers to the stage where the base system has been compiled and bootstrapped. When bootstrapped the new system can be used by changing the root into it. As access to devices are needed for using the image, the `/dev` and `/proc` file systems must also be mounted to corresponding directories of the new root. The commands necessary to mount and change root is shown in listing 2.1.

Listing 2.1: Mounting the Disk Image at `/mnt/`

```
mount -o loop,offset=32256 gentoo-6 /mnt/gentoo-6
```

[3]The value can be changed in the source code file `src/dev/disk_image.hh`.

```
mount -o bind /dev /mnt/dev
mount -o bind /proc /mnt/proc
chroot /mnt/
```

The software needed for the framework is a browser with a JavaScript engine. Beside installing that, all its dependencies also have to be installed. Gentoo Linux provides a package management system named Portage [8]. Using a package manager like Portage greatly simplifies software installation as dependencies are automatically resolved. Portage is installed by extracting an archive into the `/usr/portage` directory of the Gentoo Linux base system.

Software such as the browser might behave differently if run as root, because of security measures made to prevent root for using it. Thus to resemble a regular system, a user was created for running the browser and other user mode programs.

To test whatever was installed on the image, running up the simulator is quite time consuming. Instead a virtual machine monitor like KVM or VirtualBox can be used. Using a virtual machine monitor will run much faster because the emulation is hardware assisted, hitting almost native speed of the host if configured appropriately. KVM supported using raw disk images and loading the kernel from an external location, making it easy to use.

2.3.4 The X Window System

The framework needs to resemble the user experience as close as possible, thus it is not enough to compile the JavaScript engine itself, we also need the entire browser to simulate the behavior of the Document Object Model (DOM). The browser depends on the X Window System, a system for drawing on the screen. Thus in order to use the browser we will need an X server.

A common X server to use for testing on headless servers is Xvfb [30], which is just a minimalistic framebuffer. To test what is actually in the virtual framebuffer, the small program `xwd` can be used to capture a screenshot of the actual content of the framebuffer. This was used to verify that the system worked as expected.

Chapter 3

Configuration and Extension of gem5

Now the components to run the simulator are set, we need a way to extract information about the simulated system such that we can measure the timings of the executed JavaScript programs. This chapter is dedicated to the changes and extensions made to the gem5 simulator in order to extract this information.

We will first look into how gem5 is configured, then a description of how information is retrieved from the Linux kernel data structures and finally how trace output is generated.

3.1 Course of Simulation

Before getting into the configuration of gem5, it is appropriate with an overview of how the simulation will be carried out. As the simulator simulates the entire system, we will first have to boot it. After booting the operating system we can run the actual application we want to trace. The application is started using initialization scripts, which is handled by the `init` daemon. Gentoo Linux uses their own custom `init` daemon [8]. The initialization scripts are

used to start another script, which is injected into memory. Using this method the injected script can be changed from simulation to simulation without the need to modify the disk image.

The injected script will start the binary, which we are interested in analyzing. When inside the binary we reach the region of interest, from which we will save the trace. Details about tracing are explained in section 3.7.

Running simulations can take quite some time, therefore to reduce the time it takes to run a simulation, checkpoints can be used. Checkpoints enable the possibility to run multiple simulations without the need to boot up the system every time. The course of a simulation is shown in figure 3.1, which depicts the concepts just explained.

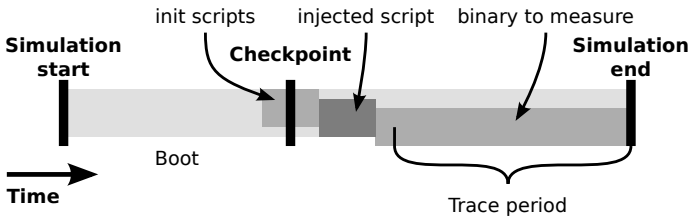

Figure 3.1: The course of a simulation.

Care should be taken when working with checkpoints, because it is not possible to create new processes during the simulation, which would require injecting instructions directly into the CPU. Instead the checkpoint is made halfway through the last initialization script, which then can load a custom script after the checkpoint. The default initialization scripts contain a lot of code to set up various things. To further speed up boot time the initialization scripts was trimmed to only hold the absolutely necessary code such as code for mounting the disk etc.

Finally when the simulation is over, statistics collected by the internal routines is saved to a file. The statistics to collect depends on the configuration of the simulator, hence most hardware devices do have counters, thus more devices attached will yield a larger collection of statistics. The amount of counters can be overwhelming, as a CPU configured with L1 and L2 cache, will output over 1000 lines of statistical output. Not all these many counters has been properly implemented and tested among the different CPU classes, thus some counters were found to yield false values. Especially the counters for cache misses did results in garbage values. Hence the counters should be used with care.

3.2 Configuration

Knowing how the simulation should be done, we need to configure gem5 in order to carry out the simulation. Running simulations in gem5 requires a configuration describing the system to simulate. The configuration needs to include all the hardware of the system such as CPU, memory, disk, etc. The configuration is not a regular static file, but instead a program written in Python. Having a Python program controlling the simulation have multiple advantages; it makes it possible to simulate complex systems without need to modify and recompile the C++ code and it enables the user to switch settings dynamically during the simulation.

The gem5 project ships with a default set of configurations files, which serves as a good template for configuring gem5. Many variables can be modified just using the command line arguments, but to avoid long command lines, the command line arguments should only be used to set settings which needs to be changed from simulation to simulation. Any other settings should go into the configuration files to avoid unnecessary clutter.

3.3 Starting the Simulator

An important detail to notice when working with the dynamic configuration is that it does matter in which order the different components are initialized, as the actual C++ objects are instantiated while executing the configuration files.

The execution of the default configuration files is shown in figure 3.2. It can be seen that command line arguments are parsed in both `main.py` and the given configuration script `<script>.py`, and later again `FSConfig.py` is called which also sets a lot of options. For parsing command line arguments the Python module `optparse` is used, which greatly reduces the amount of code needed to parse the options, but also allows the code to spread across the scripts. Thus every file has to be inspected to be sure the options are correct. As these difficulties have arisen as the complexity grew, the configuration can be dumped to a file for inspection.

When the configuration files are loaded the execution ends in the Python file `simulation.py` where it runs an infinite loop. The `simulation.py` Python script calls the C++ file `simulate.cc` where the actual simulation is done. The advantage of running the infinite loop in Python is the possibility

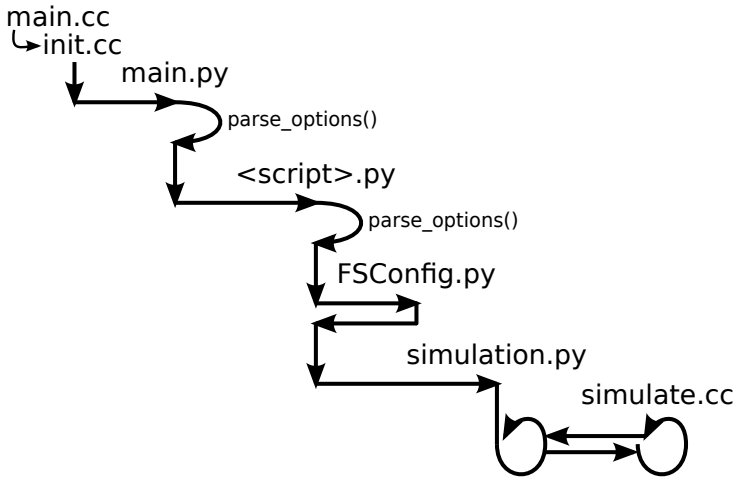

Figure 3.2: The execution path of the default configurations.

of running the simulator interactively through Python.

3.3.1 CPU Configuration

The gem5 simulator supports three classes of CPU models called SimpleCPU, Out-of-Order and In-Order. The SimpleCPU class of CPU models are the simplest, they are all in-order and has no pipeline. Out-of-Order and In-Order both have pipelines and execute instructions out of order and in order respectively. Currently only the SimpleCPU class are stable on x86 64-bit, the Out-of-Order CPU class works on Alpha and ARM, and the In-Order class is not yet working on any architecture. The current status of the CPU classes and types are shown on tabular form in [11].

Currently the SimpleCPU class holds two CPU models: AtomicSimpleCPU and TimingSimpleCPU. The AtomicSimpleCPU is the most basic CPU model and also the fastest. All memory accesses are done atomically, that is, the accesses are done instantaneously, thus no timings needs to be recorded. TimingSimpleCPU on the other hand have timed accesses of memory, which makes it possible to simulate queuing delays and resource contention. The system is modeled by connecting a bridge between the CPU and the memory bus and then specifies a delay on that bridge.

A common technique to speed up the simulation is to switch the CPU type during the simulation. This relies on the fact that a CPU like AtomicSimpleCPU works faster, because it does much less bookkeeping than other CPU models, which have complex memory systems to keep track of. This feature however does not currently work on x86 64-bit architectures, thus in all simulations done here the same CPU type is used during the entire run of a simulation.

3.4 Development Activity

The gem5 simulator is under heavy development, thus many changes are added to the gem5 repository as illustrated in figure 3.3[1], which graphs the commits added to the gem5 repository at daily basis. Due to the heavy activity this project pulled changes once a week from the gem5 repository. This project used the same source control system as the gem5 project, namely Mercurial. This made pulling of changes from the main gem5 repository easy, because it is a distributed version control system, meaning merging different repositories is a common task. The code added in this project was kept on top of the gem5 changes by using the `rebase` extension for Mercurial. This works by removing my changes into a patch queue, apply their latest changes and reapply my changes again afterwards. In this manor the revision tree was kept much cleaner, and things were less prone to break because of code submitted long time ago. This also made it easier to discuss problems in the gem5 mailing list, because my own changes could easily be removed and reapplied to conclude wherever the bug was caused by my code or not.

On January 28th a major change was submitted [6] to the main gem5 repository to merge the full system mode and system call emulation mode. This changed allowed running both modes with the same binary. As this update made major changes to the source, there has not been pulled from the repository since that date as the major changes might break our code.

3.5 Special Instructions

Before the merge of M5 and GEMS, M5 contained some special instructions for controlling the simulator from inside the simulated system. For legacy reasons these instructions are still prefixed M5. The instructions can be used

[1] Generated using the Activity Mercurial extension.

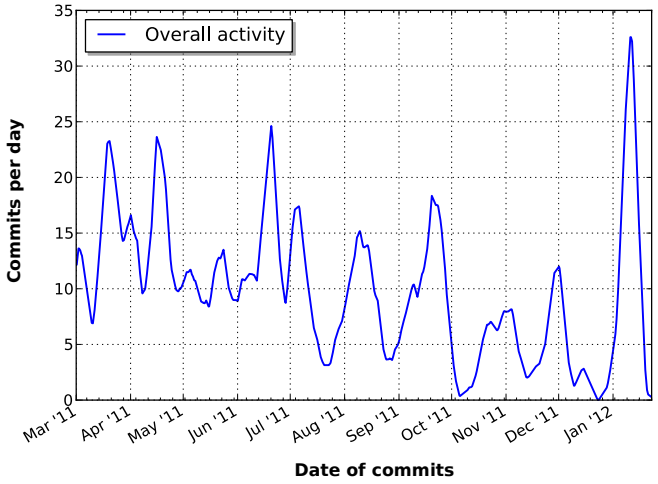

Figure 3.3: Daily commit activity in the gem5 project.

to exit the simulation, reset and dump statistics, create checkpoints, and read a file into memory. The special instructions can be used in two ways, either by compiling them into your own binaries or by using a utility binary called `m5`. The `m5` binary makes it very simple to measure an entire program, by calling reset statistics before starting the program to measure and finally dump the statistics when the program is done executing.

3.6 Inside Linux Kernel Structures

Getting information about the underlying Linux system is a necessity for distinguishing the different programs running, and thus to accurately say something about what code is currently being executed. Hence to identify the lines of code being executed and their timings an understanding of the kernel structures of the Linux kernel is needed to be able to extract this information from it. First we need information to identify a process, either by a unique identifier or the absolute path of the binary file from which it is spawned and information about the memory maps owned by the process. From that information we can tell which binary is executing and what code it is executing. The details of identifying a process, get the name of the binary and read the memory map is described in their individual sections below, but first we need to see how to read from the memory of the simulated system.

3.6.1 Reading from Memory

From the simulator the memory is accessed using physical addresses, but the program running inside the simulated operating system will use virtual addresses. Thus we need to translate virtual to physical addresses. The simulator has a function to scan the page table of the simulated system and translate the virtual address into the physical. The translation is handled in the architecture-specific code, since the handling of the page table is highly dependent on the CPU architecture. The information we are fetching below, is only for analysis purposes and is not part of the actual simulation. Thus the memory accesses should be done in such a fashion that it does not affect the timing of the simulation.

The following sections is primarily based on information about the Linux kernel gathered from [7] and the Linux kernel source code itself. The Linux source code is the best resource because other documentation is often outdated.

3.6.2 Identifying the Currently Running Process

A process in a Linux system is uniquely identified by a number called the Process ID or PID for short. This number is on most Linux systems in the range 0-32767. The number is assigned sequentially to newly created processes. The default number of 32768 processes is usually enough for most systems, however systems with need for more have the possibility to increase the number.

In an operating system code runs either in kernel mode, having access to all memory, or in a protected mode called user mode where only certain memory addresses are accessible. In this way multiple programs can run in user mode without compromising the behavior of other programs. To switch between the two modes the kernel asks the CPU to switch mode. The user mode program will then execute until it either needs resources controlled by the kernel or an interrupt occurs. Entering kernel mode is called a `SYSCALL` and entering user mode is called a `SYSRET` [1]. On x86 64-bit specialized registers are used to speed up the mode change. When a `SYSCALL` is made the kernel stack pointer is restored from the model-specific register (MSR) address `C000_0102h` to the hidden portion of the `GS` selector register. When making a `SYSRET` the kernel stack pointer is saved in the MSR, where the user mode process cannot access it. To switch between kernel mode and user mode, and vice versa, a special instruction called `SWAPGS` is used. So to check wherever we are currently in

kernel mode or not, we just need to check the value of the GS or the MSR register.

When we are in user mode, we need to find the PID of the currently executing process. The data in the Linux kernel is stored in structures, which is a collection of data entries stored sequentially in memory depending on alignment and padding options. So to retrieve the information about the currently executing process we need to access the structure holding information about the current process.

Information about processes is stored in a structure called task_struct. Because this structure is quite large, the information about the currently executed process is stored in a smaller structure called thread_info, which resides in the CPU private area. That is a specific part of memory only a single CPU has access to, in order to avoid synchronization errors when having multiple CPUs. The thread_info structure fits in lower part of the kernel stack, which has the size of two pages, with a page size of 4096 bytes. The kernel stack is illustrated in figure 3.4. This makes the thread_info structure easily accessible, since the kernel stack pointer is stored in the RSP register, the value of the RSP register can just be applied a mask to get the address of the thread_info structure.

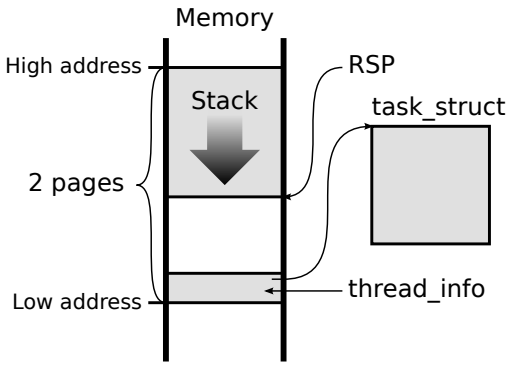

Figure 3.4: Location of the task_struct structure on the kernel stack.

With a pointer to the first kernel data structure, we can access other structures. In order to browse the kernel structures we need the offsets inside the structures to find the right pointers to the other structures. These offsets can be hard to find, because the structures can contain many members and due to alignment, gaps might appear in memory. To find the right offsets, additional code was compiled into the kernel, because the sizes are known compile time,

these sizes could then be read from the debug information. Sensible names made it easier to find the symbols in the binary later. Using the `offsetof` macro the offset of a member in a structure can be found. An example of the code to find the offsets and sizes of the structures is shown in listing 3.1. The last line in listing 3.1 shows how to get the size of a member inside a structure, which is useful for getting the size of a structure embedded inside another structure.

Listing 3.1: Example code for finding offsets.

```
const int thread_info_size = sizeof(struct thread_info);
const int thread_info_task = offsetof(struct thread_info, task);
const int vm_area_struct_vm_flags_size =
  sizeof(((struct vm_area_struct*)0)->vm_flags);
```

We were in pursuit of the PID which is stored in the `task_struct` structure. With the PID we can deduce which executed instructions belong to which process.

3.6.3 Locating the Binary

However we do not know the PID of the executable we are interested in. This is because the PID is first assigned to the process when it is created and there is no method for forcing a specific PID. A way to handle this problem is to use the absolute path of the executed binary, since we know this before we start the simulation and it is fixed throughout the simulation.

Finding the absolute path to the binary requires some browsing through the kernel data structures. From the `thread_info` structure there is a pointer to the `task_struct` structure. The `task_struct` structure has a pointer to the memory descriptor structure (`mm_struct`). The memory descriptor structure holds a pointer to a doubly linked list of memory regions (`vm_area_struct`), which are the memory regions belonging to the process. If a memory region maps to a file, a pointer is stored in the memory region structure to the file structure (`file`). Finally a `path` structure is stored within the file structure, where the `path` structure holds a pointer to a `dentry` structure. The `dentry` structure holds the name of the file and a pointer to a parent `dentry`, which represents the parent directory of the file. The last `dentry`, also referred to as the root (`/`), has the parent pointer pointing to itself. Inside the `dentry` structure is `qstr` structure which holds a pointer to a `char` array with the name of the binary.

A schematic view of how the structures were browsed is shown in figure 3.5. On the figure is also shown the names of the pointers as they appear in the Linux kernel source code.

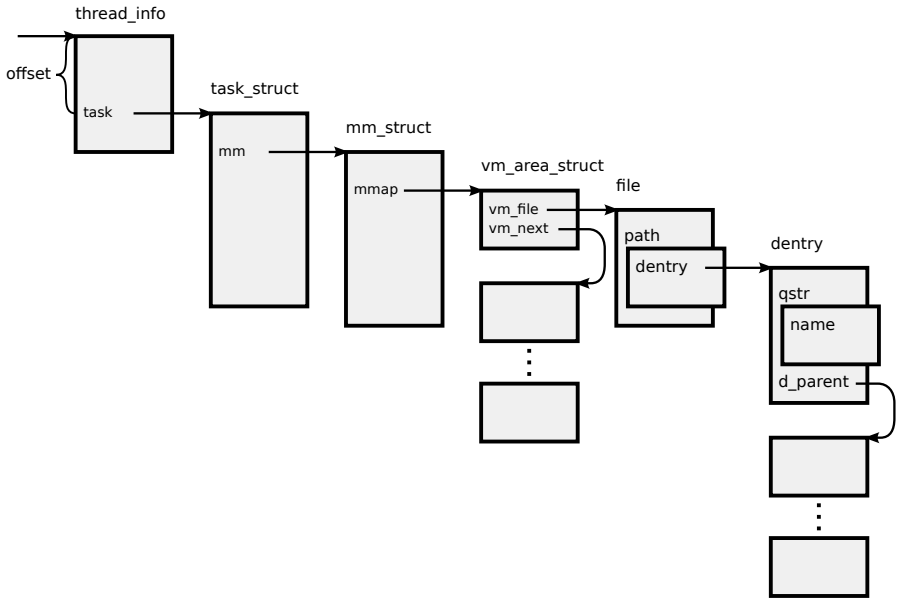

Figure 3.5: Browsing the kernel data structures.

One might think that more processes could exist where one uses the binary and another loads the binary as a library and then the wrong process was chosen due to the fact that the traversal of memory areas will chose the first who has a pointer to a file with the right name. However a binary with a main function cannot be loaded as a library, thus this scenario cannot happen.

3.6.4 Memory Map

A key in knowing what code is being executed, is where it is located in memory. As programs make heavy use of libraries. Consider the program `sleep.c` shown in listing 3.2.

Listing 3.2: sleep.c

```
#include <stdio.h>

void loop() {
  while (1) {}
}

int main() {
  printf("started..\n");
  loop();
  return 0;
}
```

The program prints out a line and then goes into an infinite loop. The line printing text to the screen makes use of the standard library, thus we know that at least the standard library (or libc) will be loaded dynamically.

On Linux systems the map of memory allocations by a program, can be seen in the `proc` pseudo file system in the file `/proc/`*pid*`/maps`, where *pid* is the PID of the inspected process. Listing 3.3[2] shows the content of `/proc/`*pid*`/maps` for the `sleep.c` program. The reason for using an infinite loop is that we needed a program that did not stop immediately before we could save the content of the `/proc/`*pid*`/maps` file.

Listing 3.3: /proc/*pid*/maps

```
00400000-00401000 r-xp 00000000 08:01 1280443 /home/user/sleep
00600000-00601000 r--p 00000000 08:01 1280443 /home/user/sleep
00601000-00602000 rw-p 00001000 08:01 1280443 /home/user/sleep
7f6e2202a000-7f6e221b4000 r-xp 00000000 08:01 3935978 */libc-2.13.so
7f6e221b4000-7f6e223b3000 ---p 0018a000 08:01 3935978 */libc-2.13.so
7f6e223b3000-7f6e223b7000 r--p 00189000 08:01 3935978 */libc-2.13.so
7f6e223b7000-7f6e223b8000 rw-p 0018d000 08:01 3935978 */libc-2.13.so
7f6e223b8000-7f6e223be000 rw-p 00000000 00:00 0
7f6e223be000-7f6e223df000 r-xp 00000000 08:01 3935965 */ld-2.13.so
7f6e225b3000-7f6e225b6000 rw-p 00000000 00:00 0
7f6e225db000-7f6e225de000 rw-p 00000000 00:00 0
7f6e225de000-7f6e225df000 r--p 00020000 08:01 3935965 */ld-2.13.so
7f6e225df000-7f6e225e1000 rw-p 00021000 08:01 3935965 */ld-2.13.so
7fff1fa84000-7fff1faa5000 rw-p 00000000 00:00 0          [stack]
7fff1fbbf000-7fff1fbc0000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Each line holds the following information: The start and end address in memory, the permissions of the stored data, the offset in the binary, the device, the

[2] Formatted to fit the page, thus whitespaces has been removed and the path `/lib/x86_64-linux-gnu` has been replaced with "`*`"

inode[3] and finally the path to the binary, which the code belongs to. Memory allocated dynamically, either by the executable itself or a library, does not belong to a file, since no binary has been mapped into memory. This is all the information we need to map memory to code. What is surprising, is how many allocations are needed for such a simple program, which only includes a single file from the standard library.

However, we cannot read the `proc` pseudo file system, because it is a pseudo file system, the files are not actually there, but rather an abstract representation of a program. Instead we will have to extract the information in the same way as the program behind the `proc` file system, by reading the kernel data structures. From the previous section we have a pointer to the `task_struct` structure. This structure holds a pointer to the memory descriptor structure (`mm_struct`), which in turn holds a pointer to the doubly linked list of memory area structures (`vm_area_struct`). The browsing of structures can be revisited in figure 3.5. Each memory area structure corresponds to a line in listing 3.3, thus it holds information such as start and end address, permissions and a pointer to the file structure as described in the previous section.

The permission bits stored with each memory region are stored in a `vm_flags` structure. There exist far more flags than the four used here, they are all well described in the kernel source code file `mm.h`. The four flags we need are the read, write, execute and shared/private flag i.e. `VM_READ`, `VM_WRITE`, `VM_EXEC` and `VM_MAYSHARE`.

The output format of the memory map was kept the same as that of the `proc` pseudo file system to make it easier to verify the correctness of the code.

3.7 Tracing

To analyze the execution of the simulated system, we need to save a trace. A trace is a full listing of all the instructions the simulator has executed. For generating the traces we need the following: A method for controlling when to start and stop the tracing, and a way to control what output is written to the trace file.

When creating the traces, we are interested in making the traces as short as possible, due to the fact that the files will grow rapidly in size. To minimize the file size, we want to start and stop the tracing as accurately as possible,

[3] The *index-node* identifying the file in the file system.

in this way we can wait to start the tracing until we are at the region of interest. The standard method of controlling events in gem5 is by time, which does not suit our needs, as we do not know the time in advance and that is what we would like to measure. We would rather like to control the trace output using locations in the executed binaries. This can either achieved by compiling special instructions into the binary as explained in section 3.5 or by using a breakpoint based approach. In the breakpoint approach, the simulator loads the breakpoint before simulation and stops, when the given program counter (PC) address is reached. The functionality for setting breakpoints exists already in the simulator, but has to be extended to allow controlling the trace facility. This will be explained in the following section.

3.7.1 Breakpoints

The simulator already supports setting breakpoints, a module used by the GDB stub (further explained in section 3.8). The breakpoints are set by creating a new event object. The breakpoint object inherits from a more general class called `PCEvent`. The `PCEvent` class is used to schedule events based on the program counter, which is exactly what our traces should be scheduled on as well. So a new class called `TracePCEvent` was created, which also inherits from the `PCEvent` class. The class definitions can be reviewed in appendix B.1 where a subset of the header file is shown.

As different processes run in the same virtual address space, a specific PC address might be triggered by the wrong executable. To avoid this, the `TracePCEvent` verifies the executable when triggered, using the technique described in section 3.6.3.

Creating a breakpoint at a specific PC address is currently only supported using the SimpleCPU class of CPU models, thus additional changes will have to be made, to make breakpoints work on other CPU types such as the Out-of-Order model. Furthermore the breakpoints for other architectures than Alpha did not work before a bug in handling the PC address alignment was fixed [14].

3.7.2 Breakpoint Insertion

Being able to control when to start and stop the tracing, the next thing we need to control is what is actually saved in the trace files. To make gem5 run as described in section 3.2, the breakpoints controlling the tracing should be

inserted automatically using the simulator configuration files. Hence we need a way to insert the breakpoints in the Python configuration. In order to access the main objects of the simulation from the Python configurations, a wrapper between C++ and Python is needed.

In gem5 all objects shared between C++ and Python derives from the SimObject class. In this way parameters can be defined from Python, making them flexible. The state and behavior of the object is implemented in C++ for speed. To make the Python interact with the C++ objects interfaces are used. These interfaces are generated using SWIG [27]. SWIG works by specifying an interface file with ANSI C or C++ declarations and some specific SWIG directives, which can then generate the wrapper files for accessing the C++ code from Python.

The creation of breakpoints using `setPCTrace` can first be done when the CPU object has been initialized. From figure 3.2 this is done in the file `simulation.py`.

An extension to the method had to be made, in order to support the execution of WebKit because it executes as a dynamically loaded library. The location of the library in memory is first known when the program using it is started, thus the PC address is only available when the program is started. The extension is made such that the insertion of the breakpoint is done in two steps. First an additional placeholder breakpoint is placed in the beginning of the program and when that breakpoint is triggered, the library breakpoint can be calculated and inserted. The library location is calculated using the offset within the library added with the address of the library in memory, which is found using the memory map explored in section 3.6.4. The two step process is illustrated in figure 3.6 where the additional breakpoint at `addr1` from binary `x` is used to insert the breakpoint at `addr2` in the library `y` using the offset `offset` found when the first breakpoint was triggered.

3.7.3 Debug Flags

The gem5 simulator already has a flexible system for outputting information about the internals execution. This system consists of a printing method and some defined variables to control them. The variables are set based on the flags passed to the simulator through the command line arguments. These flags can however also be controlled programmatically within the simulator. The flags are very efficient for debugging the simulator, as you will only receive output on the specified flags. Flags are made specific for different parts of the simulator.

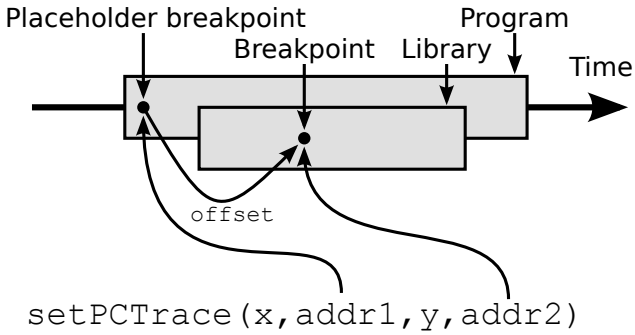

Figure 3.6: Breakpoint insertion in a dynamic library.

The implementation of the flags are generated using a custom Python script build into the SCons[4] build system. This means the implementation of the flags, which is rather simple, will be created at compile time. The flags consists of a string (the name of the flag), which then should be used when creating the print statements in the code. It is also possible to combine more flags by creating compound flags. For the trace output a new compound flag named `ExecTrace` was added which is compound of the options:

- `ExecEnable`, enables execution trace output (needed for all following to produce output).
- `ExecTicks`, prints the tick number.
- `ExecEffAddr`, prints the address of the instruction.
- `ExecMacro`, prints macro instructions as opposed to micro instructions [12].
- `ExecFaulting`, prints faulting instructions.
- `ExecUser`, prints instructions run in user mode.
- `ExecKernel`, prints instructions run in kernel mode.

The amount of options might seem overwhelming, but as the output is massive, having very granular control makes sense.

[4] A build tool similar to `make` and `autoconf`, written in Python.

3.7.4 The Trace File Format

The gem5 simulator implements certain steps to execute an x86 instruction. First the instruction need to be interpreted from a stream of bytes, which is done by the predecoder, this is necessary because x86 instructions have no alignment. When predecoded the instruction has a uniform format that the gem5 CPU can understand. It is either executed directly or broken into several steps which can then be executed. When we save the trace of what is going on, we need a hook into the decoder and print the decoded instruction. As the trace is useful to compare output from the simulator with the assembler code of the executable, the output should be the decoded macro instructions.

The trace file format used here is a simple ASCII format, since using it for debugging is much easier when human readable. The trace file format is shown in figure 3.7.

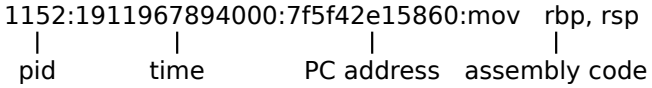

Figure 3.7: Trace file format.

The trace file format is a modified version of the debug output, which is build into gem5. Furthermore the trace file format have been trimmed to keep file size minimal. There is no reason to keep the two `0x` characters in front of the hexadecimal address, since we know it is in hexadecimal. There is also removed a lot of white spaces, which was there to improve readability, but in order to provide easier and faster parsing, they were removed.

ASCII files are very inefficient to store in terms of file size, to minimize the space used the output is saved in GNU zip format. This is achieved by piping the output of the simulator through `gzip`. The advantage of this is less storage used and as computers of today have plenty of cores to run the simulation, which is single threaded, there are plenty of free cores to compress the trace while preserving simulation speed. When analyzing the trace files, the files should be uncompressed before processing, which is done by piping through `gunzip`.

Alternatively a binary format could have been used, which would decrease saving and processing speed. However a binary format was not chosen as it is much harder to debug.

3.8 GDB Support

To aid the development and extension of gem5, the remote debugger GDB is supported. However GDB support was only implemented on Alpha and ARM architectures, and not on x86 64-bit. To implement the GDB support for x86 64-bit the network code from Alpha and ARM architectures could be reused. The architecture-specific code such as reading registers, writing registers, address translation, adding and removing breakpoints were implemented according to the remote GDB protocol [26].

GDB support was very valuable because it made it possible to stop the execution of the simulator, inspect the memory and continue execution. This was helpful for browsing the Linux kernel data structures described on section 3.6.

Chapter 4

Tracing JavaScript Programs

The previous chapters was used to set up the framework, this chapter will look into the capabilities of it. This means examining the execution of JavaScript programs, which will be carried out by the browser, or more precisely the JavaScript engine in the browser. To analyze the execution, we will save a full trace of the execution using the tracing capabilities implemented in section 3.7. Finally we show the capabilities of the framework by generating a fine-grained trace of a small JavaScript program.

Initially we will look at the choice of browser and JavaScript engine. Then the identification of breakpoints to control the trace will be explained. We will go through the tool to analyze the trace files. Then we will make a small case study of WebKit and a tool named JSC.

4.1 Web Browsers and JavaScript Engines

Today the market of browsers is dominated by three browser engines: Trident, WebKit and Gecko. The Trident engine is made by Microsoft used in their Internet Explorer browser series, WebKit is the engine used in Safari by Apple

Inc. and in Chrome by Google Inc., and Gecko is the engine in Firefox by the Mozilla Foundation.

The browser chosen here is WebKit. WebKit is currently considered to have a slight advantage over Gecko, according to the benchmark in [22], where the major browsers have been compared using JavaScript benchmarks, page loading times and HTML5 features. Trident is closed source, which is why we did not choose it, as access to the source code is essential for understanding the traces.

The JavaScript engine in WebKit is independent of the browser core, thus it can be changed. For instance Chrome consists of WebKit, but with their JavaScript engine called V8. The default JavaScript engine in WebKit is JavaScriptCore also called SquirrelFish and SquirrelFish Extreme, or Nitro and Nitro Extreme in Apple terms [15]. The JavaScript engine used here is JavaScriptCore.

4.1.1 Finding the Entry and Exit Points

The benchmarks of JavaScript programs should be carried out, such that only traces of the relevant parts of the code will be saved. To control the trace breakpoints should be inserted at the start and end of the relevant code.

In order to start and stop tracing based on a specific address inside a binary, we need the name of the binary and the address. By compiling debug information with the program, we are able to convert a line of code into an address. This can be done by reading the debug symbols from the binary[1] using tools like GDB or `objdump`. However because WebKit acts as a dynamically loaded library we need to look further into the stubs used to connect the program with the WebKit library.

The problem with dynamic loaded libraries is that our program needs the addresses of the functions to call. Our program has its own virtual address space, thus the libraries cannot run in the same address space, as they would have to include absolute virtual addresses. Instead the libraries need position independent code. To solve the problem of having position independent code, a global offset table (GOT) holds absolute addresses private to the binary. In that way the GOT can be used to translate position independent references into absolute addresses. However often we would like to call functions in the library, which is done through the procedure linkage table (PLT). The PLT

[1] Assuming the binary is in the ELF format with the debug information in DWARF format inside.

works much like the GOT, but instead of translating references, PLT translates function calls to absolute addresses. Thus the compiled program will contain a small PLT stub for each function call to a library.

The concepts of GOT and PLT are specified in the System V Binary Interface [19]. Thus it can be used on all systems supporting the interface, e.g., Linux systems, as used here.

4.2 Analyzing Trace Files

As complete trace files are saved, these are useful when extremely detailed information is needed. However to get more general information from these files, small Python tools have been developed to summarize information from them. The tools are made to parse the trace file using regular expressions to match the trace file format as described in section 3.7.4. Besides loading the trace files, loading the memory map has also been implemented, to be able to map code to a binary.

Each line in the trace file consists of a PID, a timestamp, an address and the assembly code. When inside the kernel, the PID is simply omitted. The time spent on each instruction is calculated by subtracting the timestamp from the previous line.

The line of source code a given instruction maps to, can be obtained using the GNU Binutils tool `addr2line`, which, given a binary with debug symbols and an address, will print the filename and line number. Using the memory map we can determine both the binary and calculate the right address based on the offset.

Obtaining the line of source code from a given address is not always precise or possible. If the binary is compiled with optimizations, some line numbers might not be correct as the source code is changed, because variables can be removed due to optimizations, control flow may be changed or code might be moved out of loops etc. In case the address is in the PLT, the source code line cannot be determined, as the PLT code is appended by the linker and hence does not correspond to a line of code.

One line of source code can correspond to multiple instructions. When counting the number of times a given line of source code has been executed, the instructions should be grouped together, instead of counting each instruction as an execution of a line of source code. This can be achieved by aggregat-

ing instructions laying side by side which maps to the same source code line, however it should be ensured that the PC address is increasing as a source code line could call itself and then only be counted as one. This would be the case for line 4 (the infinite loop) in `sleep.c` shown in listing 3.2.

4.3 Tracing WebKit

WebKit depends on a graphical user interface (GUI), however WebKit is implemented to use various GUI frameworks to build the GUI. Since GUI frameworks are often platform dependent[2] it makes sense to support different GUI frameworks in order to be available across platforms. From the GUI framework creators point of view it is also an advantage to support WebKit, because WebKit as a widget extends the feature set of the GUI framework significantly.

For this project we have chosen the GTK+ GUI framework because of its native execution on Linux. The choice between GTK+ and Qt was only a matter of preference, as GTK+ has fewer dependencies.

The code executed in WebKit is a simple JavaScript program computing the first 10 Fibonacci numbers. The program serves two purposes, it keeps the trace file short and it does a calculation, if no calculation was done the program might just be skipped due to optimization. The JavaScript program is embedded in HTML to load in the browser. The source code is shown in appendix B.2.

The breakpoints for loading the page was set in the GtkLauncher[3] application. The load call was identified in the source code and later found using the debug output. GtkLauncher calls the function `webkit_web_view_load_uri` in the WebKit library using the PLT as explained section 4.1.1. The address was found to `0x403fc6` in the disassembled output as shown in listing 4.1.

Listing 4.1: Disassembly of GtkLauncher (start)

```
403fc6: e8 5d ea ff ff  callq  402a28 <webkit_web_view_load_uri@plt>
403fcb: 48 8b 45 c8     mov    -0x38(%rbp),%rax
```

The stopping condition was less obvious, because WebKit launches a separate thread for loading the web page. As the separate thread is executing the

[2] Many exceptions exist such as Qt, GTK+ etc., but a framework like GTK+ has Linux as its native platform.

[3] A simple application using the WebKit GTK+ widget.

JavaScript, simply triggering the line after execution would end the trace before the JavaScript execution might even have started, i.e. the calculation of Fibonacci numbers. Thus to stop execution, an additional line of JavaScript was added to close the browser window and the stopping condition could then just be placed on the other side of the GUI main loop. The stopping address was found to be `0x40400a`, which is right after the exit of the main loop (`gtk_main()`) as shown in listing 4.2.

Listing 4.2: Disassembly of GtkLauncher (stop)

```
404005: e8 9e e6 ff ff  callq  4026a8 <gtk_main@plt>
40400a: b8 00 00 00 00  mov    $0x0,%eax
```

4.3.1 WebKit Results

When the simulation was run, it did not stop although debugging information verified that the breakpoints were successfully inserted. The problem was identified to be missing instructions in the simulator. This was due to the fact that the simulator does not fault on unknown instructions, it simply skips them. Thus the instructions skipped were necessary for WebKit to progress, leaving the execution in an endless loop. Debug information showed that the following instructions were skipped: `fsincos`, `fnstsw`, `fldpi`, `fadd`, `fxch`, and `fprem1`. As the names of the instructions suggest, gem5 is lacking floating point operations.

4.4 Tracing JSC

As WebKit is a large code base depending on several libraries such as X, there is a reason why the JavaScript engine is decoupled from the core of the browser. First, as previously explained, it is possible to change it, another advantage is that it can be compiled individually from the browser core. This greatly decreases build time and makes testing much faster. To test the engine a small program was made by the WebKit team to act as a simple wrapper to execute JavaScript programs. This program is called JSC, short for JavaScriptCore, the name of the JavaScript engine itself. The source code of JSC is stored together with the JavaScriptCore source code in the WebKit repository. JSC works as a small program using the JavaScriptCore library, thus the two step breakpoints should be used for this as well. Because JSC depends only on the

standard library and the JavaScriptCore library, tests have shown that complete execution of JSC is possible in the gem5 simulator.

The entry and exit point of JSC is found to be the address `0x4109f8` and `0x410ae4` respectively using the same technique as described in section 4.3.

To illustrate the tracing capabilities of the framework the Fibonacci number generating JavaScript program from section 4.3 has been used. However the version used for JSC is a JavaScript only version, the source is shown in appendix B.3.

4.4.1 JSC Results

The trace output yields a total of 9579204 instructions, the time simulated was 0.029 seconds, and it took 162.92 seconds on the host computer to carry out the simulation. The trace output generated is massive when remembering that all we executed was 12 lines of JavaScript code.

What is interesting to see is what code has been executed. A measure for that is to group the code by binary. This has been done in figure 4.1, by grouping the 9579204 lines of assembler to chunks belonging to each binary. The graph in figure 4.1 does not show a very clear grouping, due to the fact that we are trying to draw a total of 113272 chunks distributed on the different binaries. The gray color on the figure is an outline to emphasize where things have been executed, where the colors are actual execution time. Without the outline it would not be possible to see the execution in the e.g. libstdc++. What might be surprising is how much execution switch back and forth between binaries. This is exactly the information that would otherwise be hard to collect, because profiling using sampling is very error prone to these frequent changes of binary where only a few lines of code are executed.

A subset of the trace shown in figure 4.1 has been plotted in figure 4.2. In figure 4.2 it can be seen how there is a lot of jumping back and forth between the JSC binary and the JavaScriptCore library, which is primarily caused by the PLT in JSC. Another noticeable observation in this example is how the kernel is triggered by the libc library, and before returning, some code in the unknown area is executed.

Essentially, what can be concluded from the small examples shown here is how extreme the amount of assembler code is. This is the reason why we do have higher level tools, as information processing at the assembler level is

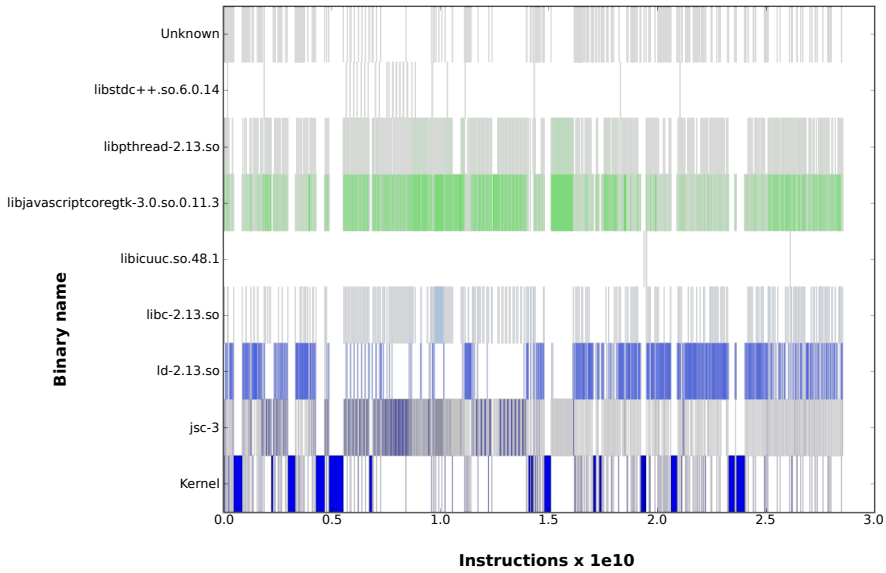

Figure 4.1: Execution time distributed on binaries from `fib.js` trace.

very time consuming. However the detailed view has advantages over sampling techniques, because of the higher precision. Where sampling techniques require several runs to yield reliable results, the simulation results will include every detail.

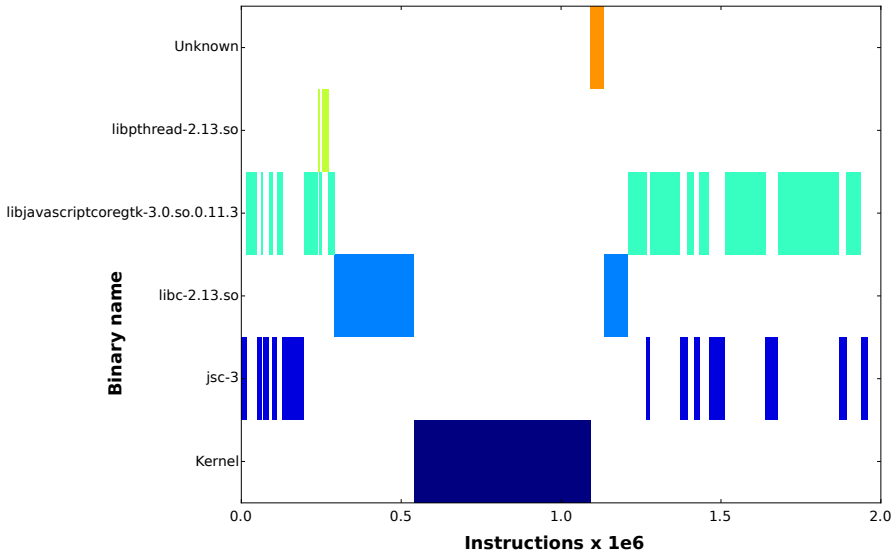

Figure 4.2: Execution time distributed on binaries from trace subset.

Chapter 5

Performance Comparison

This chapter analyses the performance between different CPU models available in the SimpleCPU class of CPUs. The performance measured is used to compare the different CPU models with each other and to execution on physical hardware. The tests executed are some of the most popular JavaScript benchmark suites. The goal is to test the different CPU models in the simulator and find the best match compared to execution on physical hardware.

5.1 Test Setup

The configuration used to produce the results of this chapter is shown in table 5.1

CPU frequency	2.00 GHz
Memory	1 GB

Table 5.1: Specification of the simulated system.

The SimpleCPU class holds two CPU models, AtomicSimpleCPU and TimingSimpleCPU, as explained in section 3.3.1, however it is possible to extend

the TimingSimpleCPU with different cache hierarchies. We have chosen to use four different CPU models, the AtomicSimpleCPU (Atomic) and three differently configured TimingSimpleCPUs. The different configurations varies in CPU complexity, thus we will use a TimingSimpleCPU with no cache (Timing), a TimingSimpleCPU with L1 cache of 32 KB + 32 KB (Tim L1), and a TimingSimpleCPU with both L1 and L2 cache of 2 MB (Tim L2). The L1 cache is split in data and instruction cache respectively. The CPU models and their timings is shown in figure 5.1.

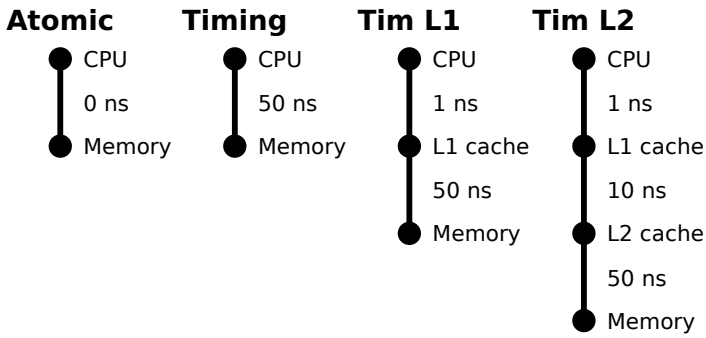

Figure 5.1: Visual comparison of simulated CPU models.

As the SimpleCPU used in the simulator is in-order it produces deterministic results, thus running multiple runs will yield the exact same result. Therefore the tests have been run only once.

Besides recording the time used inside the simulator, we also recorded the time used by the host computer, in order to analyze the efficiency of the simulation method. However the times recorded from the host computer might not be as accurate as the simulations are only run once and the load on the compute server might vary over time.

5.2 Physical Hardware

To compare the results with physical hardware the machine with the specification shown in table 5.2 was used. The L1 cache is split between data and instruction cache as in the simulated CPU. The timings measured on physical hardware were captured using the `time` utility, and the measurements made are an average of 10 runs.

CPU	Intel(R) Core(TM) i5 CPU, M430 (2 cores) [16]
CPU frequency	2.27 GHz
L1 Cache (per core)	32 KB + 32 KB
L2 Cache (per core)	256 KB
L3 Cache (shared)	3 MB
Memory	4 GB

Table 5.2: Physical hardware specification.

The CPU is dual-core and WebKit is a multi-threaded application, but the execution of JavaScript is single threaded, which is why it makes sense to test a multi-core physical CPU against a single-core simulated CPU. Future plans for WebKit2 [24] contain a new threading model, which exploit threading better, but there is no primitives in JavaScript to support multi-threading, hence execution of JavaScript will remain single threaded.

For the `time` utility to work, the browser should finish its GUI main loop when JavaScript execution is finished. Thus the same solution, where the JavaScript command `window.close()` is called in the end of the program, as in section 4.3 is used here.

5.3 Benchmarks

In order to compare the performance of the different CPUs, we need different tests to run. As we are focusing on JavaScript execution, different JavaScript benchmarking suites will be used. The benchmarks used here are the popular JavaScript benchmarks, V8 Benchmark Suite and SunSpider, both run using the JSC tool. We also considered using Dromeao and Kraken, but as they require access to the DOM model, they would require WebKit to run.

Since the tests run on physical hardware use the `time` command to record the timings, there are no need to use breakpoints on the simulated system, since we can just use the special instructions explained in section 3.5 using the M5 binary.

5.3.1 V8 Benchmark Suite

The V8 Benchmark Suite [13] is developed by Google Inc., in connection with their development of the V8 JavaScript engine. For the benchmarks carried out here version 6 of the benchmark suite was used.

The V8 Benchmark Suite consists of a set of individual tests. The following list briefly describes each test. The tests are listed in the order, in which they appear in when run altogether (`v8all`). The tests have also been run separately, where the name in parenthesis will be used to identify them later.

1. Richards (`v8ric`), an implementation of the Richards benchmark, which simulates a task dispatcher of an operating system.
2. DeltaBlue (`v8del`), an incremental constraint hierarchy solver.
3. Crypto (`v8cry`), implements a subset of the RSA encryption routines.
4. Raytrace (`v8ray`), a simple ray tracer.
5. Earley Boyer (`v8ear`), a classic scheme type theorem prover.
6. RegExp[1], tests the speed of regular expressions.
7. Splay (`v8spl`), builds and modifies self-adjusting binary search trees.

The tests have been run separately, because they are very different in terms of code, thus they might exploit the CPU differently. Most tests run so fast on physical hardware, that good timings could not be measured. To prevent inaccurate timings the tests `v8ric`, `v8del`, `v8cry`, and `v8ray` have been extended to run 10 times both in the simulator and on physical hardware.

5.3.2 SunSpider

The SunSpider benchmark is made by the WebKit team. Unlike the V8 Benchmark Suite, which is a collection of JavaScript programs, SunSpider utilizes a shell script which loads the JavaScript engine wrapper and feeds it with the JavaScript programs. This makes it more complicated to use.

When running the benchmark using the Timing CPU type, the output contained false values. The running time was obviously affected, as it ran much faster, thus the result for the Timing CPU has not been included.

[1] This test failed to run in the simulator, and has not been included in any test.

5.4 Results

The results gathered can be seen in appendix A. To find the CPU configuration, which best matches the physical CPU, each simulated CPU has been plotted against the physical CPU in the four plots in figure 5.2.

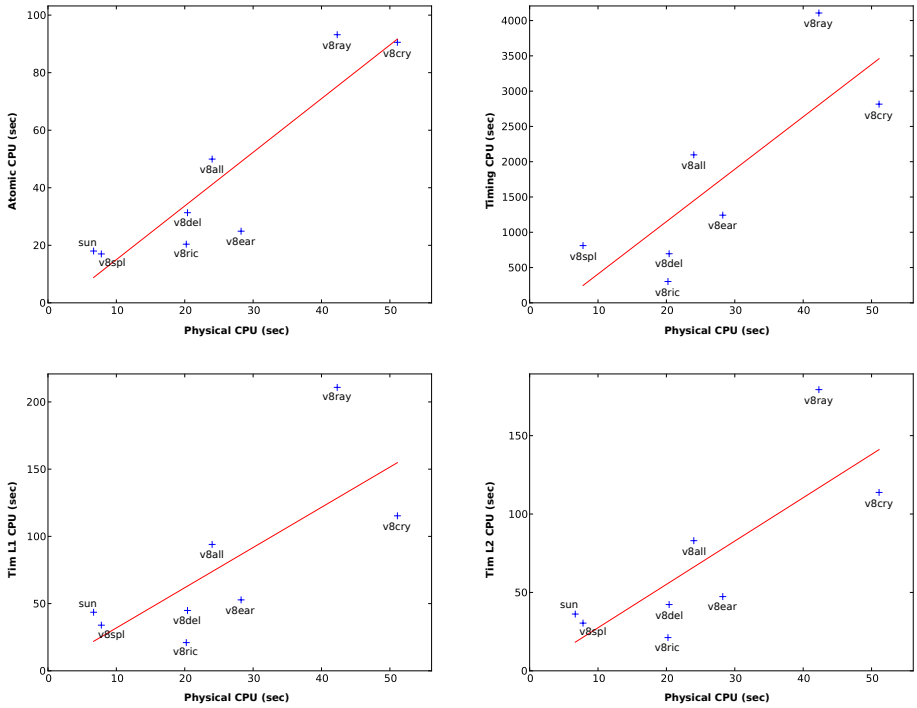

Figure 5.2: Comparison of Simulated CPUs and the physical CPU.

The results are rather similar between the different simulated CPUs, although the scaling differentiates significantly, especially the timing CPU shows much different timings. The scaling is due to the timings on the different components.

Using the normalized mean square error, as shown in table 5.3, our tests show that the Atomic CPU type has the best correlation with the physical CPU. This is the opposite of what we would have expected, as the memory hierarchy of the Atomic CPU should differ most from the one of the physical CPU.

The conclusion is, although the simulated CPUs model cache hierarchies and memory timings, they are not very similar to the physical CPU, because the

	Atomic	Timing	Tim L1	Tim L2
r^2 (norm)	0.023	0.053	0.033	0.032

Table 5.3: Normalized mean square error results.

most simple simulated CPU was the most similar.

Another interesting measure is the time it took to complete the simulations on the host computer. The times are shown in appendix A.4. The times are compared in figure 5.3, where the timings are shown for each CPU type grouped on each benchmark.

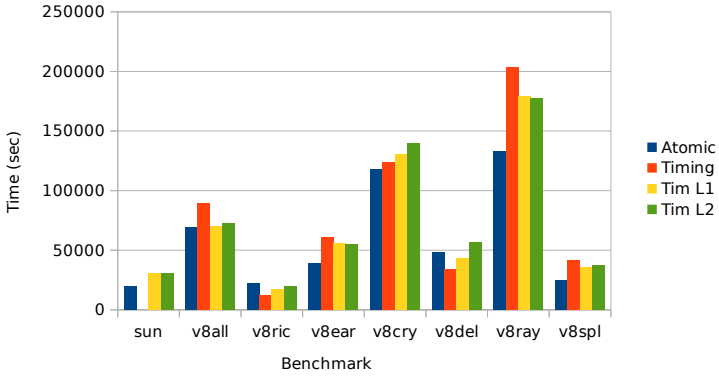

Figure 5.3: Execution times on host.

The Atomic CPU was expected to be faster, since it does less bookkeeping of timings than the others. The results show that it is not significantly faster than the others, and in some cases even slower.

Simulation was known to be much slower than normal execution, the measured performance of the framework showed that simulation is approximately 2400 to 3100 times slower than normal execution.

Chapter 6

Future Work

This section is dedicated to suggestions for further work as well as ideas for optimizing the benchmarking framework. It is important to keep in mind that this work only provides a glimpse of what this framework is capable of. As the performance comparison in chapter 5 shows, the simulator has still work needed in order to accurately simulate a physical CPU, thus it should be kept in mind before looking into the suggested changes. Eventually the on-going development of gem5 will increase its maturity, going a step further towards complete x86 64-bit simulation.

6.1 Improving the Simulator

If current development of gem5 continues, it will reach a state where it yields more reliable results than shown in chapter 5, and when it does, the following suggestions would improve the functionality of this framework.

6.1.1 The Ruby Memory Model

There is ongoing work to make the Ruby memory model work for the x86 64-bit architecture, which will greatly increase the modeling possibilities and accuracy of the simulation. As the result in section 5.4 show, the simulator lacks memory behavior to accurately resemble current physical CPUs.

6.1.2 Extend Libelf Functionality

The gem5 simulator already depends on Libelf[1], for reading the symbol table of the Linux kernel. This aids the debugging of the kernel, because the simulator can reference the code rather than addresses. An improvement to this could be to extend the functionality to load multiple symbol tables, such that any application running on top of the simulated Linux system could have its symbol table loaded. This would enable us to load debug symbols of the entire software stack into the simulator, given that the source code is available. The lookup would require more memory, but implemented efficiently, e.g. using hash maps, the overhead should be almost negligible. In this way the traces saved would require a lot less post processing.

6.1.3 Share State with Virtual Machine Monitors

While images can be shared among the simulator and virtual machine monitors as mentioned in section 2.3.3, it would be even better if they could share the state as well. This would enable us to run only the region of interest in the simulator and all other code in a virtual machine monitor. Since we only need to record statistics and trace within the region of interest, there should be no reason why the state could not be shared. The advantage of using virtual machine monitors like KVM or VirtualBox is, that they use hardware acceleration extensively, leading to a significant speedup.

Besides sharing the state such as memory, registers etc. the biggest issue of sharing a running system will be that the hardware emulated should be exactly the same for both the simulator and the virtual machine monitor.

[1] Library for creating, reading and modifying ELF binaries.

6.2 Extending the Capabilities of the Framework

This framework is not limited to benchmarking of JavaScript programs, as the method of tracing using simulation of the x86 64-bit instruction set could be applied to any x86 64-bit binary. In fact, this experiment shows that any x86 64-bit binary can be traced, because we showed that every layer of the software stack was traceable. The framework could be used to trace every layer of the software stack, from high level code to low level operating system code. This project showed the flexibility of the tracing output, which could easily be extended for more specific needs.

Simulation tests are particularly relevant compared to other measuring techniques when running very small tests. In these cases sampling has its difficulties. To get more accurate results, tests are usually run multiple times in order to warm up the caches.

The tool made to summarize information from the trace files is rather simple, using existing tools to visualizing traces of programs would be an improvement. Further experiments could be done to show how these tools could be used.

A big advantage of using a simulated system, is that the results will be comparable across computers as it is not hardware dependent. Thus more browsers and JavaScript engines, such as Chromium with V8 and Firefox with SpiderMonkey, could use the framework to compare traces and measured execution times. It would also be possible to document improvements over time as long as the same configuration of the simulator was used.

The gem5 simulator support running other platforms than the x86 64-bit platform. The platforms are mentioned in section 2.2. It would be interesting to compare the performance of the same software such as WebKit running on different platform. A platform like ARM is gaining popularity through the smartphone market, even Microsoft is working on porting their next operating system, Windows 8, to run on ARM [25]. Comparing ARM and x86 could give a better understanding of the limitation and benefits of the different architectures.

Chapter 7

Conclusion

With the rising popularity of large web applications, fast execution of JavaScript is crucial. In order to optimize the execution of JavaScript programs, detailed information on the execution is necessary. Where effort has been put into high level profiling, no tools currently collect low level information. As the user experience depends on the execution time of the entire system, the entire software stack should be examined.

The aim of this thesis was to develop a benchmarking framework to accurately measure execution timings of JavaScript programs. The framework developed enables JavaScript engine developers to collect a fine-grained trace of the execution. The framework makes it possible to view detailed information about time consumption, wherever it is in the program itself, a library or the kernel. The framework is capable of saving full traces of the execution with timings on the individual instructions. Using the saved memory map, it was also possible to map the instructions back to the source code.

The framework consists of complex components such as an instruction set simulator, a Linux system and a browser. In order to make these components play together, a deep understanding of each component was necessary. The current development state of gem5 caused some additional challenges. Despite the challenges we were able to show the tracing capabilities of the framework. We were also able to run several JavaScript benchmarks, to in-

spect the performance of four different simulated CPUs. Despite the various memory hierarchies configured in the simulated CPUs, they did no better in resembling the physical CPU.

During this project we encountered the challenges associated with using and extending an advanced simulator such as gem5. Despite that, we managed to contribute to the gem5 simulator, both with patches to their repository and through discussions in their mailing list.

The use of low level details is relevant for analyzing all layers of the software stack, which is a good extension to our current profiling tools. The capabilities of comparing entire systems will also be essential as popularity of platforms like ARM increases, especially if the different platforms could be compared.

Appendix A

Measured Data

Benchmark	kraken	sun	v8all	v8ric	v8ear
	252.73	6.92	24.11	20.25	97.65
	251.01	6.63	23.94	20.21	106.23
	251.40	6.67	24.01	20.22	11.01
	250.30	6.55	24.29	20.25	10.75
	251.74	6.63	24.10	20.20	8.90
	252.47	6.64	24.15	20.22	10.63
	252.28	6.62	24.30	20.23	8.87
	254.69	6.67	23.99	20.23	8.95
	254.40	6.64	24.11	20.23	8.82
	251.68	6.74	23.07	20.22	10.60
Mean	252.27	6.67	24.01	20.23	28.24
Std. dev.	1.32	0.09	0.33	0.01	36.91

Table A.1: Execution times on physical hardware #1.

Benchmark	v8cry	v8del	v8ray	v8spl
	51.16	20.39	42.67	7.10
	51.25	20.41	42.37	7.98
	51.14	20.37	42.72	7.98
	51.04	20.46	41.84	7.97
	51.03	20.38	41.67	7.94
	51.21	20.44	41.81	7.99
	51.16	20.42	42.60	7.97
	51.11	20.36	42.28	8.06
	50.98	20.35	42.40	7.89
	50.89	20.43	42.62	7.18
Mean	51.10	20.40	42.30	7.81
Std. dev.	0.11	0.03	0.37	0.34

Table A.2: Execution times on physical hardware #2.

	Atomic	Timing	Tim L1	Tim L2
sun	18.01	-	43.57	36.21
v8all	49.96	2096.81	94.01	83.01
v8ric	20.39	301.97	20.99	21.2
v8ear	24.90	1242.42	52.78	47.34
v8cry	90.55	2815.62	115.31	113.78
v8del	31.33	695.72	44.89	42.26
v8ray	93.19	4107.13	210.83	179.36
v8spl	17.00	811.32	33.96	30.43

Table A.3: Execution times for simulations.

	Atomic	Timing	Tim L1	Tim L2
sun	19749.11	-	30237.99	30246.57
v8all	69237.23	89390.60	70341.44	72491.95
v8ric	22071.54	12432.90	17090.29	20090.63
v8ear	39381.61	61027.51	56095.86	55206.31
v8cry	117844.92	123562.40	129975.20	139477.91
v8del	47891.86	33921.00	43299.96	56495.83
v8ray	133296.29	203146.17	178926.94	177432.23
v8spl	24833.86	41097.21	35553.73	37498.03

Table A.4: Simulation times on host.

Appendix B

Source Code

Listing B.1: Part of pc_event.hh

```
class PCEvent
{
  protected:
    std::string description;
    PCEventQueue *queue;
    Addr evpc;

  public:
    PCEvent(PCEventQueue *q, const std::string &desc, Addr pc);

    virtual ~PCEvent() { if (queue) remove(); }

    // for DPRINTF
    virtual const std::string name() const { return description; }

    std::string descr() const { return description; }
    Addr pc() const { return evpc; }

    bool remove();
    virtual void process(ThreadContext *tc) = 0;
};

class BreakPCEvent : public PCEvent
{
  protected:
    bool remove;
```

```
  public:
    BreakPCEvent(PCEventQueue *q, const std::string &desc,
                 Addr addr, bool del = false);
    virtual void process(ThreadContext *tc);
};

class TracePCEvent : public PCEvent
{
  protected:
    bool enable;
    bool fire;
    std::string bin;
    Addr libaddr;
    std::string lib;
    std::string memfile;
  public:
    TracePCEvent(PCEventQueue *q, Addr addr, const std::string &bin,
              bool enable, Addr libaddr, const std::string &lib,
              const std::string &memfile);
    virtual void process(ThreadContext *tc);
  private:
    void saveMemMap(ThreadContext *tc, const std::string filename);
    Addr findLibAddr(ThreadContext *tc, std::string libname);
    std::string memFilename(Addr node, VirtualPort *vp);
    std::string memFilePath(Addr node, VirtualPort *vp);
};
```

Listing B.2: fib.html

```
<html>
<head>
<script type="text/javascript">
function fib(num) {
    var a = 0;
    var b = 1;
    var sum;
    for (var i=2; i<num; i++) {
        sum = a + b;
        a = b;
        b = sum;
    }
    return sum;
}
function run(num) {
    var t = fib(num);
    window.close();
}
</script>
</head>
<body onload="run(10)">
</body>
</html>
```

Listing B.3: fib.js

```
function fib(num) {
    var a = 0;
    var b = 1;
    var sum;
    for (var i=2; i<num; i++) {
        sum = a + b;
        a = b;
        b = sum;
    }
    return sum;
}
fib(10);
```

Bibliography

[1] AMD. AMD64 Technology AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2010.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15:357–390, November 1997.

[3] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43:52–61, January 2009.

[4] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, July 2006.

[5] Gabe Black. gem5: changeset 8626:19eed0015983, x86: Fix a bad segmentation check for the stack segment. http://repo.gem5.org/gem5/rev/19eed0015983. Submitted December 1st, 2011.

[6] Gabe Black. gem5: changeset 8807:1a84c6a81299, SE/FS: Make SE vs. FS mode a runtime parameter. http://repo.gem5.org/gem5/rev/1a84c6a81299. Submitted January 28th, 2012.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, November 2005.

[8] Gentoo Foundation. Gentoo Linux AMD64 Handbook. Accessed February 13th, 2012.

[9] gem5 team. Disk images. http://gem5.org/Disk_images. Accessed October 14th, 2011.

[10] gem5 team. Linux kernel. http://gem5.org/Linux_kernel. Accessed January 16th, 2011.

[11] gem5 team. Status matrix. http://gem5.org/Status_Matrix. Accessed October 5th, 2011.

[12] gem5 team. X86 instruction decoding. http://gem5.org/X86_Instruction_decoding. Accessed December 20th, 2011.

[13] Google Inc. V8 JavaScript Engine. http://code.google.com/p/v8/. Accessed January 16th, 2012.

[14] Anders Handler. gem5: changeset 8670:aae12ce9f34c, cpu: Remove alpha-specific pc alignment check. http://repo.gem5.org/gem5/rev/aae12ce9f34c. Submitted January 9th, 2012.

[15] Ariya Hidayat. Javascriptcore. http://trac.webkit.org/wiki/JavaScriptCore. Accessed February 12th, 2012.

[16] Intel Corp. Intel Core i7-600, i5-500, i5-400 and i3-300 Mobile Processor Series, Volume One, January 2010.

[17] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 –58, feb 2002.

[18] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:92–99, November 2005.

[19] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. AMD, draft version 0.99.5 edition, September 2010.

[20] Shanku Niyogi, Amanda Silver, John Montgomery, Luke Hoban, and Steve Lucco. Evolving ecmascript. http://blogs.msdn.com/b/ie/archive/2011/11/22/evolving-ecmascript.aspx. Accessed January 4th, 2012.

[21] Jolie O'Dell. Watch out, ruby on rails: Node.js is the most popular repo on github. http://venturebeat.com/2011/11/28/node-is-one-of-the-cool-kids-now/. Accessed February 15th, 2012.

[22] Adam Overa. Web Browser Grand Prix VI: Firefox 6, Chrome 13, Mac OS X Lion. http://www.tomshardware.com/reviews/web-browser-performance-standard-html5,3013.html. Accessed September 17th, 2011.

[23] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. *USENIX Conference on Web Application Development (WebApps)*, June 2010.

[24] Adam Roben, Andreas Kling, Daniel Bates, Maciej Stachowiak, Mark Rowe, Sam Weinig, and Simon Fraser. Webkit2 - high level document. http://trac.webkit.org/wiki/WebKit2. Accessed January 5th, 2012.

[25] Steven Sinofsky. Building Windows for the ARM processor architecture. http://blogs.msdn.com/b/b8/archive/2012/02/09/building-windows-for-the-arm-processor-architecture.aspx. Accessed January 5th, 2012.

[26] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 9th edition, January 2002.

[27] SWIG. Swig-1.3 documentation. http://www.swig.org/Doc1.3/SWIGDocumentation.html. Accessed January 19th, 2012.

[28] Nilay Vaish, Nathan Binkert, and Ali Saidi. Disk image format. http://www.mail-archive.com/gem5-users@gem5.org/msg00490.html. Accessed November 19th, 2011.

[29] W3C. Html5, w3c working draft 25 may 2011. http://www.w3.org/TR/html5/. Accessed March 4th, 2012.

[30] David P. Wiggins. Xvfb. http://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml. Accessed January 2nd, 2012.

[31] Matt T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 23–34. IEEE, April 2007.