FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Error detection based on execution-time monitoring

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Dieter Steiner
Matrikelnummer 0326004

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Wien, 15. September 2016

Dieter Steiner

Peter Puschner

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Error detection based on execution-time monitoring

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Dieter Steiner
Registration Number 0326004

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner

Vienna, 15th September, 2016

Dieter Steiner	Peter Puschner

Erklärung zur Verfassung der Arbeit

Dieter Steiner
Schopenhauerstraße 59/6, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. September 2016

Dieter Steiner

Danksagung

An dieser Stelle gebührt ein großes *Dankeschön!* allen, die mich beim Verfassen dieser Diplomarbeit und während meines vorausgehenden Studiums unterstützt und begleitet haben.

Allen voran gebührt mein Dank meinen Eltern – Adelina und Franz – welche mir das Studium ermöglichten und mir immer mit viel Liebe und Geduld zur Seite standen.

Ein herzliches *Danke!* auch meiner Freundin Melanie, die für mich immer viel Geduld, aufmunternde Worte und die ein oder andere Tasse Kaffee übrig hatte. Vielen Dank, dass du mich so sehr unterstützt hast, auch wenn ich gestresst oder müde war.

Ein großes *Danke!* auch meinen Studienkollegen, für die Zeit die wir in Vorlesungen, Lerngruppen oder bei angeregten Diskussionen verbracht haben, allen voran Christopher, der schon vor dem Studium ein guter Freund war und es noch immer ist.

Last but not Least ein aufrichtiges *Danke!* an meinen geschätzten Betreuer Peter Puschner für seine Geduld und Unterstützung im Zuge dieser Diplomarbeit.

Dieter Steiner
Wien, September 2016

Kurzfassung

Echtzeitsysteme finden heutzutage Anwendungen in allen erdenklichen Lebensbereichen – von der Steuerung einer Waschmaschine, bis hin zur autonomen Navigation von Marsrobotern. Bei diesen Systemen ist die Korrektheit des Systemverhaltens nicht nur von den logischen Ergebnissen der durchgeführten Berechnungen, sondern auch von dem physikalischem Moment, zu dem die errechneten Resultate vorliegen, abhängig. Die Bestimmung der maximalen Ausführungszeit einer Berechnung ist daher unerlässlich um zu bestimmen, ob diese Systeme die an sie gestellten zeitlichen Vorgaben einhalten können.

Doch enthält die Ausführungszeit einer Berechnung noch weitere verwertbare Informationen?

Im Rahmen dieser Diplomarbeit wird untersucht ob – und falls ja, wie – sich das Zeitverhalten eines Programms durch Hardwarefehler verändert, und ob es möglich ist die Beobachtung der Laufzeit eines Programms zur Erkennung von Fehlern in der Abarbeitung zu verwenden um frühzeitig ungewolltes Systemverhalten zu erkennen und zu verhindern.

Zu diesem Zweck werden repräsentative Microbenchmarks identifiziert und implementiert, ein Fault-Injection-Modell erstellt, und darauf aufbauend bewusst Fehler in den Programmcode eingeschleust. Die so modifizierten Benchmarks werden anschließend vermessen und die Ergebnisse dieser Messungen mittels geeigneter Verfahren ausgewertet. Die daraus gewonnenen Resultate und Schlussfolgerungen werden aufbereitet und sowohl graphisch als auch textuell präsentiert und interpretiert.

Im Zuge dieser Arbeit wird die Robustheit der vorgeschlagenen Methode gezeigt, es werden — algorithmenabhängig — bis zu 70% vorher undetektierter Fehler erkannt. Die Methode kann mit vertretbarem Aufwand in vorhandene Systeme integriert werden und bietet eine weitere Absicherung gegen mögliche Fehler.

Abstract

Nowadays real-time systems can be found in every aspect of modern life – from controlling a washing machine to the autonomous navigation of Mars rovers. In this systems, the correctness of a systems behavior does not only depend on the actual computation results, but also on the physical instant at which these results are produced.

Determining the maximal execution time of a calculation is, therefore, essential to ascertain if the system can meet its temporal requirements.

But is there more usable information within an algorithms execution time?

This thesis investigates if respectively how the temporal behavior of an algorithm changes in the presence of errors that are caused by hardware faults. It examines if the observation of task run times can be used to detect these errors, so that undesired system behavior and failures can be averted early.

To reach this goal, a number of microbenchmarks will be identified and implemented; a fault injection model will be defined, and – building onto it – errors will be injected into the benchmarks' program code. Those modified benchmarks will then be measured and the results evaluated with suitable methods. The gathered evaluations and insights will be presented in graphical and textual ways.

This work will show a robust method, detecting – depending on the algorithm – up to 70% of previously undetected errors. The method can be implemented into existing systems with reasonable effort and overhead, and provides an additional layer of protection against errors.

Contents

CHAPTER 1

Introduction

1.1 Motivation

With continuing miniaturization and the rise of cheap computing, embedded systems became ubiquitous within the last decades. In every aspect of modern life, inconspicuous microcomputer systems are present nowadays. From smartphones to dishwashers, from electric cars to sophisticated home automation systems — we are surrounded by computing systems, and we have started to rely heavily on them.

Logic dictates that error detection should be an important aspect of such applications, still the cost of implementing sophisticated detection methods (especially hardware based) sometimes exceed the already tight budget — in many modern applications cost is of outermost importance.

1.2 Goals of the Thesis

In the course of this thesis we will be following several major goals:

- Show that computational errors have a measurable influence on an algorithms execution characteristics i.e. devise an *experimental* setup showing that the cycle- and instruction counts differ in many cases in the presence of errors.
  Figure 1.1 illustrates the basic concept. The upper half (green) of the figure shows a histogram of the processing time used by an algorithm with varying input data. Those measurements (or a suitable theoretical model or static analysis) can then be used to establish bounds for the possible (valid) execution times. (denoted by the vertical green lines)

The lower half (red) shows an (expected) distinct change in behavior in the presence of a fault (e.g. a bit flip in the instruction stream). The instances exhibiting a temporal behavior outside the previously established bounds are considered abnormal.

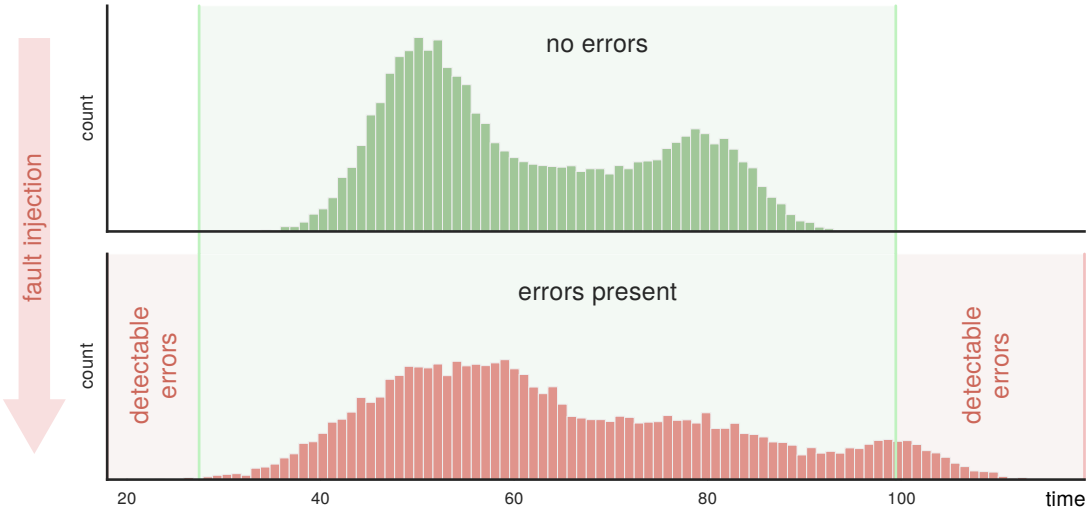

Figure 1.1: Change in execution time distribution in the presence of errors

- Show that those abnormal timing behaviors can be used as a reasonably reliable – and computationally cheap – additional tool for detecting those errors by monitoring an algorithms execution characteristics.
- Show this for diverse types of algorithms – giving a broad overview of the possibilities.
- Compare Worst Case Execution Time (WCET)-optimized algorithms against Average Case Execution Time (ACET) optimized ones to see if there is a benefit regarding the proposed method.
- Implement and execute the proposed method as a proof-of-concept on a simulator and on real hardware; compare and evaluate the results.

1.3 Structure

The structure of the thesis is as follows: Chapter 2 gives a brief introduction to the basic concepts of real-time-systems, fault injection and execution time analysis — the theoretical foundation of this work. The goal of this chapter is to establish a common notion and provide literature references to further reading. Chapter 3 defines the scope of this thesis. It details the hypothesis, reasons about temporal behavior, and lists the assumptions made. Chapter 4 is about the concept and design of the measurement environment, the needed libraries and

tools, and the chosen fault injection model. Subsequently, Chapter 5 details the infrastructure used, consisting of hard- and software tools, the adaptions and configuration settings used, and the reasons for using them. The benchmarks used are listed in Chapter 6. It explains their behavior and provides background information where necessary. It also includes an instruction level analysis if the generated code. Chapter 7 presents the evaluation results, compares the real hardware to the simulation results, explains the detection model used, shows the results of the detection algorithm, and ends with an interpretation of the results. This thesis is concluded by Chapter 8, providing a summary of the work done, and an outlook to future research topics and potential applications.

The appendix follows with detailed results (Appendix A) and source code (Appendix B) for the interested reader.

CHAPTER 2

Background and Terminology

Introduction

The scope of this chapter is to provide a brief overview of the theoretical concepts, the state of the art, and the terminology used in the following chapters.

2.1 Real-Time Systems

As detailed in [Kop11], a Real-Time Computer System (RTCS) is a computer system, in which the correctness of the systems behavior depends not only on the *results of computations*, but also on the *time* at which these results are produced. A RTCS is always a component of a larger (physical) Real-Time System (RTS) (See figure 2.1). It has to react to outside stimuli, from the controlled object or the operator, within a certain time frame dictated by its environment, the so-called *deadline*.

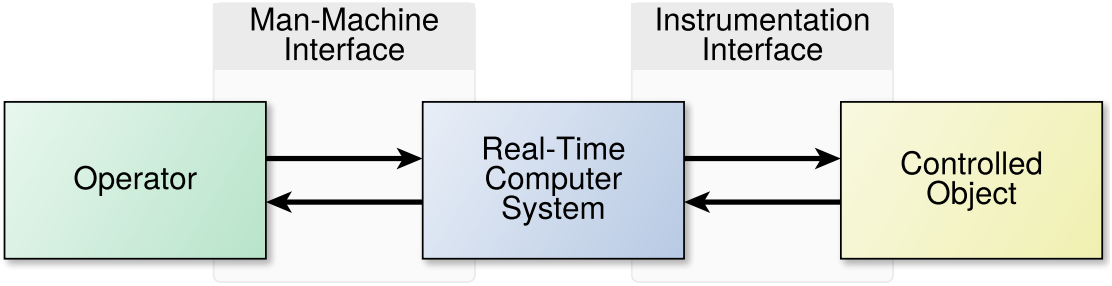

Figure 2.1: Real-Time System [Kop11]

Commonly RTSs get also referred to as cyber-physical systems (CPSs). [LS15] defines the term CPS as "[...] an integration of computation with physical processes whose behavior is defined by *both* cyber and physical parts of the system". Lee and Seshia also note that "[...] [a] CPS is about the *intersection*, not the union, of the physical and the cyber. It is not sufficient to separately understand the physical components and the computational components. We must instead understand their interaction". This is essentially the same definition as given by [Kop11], but with a clear emphasis on the *physical* aspect of RTSs.

Depending on the usefulness of a computed result within the RTS and the consequences of not producing it within the set time frame, deadlines can be classified into three categories:

- *soft* — the results usefulness degrades as time passes, quality of service is reduced (e.g. Voice over IP).
- *firm* — A results usefulness after missing a deadline is zero, infrequent misses are tolerable, but degrade the systems quality of service.
- *hard* — like *firm*, but a catastrophic event could occur if a deadline is missed — resulting in total system failure (e.g. air traffic control)

If an RTS needs to meet at least one hard deadline it is called a *hard real-time system*. If no such deadline exists it is called *soft real-time system*. [Kop11]

2.2 Faults/Errors/Failures

The terms *fault*, *error*, and *failure* are of uttermost importance when talking about RTSs and error detection. As there are different definitions through the literature, this thesis uses the one given in [Kop11] (originally taken from [ALR04]; c.p. figure 2.2):

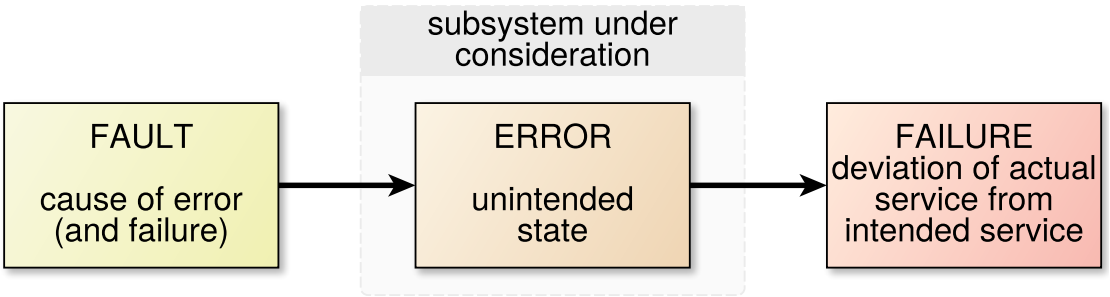

Figure 2.2: Faults, errors and failures [Kop11]

- A *failure* "occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function."

- An *error* is “the part of a systems total state that may lead to a failure – a failure occurs when the error causes the delivered service to deviate from correct service. ”
- A *fault* is “the adjudged or hypothesized cause of an error”. A fault can be either *active* – i.e. it produces an error – or *dormant* otherwise.

2.3 Fault Injection

The concept of using Fault Injection (FI) for finding errors dates back to the early 1970s (compare [CCS99]). Starting with the introduction of intentionally bad solder joints and wrong cabling, the general approach was soon recognized as promising, and a multitude of new methods and formal fault classifications were devised. (e.g. [CBC92]) Well established methods in the field of software-based Fault Injection are shown in [IKH14] and [HTI97].

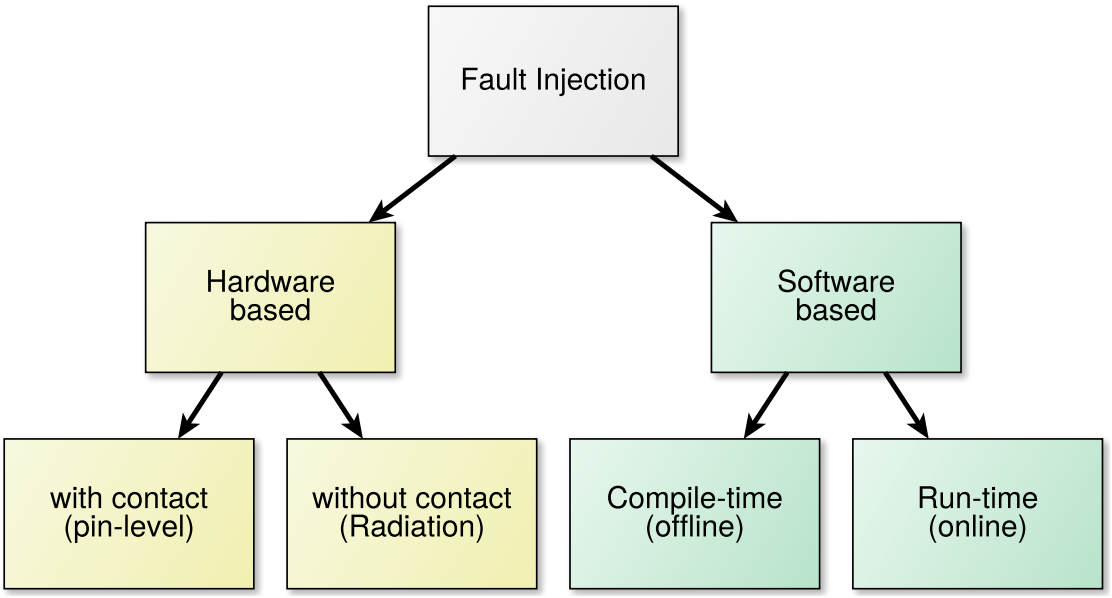

Figure 2.3: Fault Injection methods

As shown in Figure 2.3[HTI97] FI methods are commonly classified as hardware- and software-based. Software-based methods can again be divided into *Compile-time*- and *Run-time injection* methods. The latter manipulates a systems state in a dynamic way (e.g. by hooking system calls (syscalls)), whereas the former directly manipulates the programs code. This can happen by modifications to the source code (e.g. in mutation testing [JH11]), wrapping of function calls (and modifying the input/output values, [IKH14]), or by manipulation of the input data (e.g. corrupt data files)

2.4 WCET Analysis

The goal of static WCET analysis is the determination of a safe, but tight upper bound for the runtime of a certain program or parts of it. The WCETs can be used as guarantees for the timeliness of the analyzed task and are thereby critical for design and validation of hard real-time systems [WEE08].

Problems arise from the fact that the execution time of a given task can vary depending on its inputs and the state of the execution hardware.

In contrast to dynamic, execution based methods, which just measure execution time, static analysis can give a safe result whereas dynamic methods suffer from the state explosion problem (i.e. all possible inputs × all possible machine states). This means that with a dynamic approach the real WCET usually ca not be provoked. Specially crafted microbenchmarks can be an exception here, as they can be designed to be simple enough to reach full coverage. Figure 2.4 explains the problem. The graph shows the execution times of 40000 simulated test runs with different inputs and hardware states. Neither the theoretical Best Case Execution Time (BCET) nor the WCET are reached. "Upper bound" denotes a possible WCET estimate as a result of static analysis, "lower bound" for the BCET. ACET denotes the mean of all possible execution times, for which most standard algorithms are optimized.

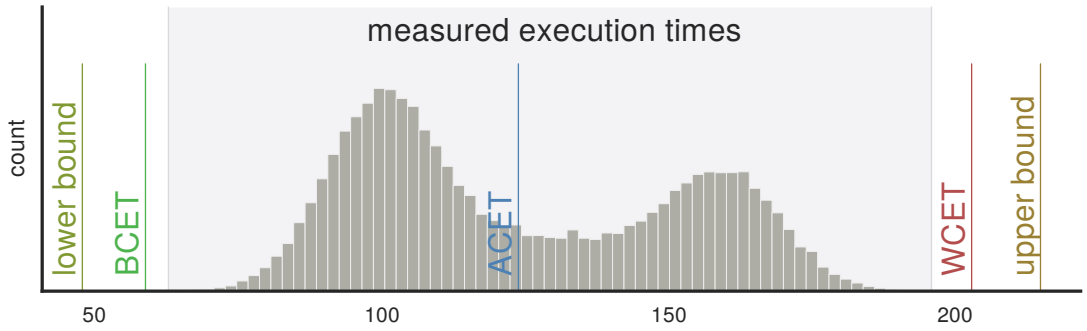

Figure 2.4: Measured execution times and BCET/WCET

One challenge that makes a static analysis difficult is dealing with the speculative, performance enhancing, features of modern machine architectures, i.e. caches, deep pipelines, branch prediction, out-of-order execution, etc. These features make it much harder to give a tight estimation of the WCET. Particularly caching can make the WCET estimation hard, as a *Hit* (needed data is in the cache) usually means the data can be accessed immediately, whereas a *Miss* (the data has to be fetched from a higher tier cache or the main memory) can result in very long delays (up to several hundred cycles on modern desktop computers) [Dre07]. A simple analysis tool could assume that every memory access is a cache miss,

which, in turn, would result in a safe WCET estimation, but the result would not be very tight, whereas an overly optimistic tool could assume that every access is a hit, which obviously violates the safety criterion.

The BCET can be easier to estimate (whenever a trivial input exists [1]), but, in general, getting a tight lower bound for it is as hard as getting a tight WCET bound.

Methods for optimizing program code towards a constant execution time are detailed in [Pus03]. (And continued in [Pus05] and [PKH12])

[1] e.g. an empty list as input for a sorting algorithm

CHAPTER 3

Scope

Introduction

This chapter outlines the scope of the work done, the underlying hypothesis, the expected temporal behavior, and other assumptions made. It will also peek into future work and possible extensions of the method.

3.1 Hypothesis

As already mentioned briefly in section 1.2, the working hypothesis used in this thesis is that computational errors have a *measurable influence* on an algorithms execution characteristics.

In the course of this thesis the evaluated characteristics are:

- *Cycle counts* – This counter is proportional to the time passed – assuming a constant Central Processing Unit (CPU) clock. It also accounts for variable-cycle-instructions, L1- and L2-cache delays, and load/store delays when accessing the main memory.
- *Instruction counts* – This value is only affected by the actual number of instructions executed. It disregards hardware dependent timing effects and only accounts for actual algorithmic differences – i.e. it will change only when an error leads to a different execution trace.

For the evaluation, faults will be injected by modifying the instruction stream of the algorithm (c.p. sections 2.3 and 4.2) and monitoring the mentioned performance counters – resulting in a measurable change in the execution-time behavior of the algorithm.

3.2 Temporal Behavior

Different algorithms can exhibit vastly different timing behaviors, depending on the input data used. A prime example for that is the *quicksort* algorithm[Hoa61], being – on average – one of the fastest generic sorting algorithms, with an average runtime behavior of $O(n \log n)$, but degrading to $O(n^2)$ for its worst case input sequence. This can make the establishment of reasonable timing bounds for it quite challenging.

Other algorithms might show simpler timing behavior, making it easier to spot deviations from the expected temporal behavior. Some (generalized) exemplary cases are:

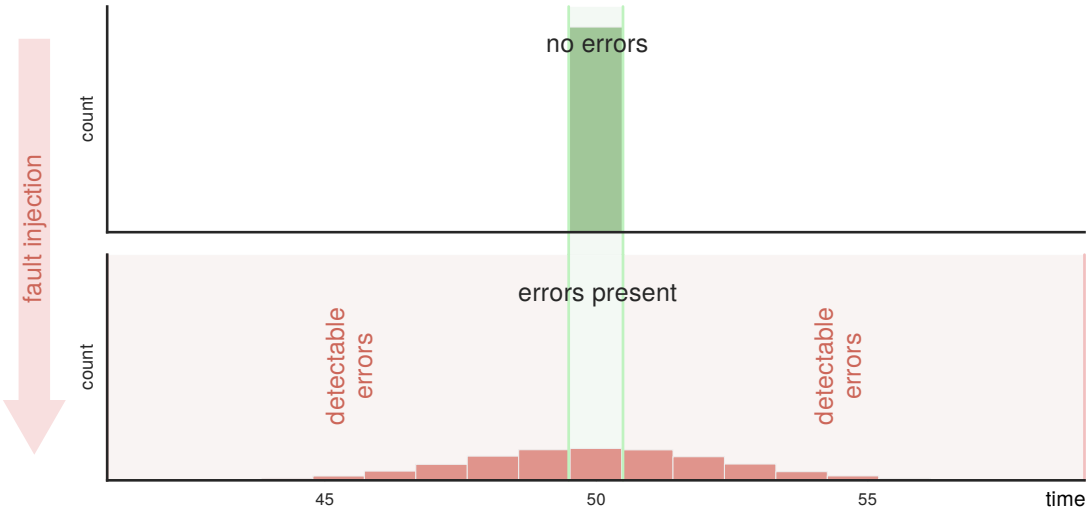

Figure 3.1: execution time behavior – constant (WCET optimized)

- Figure 3.1 shows a WCET optimized algorithm with constant run time in the absence/presence of errors in its instruction stream. The top half (green) of the figure shows a histogram of the processing time used by an algorithm with varying input data. Those measurements (or a suitable theoretical model) can then be used to establish bounds for the possible (valid) execution times – the green lines.
  The bottom half (red) shows a (expected) distinct change in behavior in the presence of faults (e.g. a bit flip in the instruction stream). The instances exhibiting a temporal behavior outside the previously established bounds are considered abnormal.
  In this (optimal) case the detection of errors is trivial, as *any* deviation from the expected constant execution time can be considered an error.

- Figure 3.2 gives a more realistic view of what happens on actual hardware. The algorithm still has approximately constant execution time behavior, but there is some jitter.

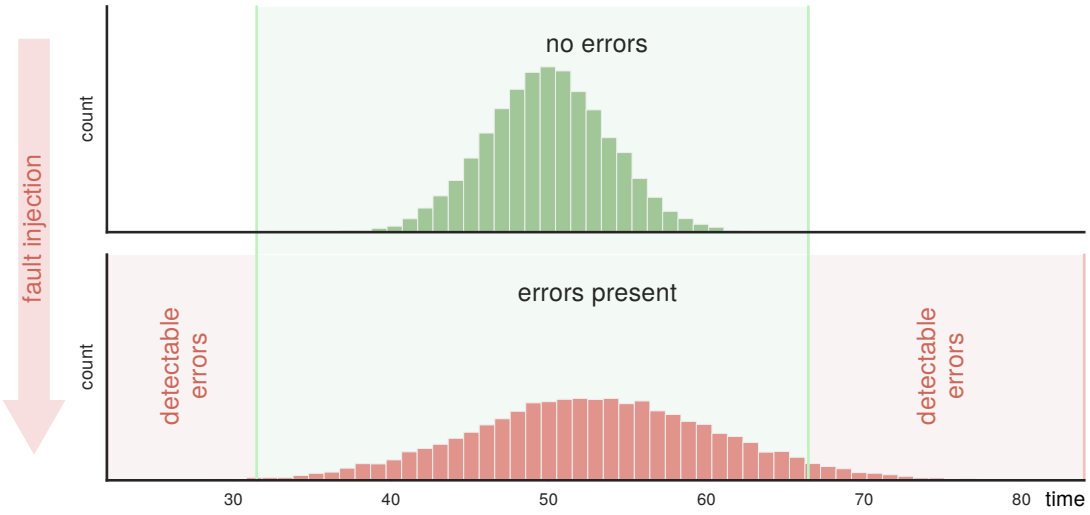

Figure 3.2: execution time behavior – jitter

This can have various reasons, one being caching effects (on non-WCET optimized architectures) – Depending on the input data, an algorithm might access its working set in a different order, affecting what is in the caches, and thereby – as loads from main memory are slower – directly affecting time. Another reason could be interrupts and/or other processes – meaning that other processes are responsible for polluting the caches. Yet another possibility for this behavior arises from the fact that many CPUs (including the ARM Cortex-A7 used in this thesis) include variable-cycle-instructions – i.e. an instruction takes a different amount of cycles to complete, depending on input, pipeline state, etc. On the ARM Cortex-A7 especially the `SDIV` and `UDIV` integer division instructions exhibit this behavior, with a difference of 16 cycles between best and worst case latency. [1]
This case obviously makes the detection of errors harder, as a whole range of timing values cannot be considered as erroneous anymore.

- Figure 3.3 shows the expected most common behavior of an average algorithm. The algorithms runtime varies a lot, depending on its inputs, some cases being more likely than others. As in the previous case upper and lower timing bounds have to be established, and the errors not resulting in obvious temporal behavior can not be detected by the proposed method.

[1] ARM does not seem to publish detailed timings for the Cortex-A7, tests performed by `http://hardwarebug.org/2014/05/15/cortex-a7-instruction-cycle-timings/` (visited on 2016-02-19) seem to indicate that it has instruction latencies comparable to its real-time derivative – the Cortex-R4 [ARM11, pp. C-14].

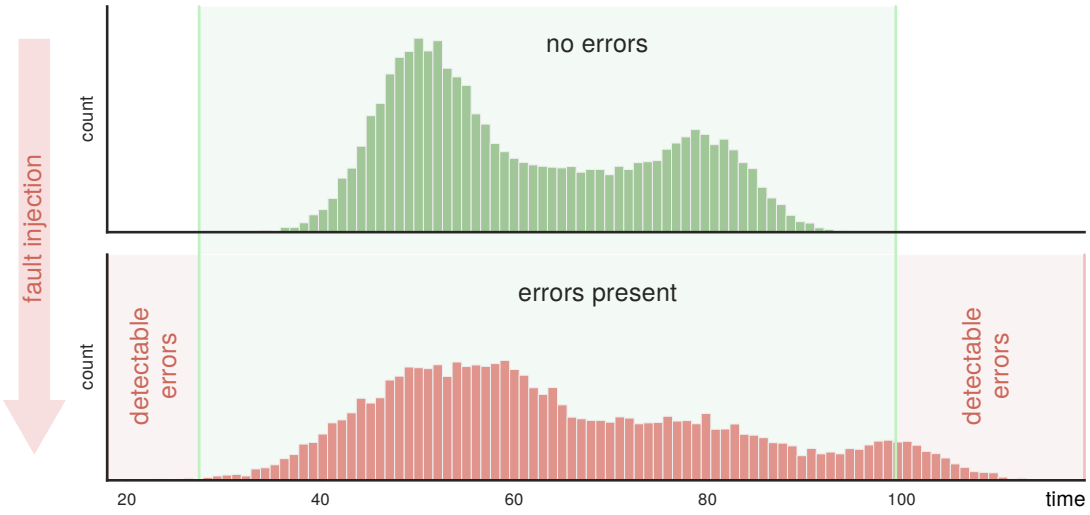

Figure 3.3: execution time behavior – average

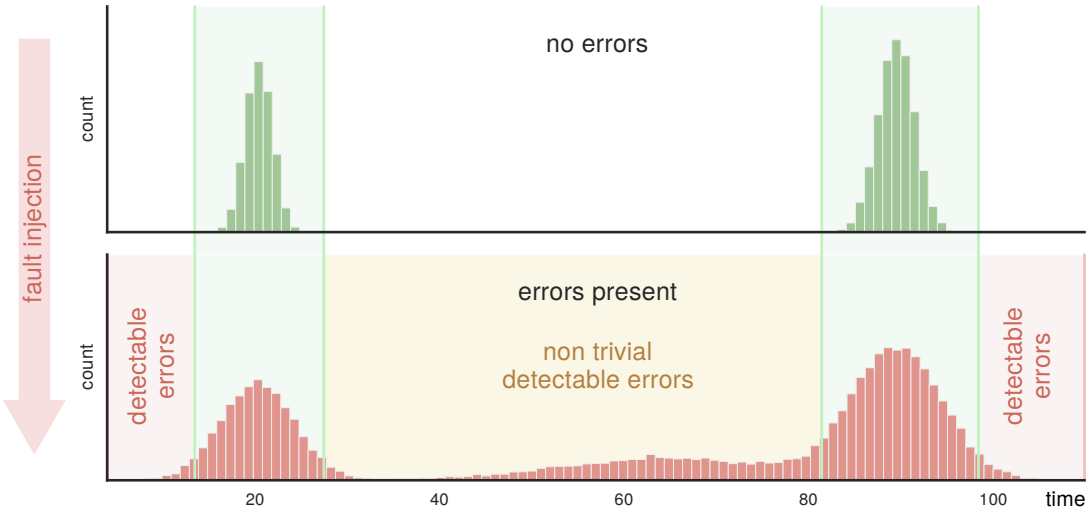

Figure 3.4: execution time behavior – peaks

- Some algorithms might pose a special challenge. In figure 3.4 an algorithm with two distinct *peaks* is shown. This could e.g. be the result of a validity check on the input data – only processing them when they pass – or just some quirk of the algorithm itself. Note that the figure presented is just the most trivial example, there could be an arbitrary number of peaks with arbitrary distances between them.

For instance, an algorithm could perform $1 \dots n$ iterations of a loop with m cycles per iteration, yielding n peaks with a distance of m between them.
The simple bound-based detection method used in this thesis is not able to catch these intervals, yet, as the same principle applies, future improvements can be made by developing a method for catching these *non trivial detectable errors* by establishing a set of intervals for acceptable timing values, rejecting everything outside and in between.

3.3 Assumptions

- Instruction cache is cold – i.e. the first run of the algorithm will be measured.
  This is due to the nature of static fault injection – the likelihood of a crash or some other side effects make a second run unreliable.
- Data caches are hot – as the benchmark fixture sets up the input data prior to running the actual algorithm they will still be cached. As this is constant throughout all runs, it can be neglected.
- The CPU will operate at a fixed frequency. Any power-saving features and dynamic clock switching will be disabled. This is necessary to get consistent results, as dynamic clock switching could introduce unexpected delays and performance shifts.
- The timing bounds are generated empirically, only upper and lower bounds are generated, no inner ones. This thesis focuses on the actual proof of concept – improvements, and fine tuning are the scope of further research.

CHAPTER 4

Measurement Environment

4.1 Concept

To perform the needed measurements, the code was split into two parts, a common framework and the benchmark, the first consisting of an actual measurement library and a common benchmark fixture, the latter being the actual benchmark code under test and a specific fixture tying it to the shared benchmarking framework. (See figure 4.1) This was done to increase modularity and reusability, keeping the duplication of code to an absolute minimum.

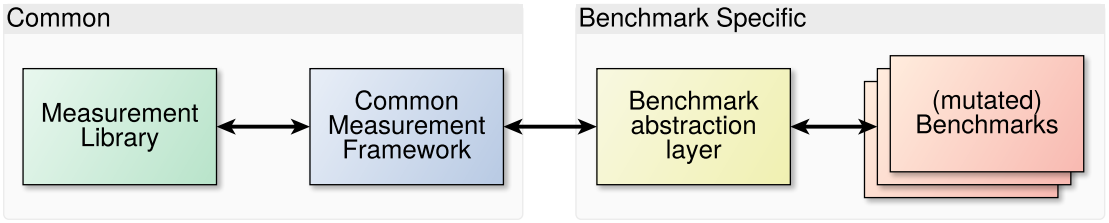

Figure 4.1: Benchmark framework concept

Measurement Library	This library provides an abstraction layer from the `sys_perf_event_open` syscall and the needed ioctls. This ensures that the other parts of the code have no direct dependencies on syscalls or other Linux-specific functions.
Common Measurement Framework	Defines the overall control flow within a benchmark, calls the benchmark specific fixture through a defined software interface, collects and prints the measurement results and provides the command line interface

Benchmark abstraction layer	Wraps a benchmark to provide a common interface. Provides all the methods needed for actually running the benchmark which is not part of the algorithm itself, and thereby will not be mutated. (e.g. memory initializations)
Benchmark(s)	The actual benchmark code under test. Provides usually just a single function as its public interface. This part will be subject to the Fault-Injection experiments.

With this basic structure in place, the benchmarks can be tightly integrated into the measurement framework. This framework can be thought of as a *wrapper* around the whole compilation and evaluation process. It takes care of creating and mutating the benchmarks, running them in gem5 and on the Raspberry Pi 2, collecting the measurement results into a database, and then analyzing the collected data, estimating an execution-time model and applying it to get detection results. (See figure 4.2)

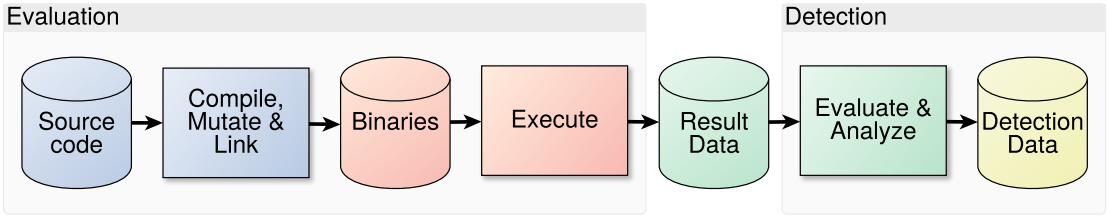

Figure 4.2: Measurement framework concept

Source code	Code base for the benchmarks, abstraction layer, measurement framework and -library (See figure 4.1)
Compile, Mutate & Link	Compile all necessary objects, create mutations of the benchmarks, and link everything together to create executable binaries
Binaries	Executable Linux binaries
Execute	Run the benchmarks in gem5 and on the Raspberry Pi 2
Result Data	Collected result data. This includes cycle- and instruction counts, compile- and runtime errors.
Evaluate & Analyze	Statistical evaluation of the results, computation of an execution time model, application of said model to the results of the mutated benchmarks.
Detection Data	Results of applying the execution time model onto the empirical results.

4.2 Fault model

Due to their invasive nature Runtime-Injection Frameworks disturb the workload and can have an impact on the measured execution times[HTI97]. To avoid these (unpredictable) perturbation overheads, the compile-time- and pre-runtime injection approach was chosen, meaning that the program-under-test will be modified before the execution of the actual binary on the target machine. (c.p. section 2.3)

The premise of this thesis implies the use of algorithms with (mostly) data-independent execution times (c.p. section 2.4), so the manipulation of input data would not be very constructive. Therefore the faults will be injected directly into the benchmarks' instruction bitstream. (the `.text` segment [San97]). For this thesis the fault model will be limited to *single-bit flips*, i.e. the inversion of a single bit in the program code, thereby simulating e.g. a faulty memory cell, data corruption in a flash cell or a transient glitch on a bus. (Figure 4.3). This model is similar to the work presented in [ASJ13] and a commonly used approach [JSM07].

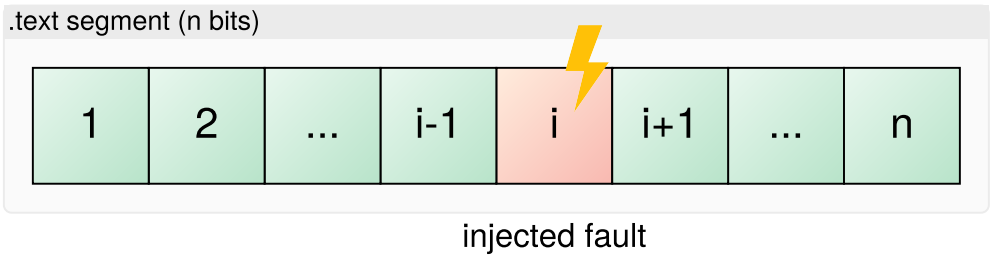

Figure 4.3: Bitstream with fault injected at bit *i*

For each benchmark tested, *every bit* in the actual benchmark's program code will be flipped once, and the resulting binary measured – thereby giving exhaustive coverage of all possible (single-bit) mutations.

4.3 Perf_event

Whereas the gem5 simulator can natively measure cycles and instructions spent during the execution (but only for the whole program), the so-called *perf_event* framework – a part of the Linux kernels Performance Counters for Linux (PCL) subsystem – was chosen for the measurements on the Raspberry Pi 2. Perf_event is a Linux-native abstraction layer for Performance Monitoring Units (PMUs) embedded in many contemporary processor designs. [Bak15; Gon15]. It is included in the Linux kernel since version 2.6.31[VCL11], introducing the `sys_perf_event_open()` syscall. The goal of the perf_event framework is the abstraction and encapsulation of performance counters from the actual hardware and providing a generic, cross-platform, and low-overhead interface for using them. The available

performance counters are abstracted and access is provided via a file descriptor returned by `sys_perf_event_open()`. To get started with the framework there also exists the `perf`[1] command line tool, which is available in any major Linux distribution.[2]

For the use in this thesis, the `perf_event_open` syscall was abstracted into a small library — *perflib* (listing B.3) — providing an RAII based C++ wrapper class, exposing just the functionality needed for the planned measurement tasks. The *PerfEvent*'s class API is reduced to:

Constructor, creating the needed data structures and initializing them with desired configuration parameters.

`Start()` starts the measurement process.

`Stop()` stops the measurement process.

`Count()` queries the measured value.

4.4 Benchmark Framework

As outlined in section 4.1/figure 4.1, the benchmark framework isolates the specific benchmark code from the common measurement code and defines the program flow of the measurement process. (See figure 4.4) A benchmark's program flow consists of:

1. Parsing command line arguments — arguments, just an optional integer, allow the selection of a specific pseudo-random input sequence.
2. Mode selection — chooses between *MUTATION* and *JITTER* mode. The former gets selected whenever no command line argument was passed, running the benchmark with a predetermined input data set and comparing the produced output to a known correct result. The latter instructs the benchmark abstraction layer to create a pseudo-random input (and discard the output). This mode is used do stochastically determine the bandwidth of possible execution times in the absence of faults. (Also compare to figure 2.4)
3. Instantiation of the measurement objects — On the Raspberry Pi 2 the PMU gets initialized and set up to perform the needed measurements, creating a cycle- and an instruction counter. On gem5 this step performs no operation as the simulator does not emulate a PMU.

[1] Part of the Linux kernels user space tools

[2] man page (2) `perf_event_open` — `http://man7.org/linux/man-pages/man2/perf_event_open.2.html`

4. Initialization — Performs one-time initializations within the benchmark abstraction layer.
5. Preparation — Calls the benchmark abstraction layer to prepare a specific input data set.
6. Start Measurement — starts the previously setup performance counters.
7. Run — Calls the actual (possibly mutated) benchmark code under test.
8. Stop Measurement — Stops the performance counters

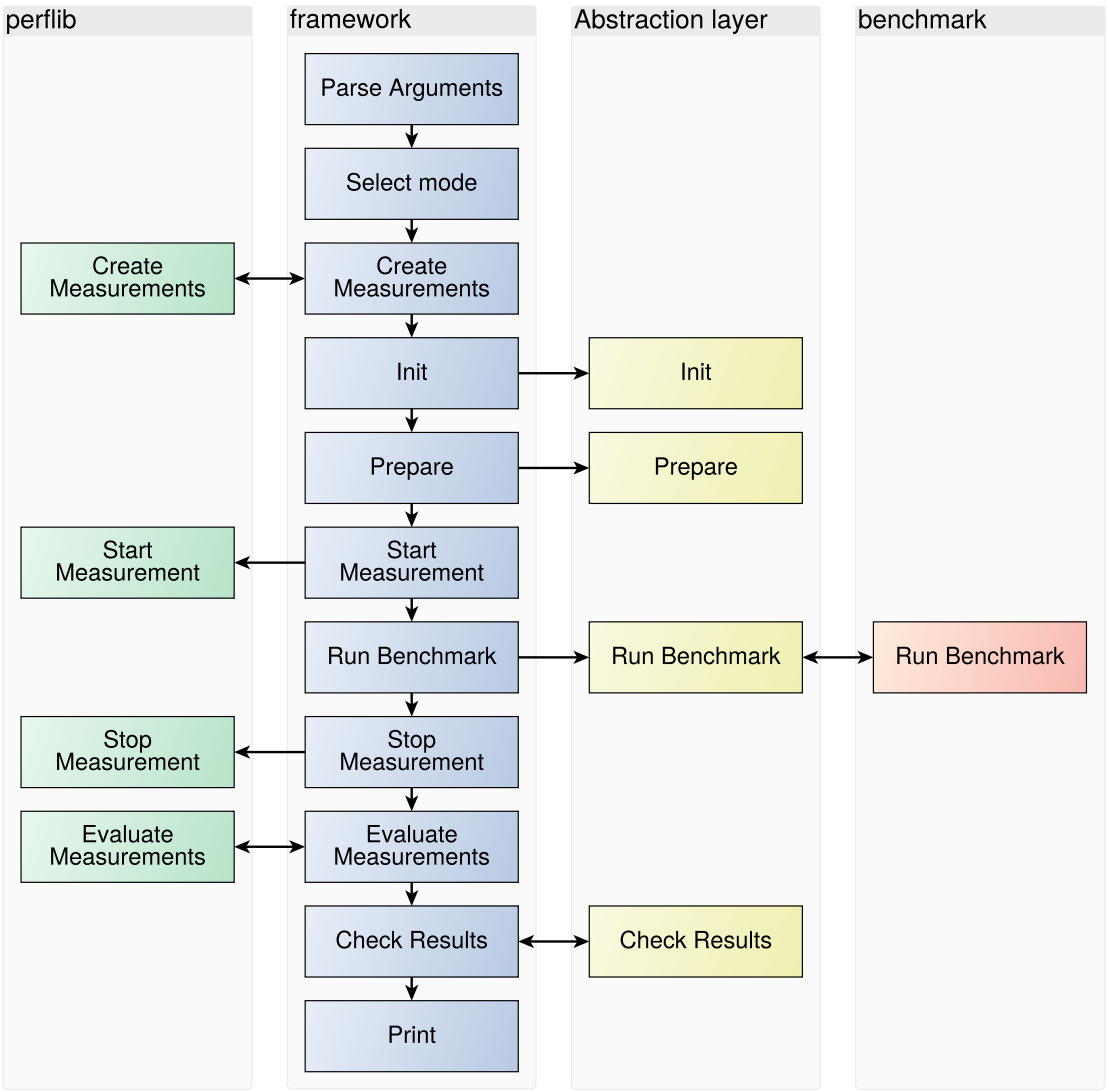

Figure 4.4: Benchmark Framework

9. Evaluate Measurement — Reads the counters and stores the values.
10. Check results — if in *MUTATION* mode, then this step calls the benchmark abstraction layer to check the computed results against the expected ones.
11. Print — Prints the measured values and, if applicable, the check result, to the console.

4.5 Measurement Framework

The measurement framework implements the concept outlined in section 4.1/figure 4.2. A detailed overview of the framework is shown in figure 4.5.

In the *Mutator* stage the source code is compiled into object files (`.o`). The benchmark framework (section 4.4) gets compiled twice — once normal, the other time in the *NOOP* configuration, being identical to the normal one, but not performing the *Run Benchmark* step (compare to figure 4.4) – all other calls into the benchmark abstraction layer still exist. This special version is later on used to establish the baseline overhead inherent to the measurement approach.

The benchmark's object file itself is then passed into the *Mutate* step, where the Fault Model (section 4.2) gets applied using the *flipbit* tool (see section 4.5.1), creating a set of mutated benchmark object files (*MUTANTS*) – exactly one for each bit in the original binary object.

In the *Analyzer* stage three types of executable binaries are built and executed:

The unmodified (*Original*) benchmark object file is linked to the *NOOP* framework, resulting in *NOOP binary*. This binary is executed in gem5 and on the Raspberry Pi 2 to establish the baseline measurement overhead.

The *Original* benchmark is linked to the regular framework library resulting in the *Original binary* and then executed to measure the unmodified programs execution-time. The same executable is also run in *JITTER* mode (See section 4.4) with (pseudo-)random input data, to collect a data set used later on for establishing the detection thresholds.

A static check is performed on each *MUTANT* object, checking for illegal instructions (treated as a *CRASH* result). If the check succeeds, the mutated object is linked to the benchmark framework, creating a *MUTANT binary*. This binary file is then also transferred to then target platforms, executed, and the results logged into the result database.

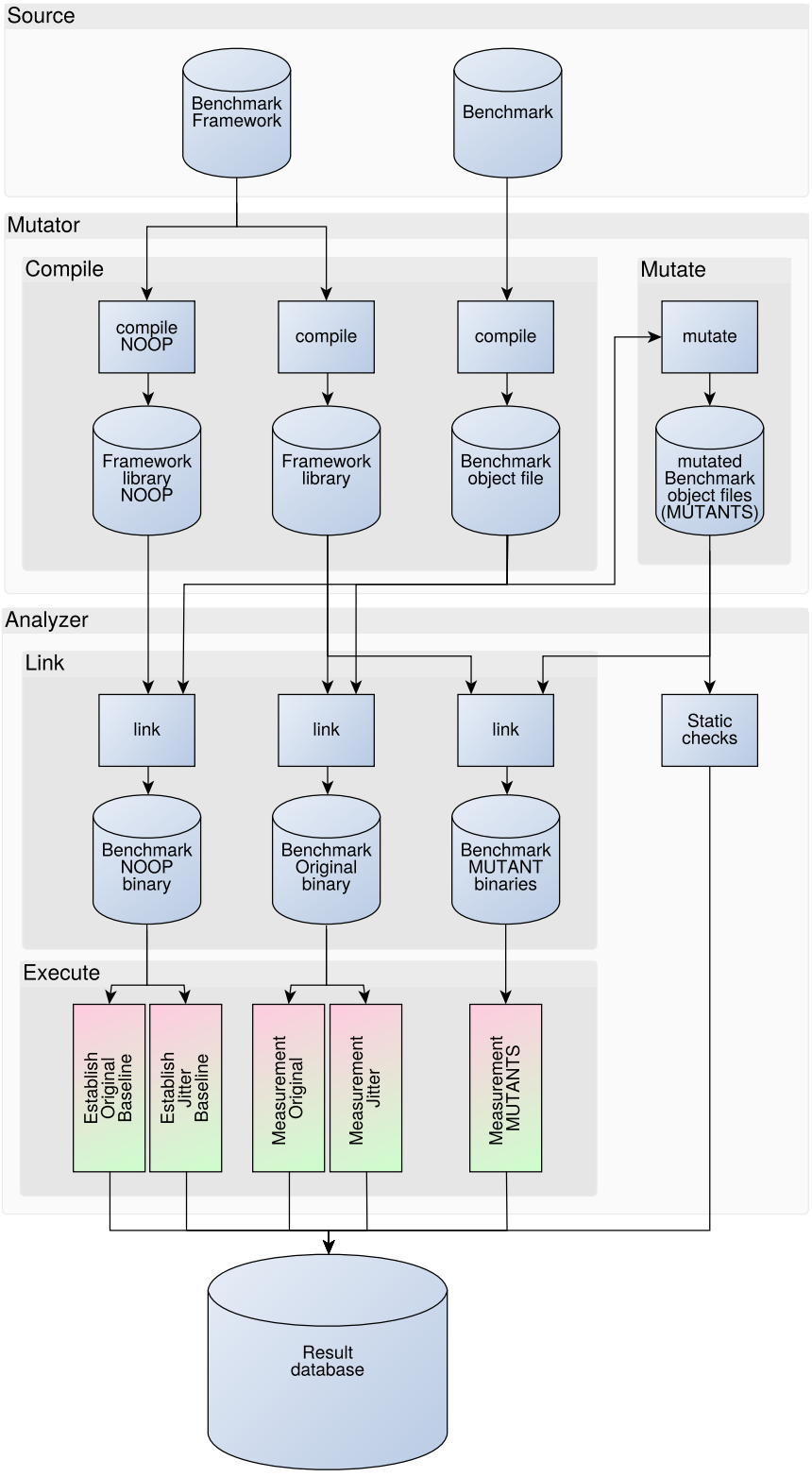

Figure 4.5: Measurement Framework

4.5.1 flipbit

Flipbit is a simple tool written in C++(See listing B.4). Its entire functionality consists of flipping a single bit within a given file. In this thesis, it is used to perform the mutation step (See above) by injecting single bit faults into the benchmarks binary code. (cp. section 4.4)

Syntax:

```
flipbit 0.0.0
Flips a single bit in a file
Allowed options:
  -h [ --help ]         produce help message
  -i [ --input ] arg    input file
  -o [ --output ] arg   output file
  -b [ --bit ] arg      bit to modify
  -v [ --verbose ]      verbose
```

CHAPTER 5

Infrastructure Description

Introduction

For reproducibility reasons this chapter gives a short overview of the hardware and software used in this project.

5.1 Raspberry Pi 2

A Raspberry Pi 2 Model B [Fou15] single board computer was used as the main target platform for this thesis. Released in February 2015, it features a Broadcom BCM2836 System-on-a-Chip (SoC) with a quad-core Cortex-A7 processor and 1 GiB of LPDDR2 RAM.

The Raspberry Pi 2 was chosen because a) similar processors are used in a lot of embedded applications, b) it is an open system with almost all parts of the software available as open source, c) it is widely used in a lot of prototyping applications and d) it is available as a COTS component for a reasonable price.

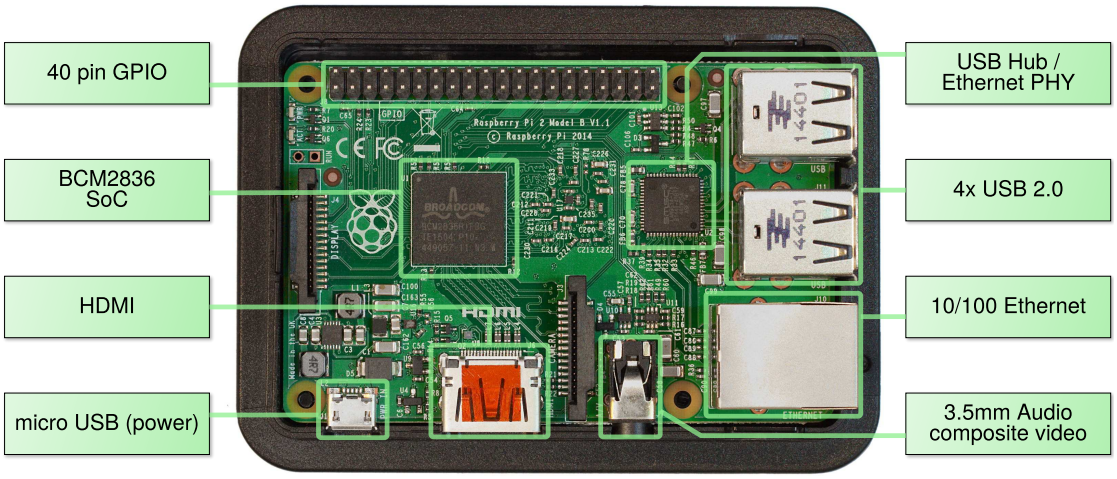

Figure 5.1: Raspberry Pi 2 Model B

SoC	Broadcom BCM2836
CPU	900MHz quad-core ARMv7 Cortex-A7
GPU	Broadcom VideoCore IV
RAM	1GiB LPDDR2 SDRAM
Ethernet	Onboard 10/100 Ethernet, RJ45 jack
USB	Four USB 2.0 ports
I/O	40 pin GPIO
Video output	HDMI, composite video
Audio output	3.5 mm jack, HDMI
Storage	microSD card slot

Table 5.1: Raspberry Pi 2 Model B hardware specifications

5.1.1 CPU Core

The Broadcom BCM2836 SoC includes four ARMv7 Cortex™-A7 MPCore™CPU cores. It is a fully compliant implementation of the ARMv7-A instruction set, featuring an *in-order pipeline* (figure 5.3) with branch prediction, dedicated Harvard style Level 1 cache (i.e. split instruction- and data cache) with a Memory Management Unit (MMU), shared Level 2 cache (figure 5.2) and a VFPv4-D32 FPU with NEON technology. [ARM13]

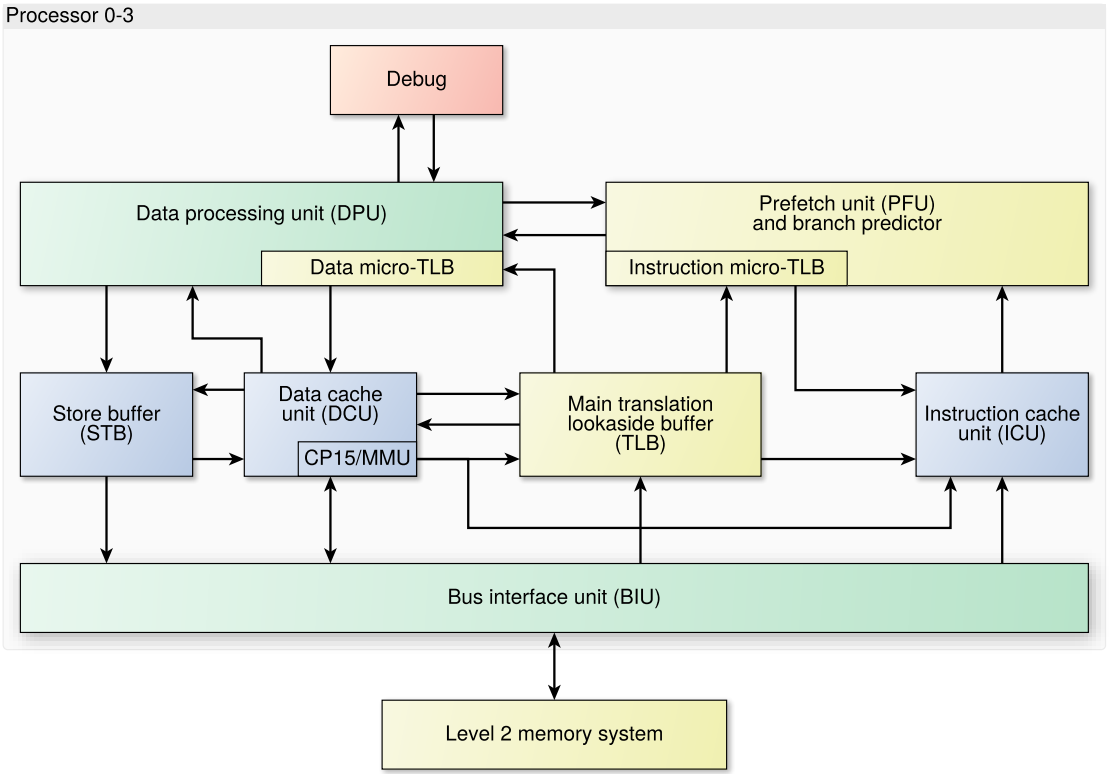

Figure 5.2: Cortex-A7 top-level design [ARM13]

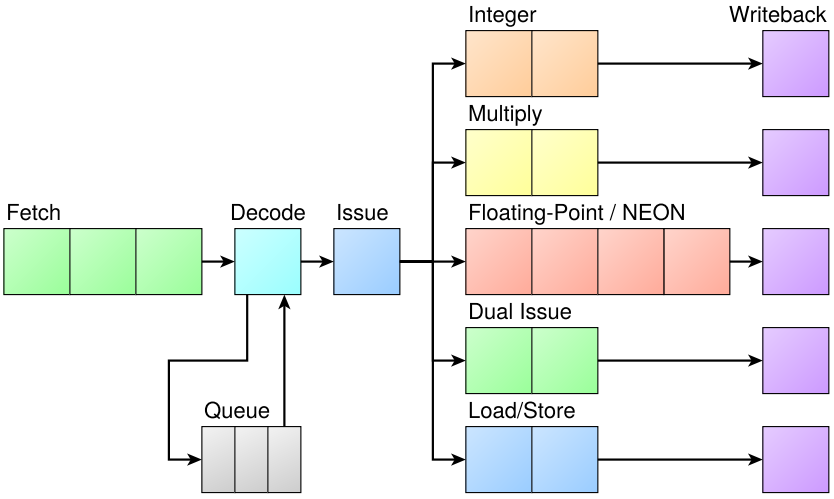

Figure 5.3: Cortex-A7 Pipeline [Jef11]

5.1.2 Operating system

Several factors led to the choice of *Raspbian* as the operating system for the Raspberry Pi 2:

- It is officially supported by the Raspberry Pi Foundation.
- It is a distribution optimized for the use on Raspberry Pi hardware, including all necessary drivers and management tools.
- Raspbian is based on Debian[1], so existing experience with package-management-, configuration-, and management tools can be applied.

The OS version used is *Raspbian 2015-05-05*[2] (Based on Debian 7.8 Wheezy), which was the latest stable version available at the beginning of the project. The Linux kernel and the board firmware where later on updated to version *4.1.5-v7+*[3] using the `rpi-update`[4] tool.

5.1.3 Configuration

Some adjustments to the default Raspbian configuration where made:

- The OpenSSH Secure Shell (SSH) server was installed through the package manager (packet: `openssh-server`) to enable remote access to the Raspberry Pi 2 over the network.
- The `isolcpus=3` kernel command line argument was added to `/boot/cmdline.txt`. This results in CPU core #3 being excluded from the task scheduler, making it available only to directly assigned tasks (using `taskset`). As the only tasks assigned to the CPU core will be our benchmarks, this means that they will be able to run uninterrupted.
- The Raspberry Pi 2s clocks were fixed and tweaked to reduce the impact of the memory system and dynamic power saving. The base clock was fixed to 250 MHz, the CPU clock was also set to 250 MHz (fixed), and the Random Access Memory (RAM) clock was fixed to 500 MHz. This was achieved by adding the following lines to `/boot/config.txt`:

```
#lower␣minimal␣arm␣freq
arm_freq_min=250
core_freq=500
core_freq_min=500
sdram_freq=500
sdram_freq_min=500
```

[1] `https://www.debian.org/`
[2] Available at `https://downloads.raspberrypi.org/raspbian/images/raspbian-2015-05-07/`
[3] commit 36237afd3c27ea735172986ebde7fa5fcc6ca6e7
[4] `https://github.com/Hexxeh/rpi-update`

```
over_voltage_sdram_p=4
over_voltage_sdram_i=0
over_voltage_sdram_c=0
```

and enforcing a fixed CPU clock in `/sys/devices/system/cpu/cpu`*n*`/cpufreq/scaling_setspeed`. (for n in 0..3)

- All unnecessary system services where disabled: `avahi`, `console-kit`, `dbus`, `ntp`, `polkit`, and `triggerhappy`. This was done to avoid any possible impact of unneeded processes running.

5.2 gem5

For simulation the *gem5* simulator [BBB11] was used. Gem5 is a modular simulator, extensively used in system architecture and microarchitecture research, and endorsed by several large companies, including AMD, ARM, IBM and Intel. Gem5 is a modular simulator platform, combining the best aspects of its predecessors, *M5* [BDH06] and *GEMS* [MSB05]. It includes support for different Instruction Set Architectures (ISAs) (currently Alpha, ARM, MIPS, POWER, SPARC and x86), multiple interchangeable CPU models of varying detail grade, power modeling, SystemC co-simulation, Python integration, and much more.[5]

Gem5 is used in Syscall Emulation (SE) mode, where it provides an emulated Linux environment. The emulated processor core is an ARM Cortex-A9 ARMv7, with additional support for Thumb, Thumb-2, VFPv3 and NEON instruction sets

While gem5 is not a perfect match to the Raspberry Pi 2, it is still reasonably close, and — more important — accurate [BGO12]. The largest differences arise when executing memory or FPU intensive workloads, as gem5 uses a simplified model here.

The gem5 version used in this thesis is gem5 version 2.0, built from source. (gem5-stable repository[6] — commit 10874:629fe6e6c781)

A trivial custom patch (Listing B.1) was applied on top of the official source code, to remap the `sys_perf_event_open` syscall, needed for the benchmarking framework, from an unimplemented function (which results in termination of the simulator) to an ignored function (resulting just in a warning when called), thereby allowing to run the benchmark binaries in gem5 without the need for modification.

Additionally a more extensive patch (Listing B.2), based on the work presented in [ECC14; ECC15], was added — extending gem5 with a Raspberry Pi 2 CPU model, mimicking the ARM-Cortex-A7 processor core found on the Raspberry Pi 2. While this model still is not spot-on, it yields results much closer to the real hardware than any of the default CPU models.

[5] See `http://gem5.org/` for up to date details.
[6] Available at `http://repo.gem5.org/gem5-stable`

5.3 Toolchain

Several tools are used on the control PC for building, running and evaluating the benchmarks:

- Host Operating System — OpenSUSE 13.2 x64[7]
  Used as the operating system for development, running the measurement environment, controlling gem5 and the Raspberry Pi 2.
- Compiler ARM — gcc-linaro-4.9 Version 2014.11-x86_64_arm-linux-gnueabihf[8]
  Used as a cross compiler for the ARMv7 platform.
- Compiler PC — gcc/g++ (SUSE Linux) Version 4.8.3 20140627[9]
  Used as the compiler for the needed host side tools.
- Build system — CMake 3.3.2[9]
  Build system used for building all created tools and benchmarks. Provided by the host OS.
- Python 2.7.8[9]
  Used for evaluating the measurement results. The libraries used are:

Library	Version	Description
matplotlib	1.5.1-104.17	A Python 2D plotting library.
NumPy	1.10.4-108.1	General-purpose array-processing package.
pandas	0.17.1-21.1	A Python package providing data structures for working with relational data.
Scipy	0.17.0-54.3	Open-source software for mathematics, science, and engineering.
Seaborn	0.6.0-1.1	Library for making attractive and informative statistical graphics in Python.
SQLAlchemy	1.0.11-88.1	An Object Relational Mapper (ORM) providing a flexible interface to SQL databases.

- Shell — GNU bash, version 4.2.53(1)-release[9]
  The measurements where automated using bash scripts.
- Database — sqlite3 version 3.8.6 2014-08-15[9]
  Used as the database to store measurement results.

[7] `https://www.opensuse.org/`
[8] `https://www.linaro.org/`
[9] Provided by the host OS

CHAPTER 6

Microbenchmarks

Introduction

This chapter details the microbenchmarks used and the reason for choosing them. All presented microbenchmarks are purely computational so they are free of any communication, synchronization or other blocking during their execution, i.e. they are *simple tasks*, defined by Kopetz as having no synchronization points within the task, so the tasks' execution time is not directly dependent on other tasks and can be determined *in isolation*.[Kop11]

6.1 List of microbenchmarks

Name	Description
nop	Dummy benchmark, performing a single `NOP` instruction. This benchmark can not produce an error, as it has no return value, however, it can fail when faults get injected into its code. As it consists of just two instructions, a `NOP` and a returning `BX` from the function call, it serves as a trivial baseline to quickly test that the whole evaluation framework is working as intended.
long_nop	Dummy benchmark, performing 100 `NOP` instructions. This benchmark is identical to the *nop* benchmark, with the only difference being the number of `NOP` instructions in the benchmark function.
bubblesort	Sorts an array of integers using the bubble sort algorithm. This is a classical speed optimized version.

bubblesort_wcet	Sorts an array of integers using the bubble sort algorithm. The algorithm is a WCET optimized version based on the single-path implementation (`bubble2`) presented in [Pus03].
heapsort	Sorts an array of integers using the heapsort algorithm[1]. Heapsort is an in-place, unstable, constant memory sorting algorithm, originally invented by Williams in 1964[Wil64] and improved with an in-place version by Floyd[Flo64]. Its runtime is bounded by $O(n \log n)$.
quicksort	Sorts an array of integers using the quicksort algorithm. Originally invented by Hoare[Hoa61]. Its average runtime is $O(n \log n)$, the worst case performance is $O(n^2)$. This implementation uses the *median-of-three* pivot selection strategy proposed by Sedgewick [Sed78].
selectionsort	Sorts an array of integers using the selection sort algorithm[2]. Selection sort is a simple in-place comparison sort algorithm with a runtime behavior of $O(n^2)$. Due to its low overhead and constant memory footprint it is sometimes used to sort small (sub-)arrays (less than $\sim$20).
shellsort	Sorts an array of integers using the Shellsort algorithm[Sed96].[3] This special variant uses the *Incerpi-Sedgewick* gap-sequence [IS83] with a runtime behavior of $O(n^{1+\sqrt{8 \ln 5/2 / \ln n}})$. Shellsort is commonly used in low memory systems as its memory footprint is constant and it does not use the call stack.
binsearch	Performs a binary search in a sorted list of `(int,int)` key,value pairs. Binary search has a worst case (and average case) performance of $O(\log n)$, and a best case performance of $O(1)$. This particular implementation was adapted from the *Mälardalen WCET Benchmarks* [GBE10][4].

[1] `http://rosettacode.org/wiki/Sorting_algorithms/Heapsort`
[2] Adapted from [Sed97]
[3] `https://www.cs.princeton.edu/~rs/shell/shell.c`
[4] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

binsearch_wcet	A WCET optimized version of the *binsearch* benchmark. The data dependence of the loop was factored out, so it performs a constant number of loop iterations for a given input length. This eliminates the $O(1)$ best case and leads to a constant runtime performance of $O(\log n)$.
duff	Copies a block of data using *Duff's Device*.[5] This benchmark's semantics are identical to `memcpy()`, but it works by employing a length dependent jump into a `switch` statement, essentially a form of loop unrolling. This particular implementation was adapted from the *Mälardalen WCET Benchmarks* [GBE10][6].
strstr	Performs a substring search in a block of text, using a reimplementation of the C-standard `strstr()` function. This is a trivial iterative version with a worst-case runtime of $O(nm)$, where n denotes the length of the pattern ("needle") and m denotes the length of the text ("haystack").
fibcall	Non-recursive 64-bit computation of the Fibonacci number F_n. This particular implementation was adapted and extended to 64 bits from the *Mälardalen WCET Benchmarks* [GBE10].
prime	Test for primality using a simple *trial division* algorithm. This involves checking whether any prime integer m between 2 and $\sqrt{n}$ evenly divides n. (without a remainder) This particular implementation was adapted from the *Mälardalen WCET Benchmarks* [GBE10].
prime_wcet	Test for primality using a simple *trial division* algorithm. This version was modified to achieve (nearly) constant runtime for all 32-bit integers by a) removing the early abort condition for non-prime numbers b) not testing up to $\sqrt{n}$ but up to $2^{16}-1$ (the largest possible square root of a 32 bit integer) c) not using the variable-time hardware *modulo* instruction, but a constant cycle software implementation.

[5] See `http://www.lysator.liu.se/c/duffs-device.html` for the history of Duff's Device

[6] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

sqrt_micro	Computes the square root of a 32-bit floating point number by using Taylor series approximation. Adapted from the *Mälardalen WCET Benchmarks*[GBE10].
sqrt_micro_wcet	Computes the square root of a 32-bit floating point number by using Taylor series approximation. This version is similar to *sqrt-micro* but optimized towards constant execution time by removing any early abort conditions and performing a constant number of Taylor steps.
sqrt_micro_wcet_neg	Computes the square root of a 32-bit floating point number by using Taylor series approximation. This version is based on the *sqrt_wcet* benchmark, but with an added initial check for negative numbers. The intention was to create a benchmark with an execution-time profile showing distinct peaks.
matmul	Performs a 32-bit integer matrix multiplication. This consists mainly of three nested `for` loops. Taken from the *Mälardalen WCET Benchmarks* [GBE10].
matmul_float	Performs a 32-bit floating point matrix multiplication. Same as *matmul*, but using floating points values.
aes	An implementation of AES[NIS01] running in ECB(Electronic Code Book) mode. Encrypts a single 16 byte block with a 128 bit key, and then decrypts it again. This benchmark uses a public-domain C99 implementation of the AES algorithm.[7]
crc32	Performs a CRC-32 calculation on a 2 KiB block of data. It uses a lookup table free implementation of the CRC-32 algorithm. This version is compatible to the one presented in the IEEE 802.3 standard (Ethernet)[IEE12]. The used implementation is from an implementation found in [War12].[8]
fft	Performs a Fast-Fourier-Transformation and an Inverse-FFT on a block of data using complex floating point arithmetic. This benchmark uses an in-place version of the recursive *Cooley–Tukey* FFT algorithm[CT65].[9]

[7] Available at `https://github.com/kokke/tiny-AES128-C`

[8] `crc32b` from `http://www.hackersdelight.org/hdcodetxt/crc.c.txt`

[9] See `http://rosettacode.org/wiki/Fast_Fourier_transform`

fir	Filters an input signal using an integer-based *Finite-Impulse-Response* filter. This algorithm is a common task among DSPs and other embedded systems. Taken from the *Mälardalen WCET Benchmarks* [GBE10].

6.2 Analysis

To give an overview of the diversity presented by the selected microbenchmarks, they were compiled, analyzed, and the used instructions roughly classified according to the *ARM Architecture Reference Manual*[ARM12, Chapter A5]:

Type	Description	Reference
BRANCH/LINK	Branch and Link instructions also known as *jumps*. This type of instructions is used to implement branches, loops and function calls. These instructions are of particular interest in this thesis, as it is easy to see that errors in this instructions can greatly affect execution time.	A5.5, A5.7
CONSTANT	Constants embedded in the `.text` segment Many single value constants are embedded within the code itself by the GNU compiler.	—
COPROCESSOR	ARM coprocessor instructions This type of instructions is used to communicate with (possible) coprocessors.	A5.6, A5.7
DATA	Data processing instructions This includes ALU operations, immediate loads and register-register moves.	A5.2
LOAD/STORE	Load/Store instructions These instructions are used to load/store data from/to the main memory.	A5.2, A5.3
MEDIA	Media Instructions This type of instructions is similar to DATA instructions. They implement some simple SIMD operations (parallel 8/16 bit), byte reversal, saturated arithmetic, but most important – integer division (`SDIV/UDIV`).	A5.4
SIMD/FP	SIMD(NEON) and FPU instructions Integer and floating point SIMD and regular FPU instructions.	A5.6, A5.7

SPECIAL	Memory hints, preload- and other miscellaneous unconditional instructions.	A5.7
SUPERVISOR	Supervisor calls (“Software Interrupts”), e.g. for syscalls. Software can use this type of instructions to call an operating-system service.	A5.6

The analysis of how many instructions of which type are present in the benchmarks are shown in table 6.3 and figure 6.1.

Not issued by the compiler were COPROCESSOR– (no coprocessor present), SPECIAL– (no preload hints, . . .) and SUPERVISOR instructions (no syscalls, as this are pure computational microbenchmarks).

BRANCH/LINK instructions have a share of about 3-20%, depending on the algorithm. Noticeable is the tendency of WCET optimized versions to contain significantly fewer instructions of this type.

DATA instructions take up a significant part of every algorithm — except for the floating point based ones, where it is split with SIMD/FP.

It is interesting to note that *aes* has the highest ratio of LOAD/STORE instructions – hinting at the number of table lookups performed by the algorithm in the substitution step.

	BRANCH/LINK	CONSTANT	COPROCESSOR	DATA	LOAD/STORE	MEDIA	SIMD/FP	SPECIAL	SUPERVISOR
nop	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0
long_nop	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0
bubblesort	11.8	0.0	0.0	58.8	29.4	0.0	0.0	0.0	0.0
bubblesort_wcet	12.5	0.0	0.0	50.1	37.4	0.0	0.0	0.0	0.0
heapsort	18.0	0.0	0.0	50.0	32.0	0.0	0.0	0.0	0.0
quicksort	13.6	0.0	0.0	63.6	22.7	0.0	0.0	0.0	0.0
selectionsort	18.2	0.0	0.0	54.5	27.3	0.0	0.0	0.0	0.0
shellsort	11.6	2.3	0.0	48.8	37.2	0.0	0.0	0.0	0.0
binsearch	15.8	0.0	0.0	57.9	26.3	0.0	0.0	0.0	0.0
binsearch_wcet	7.1	0.0	0.0	57.1	35.7	0.0	0.0	0.0	0.0
duff	3.6	14.3	0.0	48.2	32.1	1.8	0.0	0.0	0.0
strstr	20.0	0.0	0.0	46.7	33.3	0.0	0.0	0.0	0.0
fibcall	13.6	9.1	0.0	40.9	4.5	0.0	31.8	0.0	0.0
prime	11.5	0.0	0.0	80.8	3.8	3.8	0.0	0.0	0.0
prime_wcet	10.4	2.1	0.0	75.0	12.5	0.0	0.0	0.0	0.0
sqrt_micro	9.4	6.2	0.0	25.0	0.0	0.0	59.4	0.0	0.0
sqrt_micro_wcet	6.7	6.7	0.0	20.0	0.0	0.0	66.7	0.0	0.0
sqrt_micro_wcet_neg	10.0	5.0	0.0	20.0	0.0	0.0	65.0	0.0	0.0
matmul	11.1	0.0	0.0	63.0	25.9	0.0	0.0	0.0	0.0
matmul_float	11.5	0.0	0.0	57.7	11.5	0.0	19.2	0.0	0.0
aes	12.5	3.1	0.0	37.1	46.8	0.5	0.0	0.0	0.0
crc32	16.7	5.6	0.0	44.4	27.8	5.6	0.0	0.0	0.0
fft	9.9	3.0	0.0	44.0	9.0	2.0	32.0	0.0	0.0
fir	11.1	0.0	0.0	58.3	27.8	2.8	0.0	0.0	0.0

Table 6.3: Instruction classification (%)

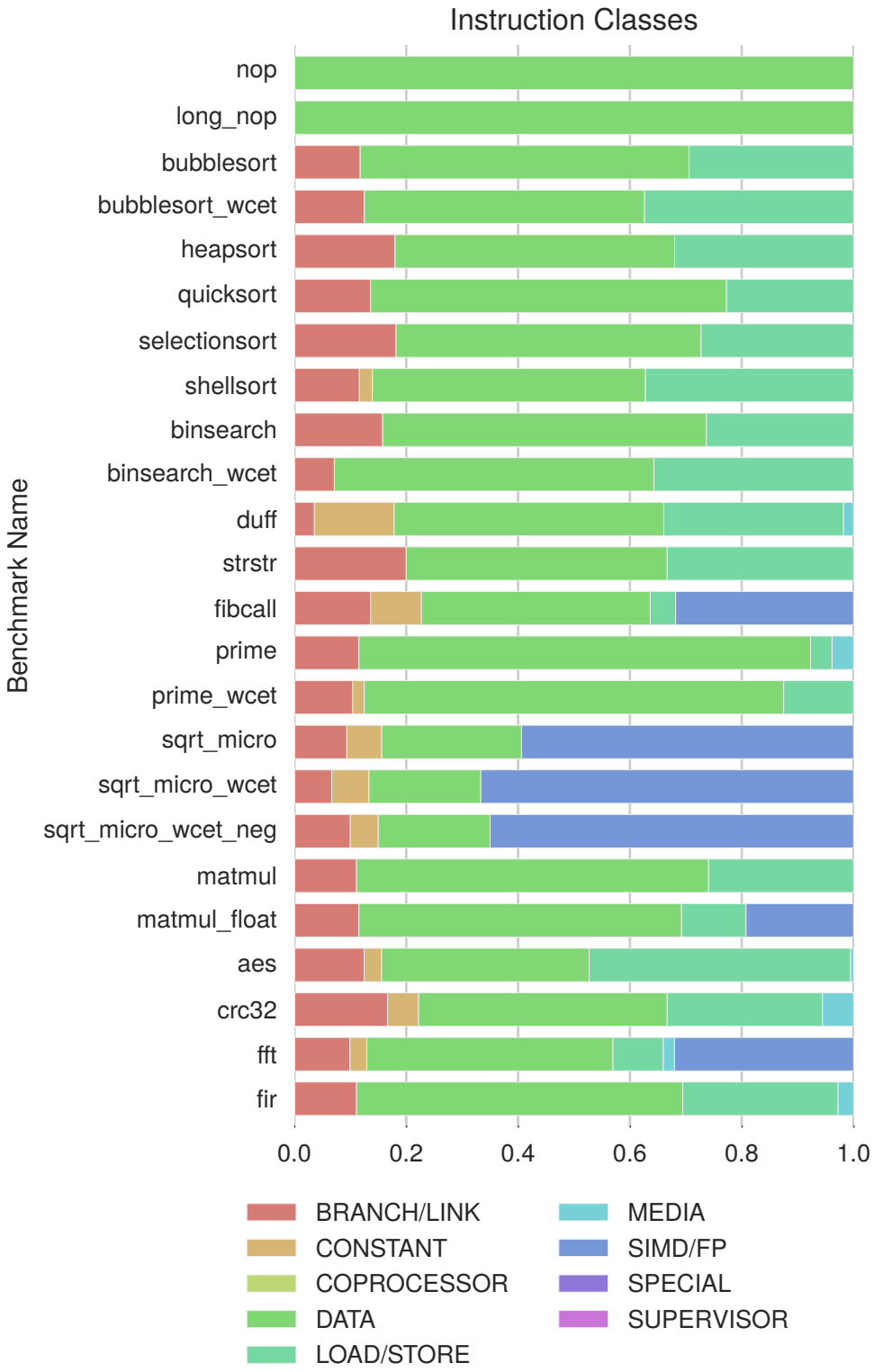

Figure 6.1: Instruction classification

CHAPTER 7

Evaluation

7.1 Fault Injection Results

As detailed in section 4.5, analysis and execution results of all mutant benchmarks are recorded into a result database, and then classified into categories:

Class name	Description of the class
CRASH	The benchmark crashes. This includes illegal- and undefined instructions, segmentation faults, preliminary calls to `abort()`, etc. pp. This type of fault is trivially detectable by defensive programming methods and evaluation of error codes, as the system was able to catch the error and report it.
ERROR	The benchmark terminates correctly, the computed result is *not* as expected. This is the most interesting fault category to observe, as these faults are *not* detected by classical watchdog designs, and can result in erroneous system behavior. This case is a prime example for silent data corruption.
OK	The microbenchmark terminates correctly, the computed result is as expected. The timing might still be off, or there could be other unintended side effects, e.g. a temporary result not stored to the intended memory location, overwriting data used by other tasks, so silent data corruption ca not be ruled out in this case, too.

TIMEOUT — Indicated that the benchmark did not terminate, and no result was produced.
The benchmark (probably) got stuck in an infinite loop. On gem5 the abort condition used is ten times the *original* binary's instruction count (including measurement overhead), on the Raspberry Pi 2 it is a hard 10 second time limit, enforced by the tool `timeout` (part of the GNU coreutils[1])
This class of faults can usually be detected by watchdog-timer designs.

The results of classifying the *MUTANT* runs accordingly are shown in figures 7.1 and 7.2 (detailed numbers in tables A.1 and A.2).

The first obvious difference between gem5 and then Raspberry Pi 2 is in the *TIMEOUT* class. Many cases leading to an endless loop on gem5 actually crash when executed on the Raspberry. This happens due to the looser memory protection implemented in the simulator, allowing memory accesses to complete that would crash running on a real OS. There are also some differences in handling ill-defined opcodes, which are executed by the real CPU core, but lead to a crash on gem5 (with message `panic: Attempted to execute unknown instruction`). This is most obvious in *nop* and *long_nop*.

It is also noteworthy that for every microbenchmark at least 30% of the injected faults did not lead to crashes (*CRASH*) or endless loops (*TIMEOUT*), and in quite some cases this number was above 60%. It is also interesting to note how large the share of the *OK* class actually is, and how widely distributed — ranging from about 5% for *aes* to 70% for *prime*. The distribution of these results is comparable to the results achieved by in Ayatolahi et al. [ASJ13].

[1] `http://www.gnu.org/software/coreutils/`

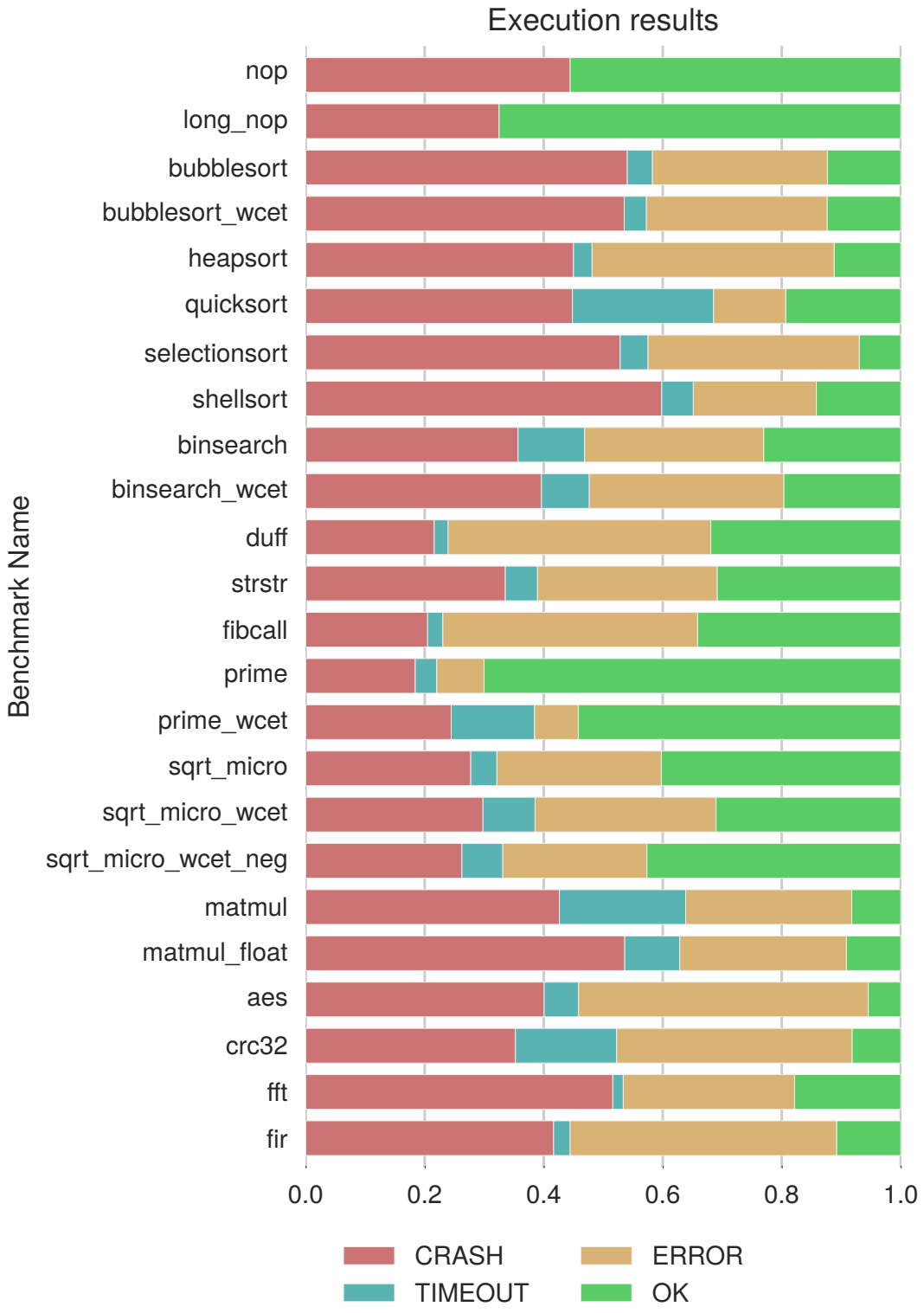

Figure 7.1: classified execution results – gem5

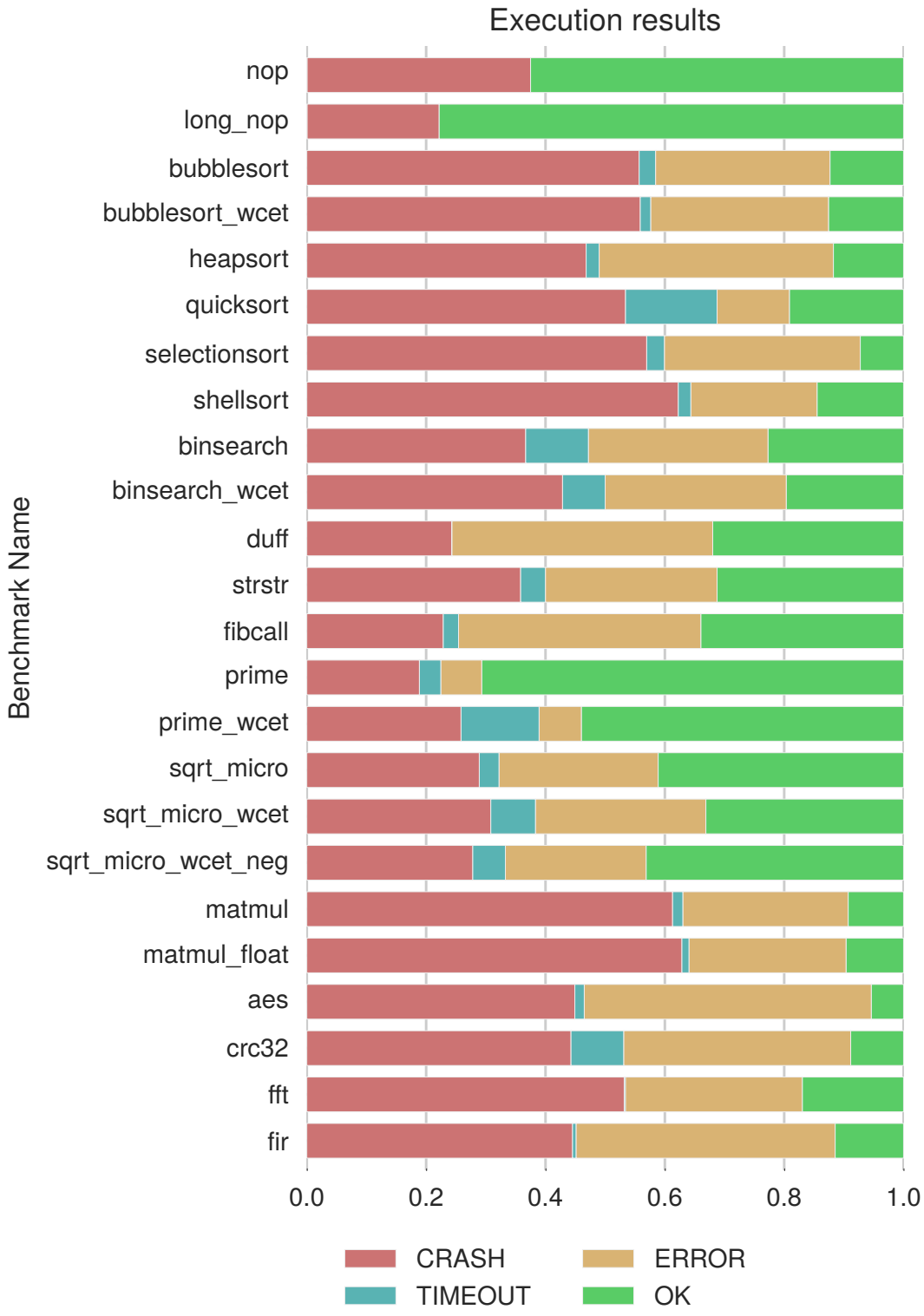

Figure 7.2: classified execution results – Raspberry Pi 2

7.2 gem5 vs Raspberry Pi 2

This section comprises a comparison between the gem5 simulator(section 5.2) and the Raspberry Pi 2(section 5.1). The aim of this comparison is to evaluate the feasibility of using the gem5 simulator as an execution environment – facilitating development testing. For this purpose the cycle- and instruction-counts measured in gem5 are compared to the *real world* Raspberry Pi 2 results.

The first obvious difference is that on the Raspberry Pi 2 the benchmark itself could be measured directly with the `perf_event` framework (section 4.3), whereas in gem5 only the values from the whole program run were available. By including *NOOP* runs (c.p. section 4.5) and working only with the difference values, this obstacle could be avoided. (Baseline removal)

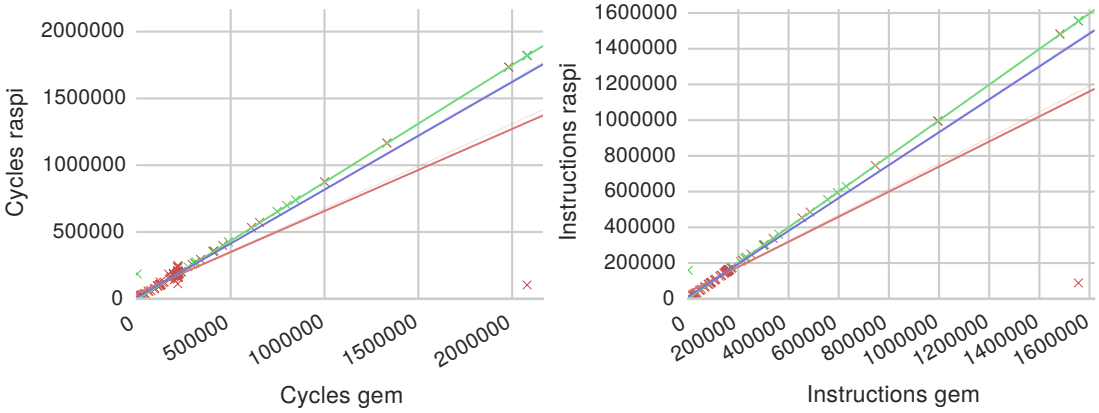

Figure 7.3: Regression: gem5 vs raspi — *fir*

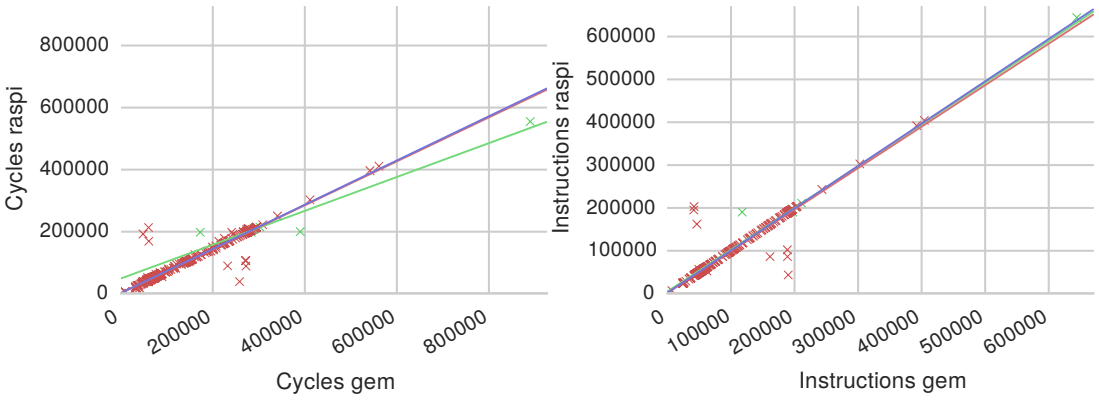

Figure 7.4: Regression: gem5 vs raspi — *heapsort*

Figures 7.3 and 7.4 exemplarily show comparison of all *MUTANT* runs for the *fir* and *heapsort* benchmarks, with the gem5 values on the horizontal- , and the Raspberry Pi 2 values on the vertical axes; cycles on the left side, number of instructions executed on the right. (For the full set of regression plots see Appendix A.1) Green marks denote *OK* results (c.p. section 7.1), red marks the *ERROR* class. The lines are linear regression models superimposed on the data set. The *green* line denotes the *OK* results, the *red* one the *ERROR* ones, and the *blue* line shows the combined regression model.

While the instruction measurements track perfectly for the *OK* cases – meaning that both platforms where executing the same code –, on the cycle values there is a) a bit of jitter — hinting the various caching systems and variable cycle instructions on the Raspberry Pi (also c.p. section 3.2) b) a benchmark dependent offset by a factor of about 10-20%; — indicating that the gem5-CPU model (Listing B.2) could need some more tweaking and the various instruction delays are not perfectly modeled; c) completely different behavior in some rare cases, probably due to some details of the Cortex-A7 not correctly emulated in the gem5 simulator.

The *ERROR* cases exhibit a small number of extreme outliers (see e.g. *fir*, Figure 7.3), right bottom corner). These can be explained as triggering undefined behavior (not well-defined instructions), where the gem5 simulator deviates from the actual implementation in the real ARM core.

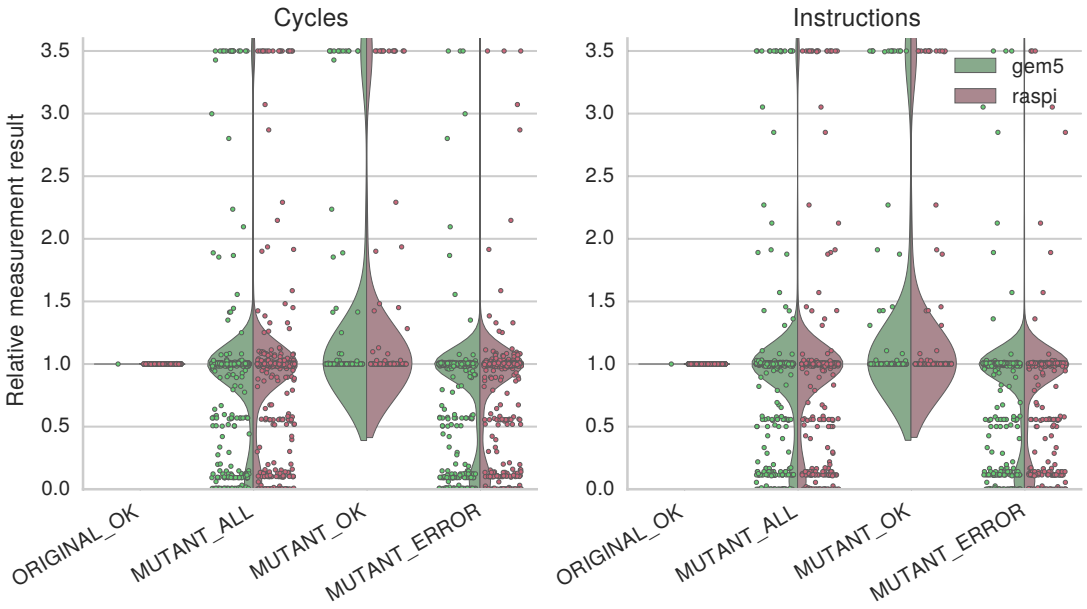

Figure 7.5: Violin Plot: gem5 vs raspi — *fir*

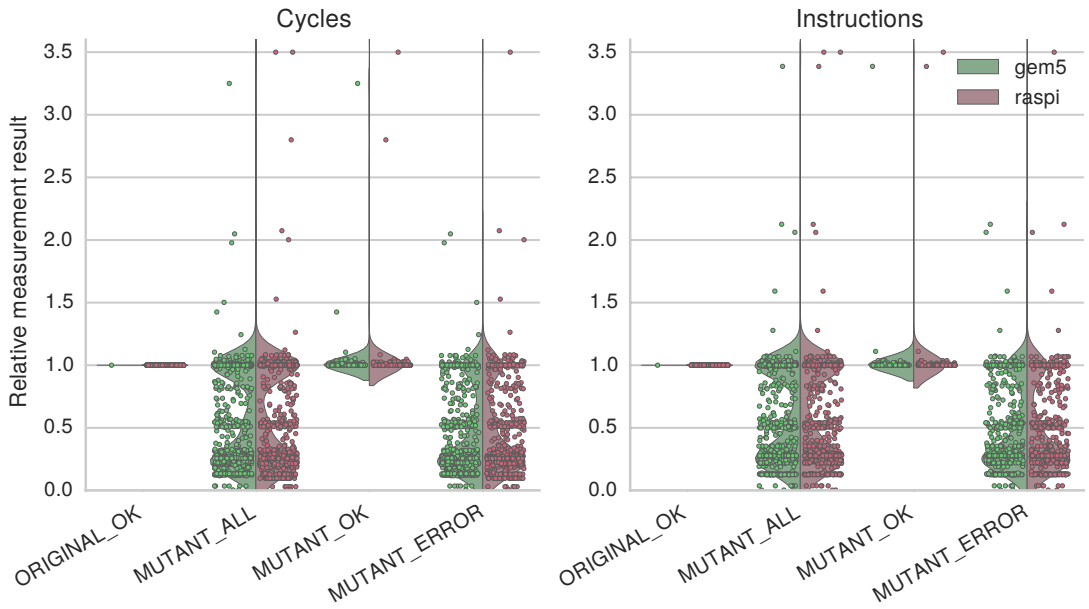

Figure 7.6: Violin Plot: gem5 vs raspi — *heapsort*

Figures 7.5 and 7.6 show the same data sets, normalized to the execution time of the *ORIGINAL* binary (also shown) on the respective platforms (green for gem5, purple for the Raspberry Pi 2), and split by result type. The dots represent the individual measurements, the *violins* are *Kernel density estimations*, i.e. an estimation of the underlying distribution. Again, the left chart shows cycles, the right one instruction counts. The categories used on the horizontal axis are:

ORIGINAL_OK are runs of the *Original* (unmodified) binary.

MUTANT_ALL are the runs of the *Mutant* binary.

MUTANT_OK is the subset of *Mutant* runs classified as *OK*

MUTANT_ERROR is the subset of *Mutant* runs classified as *ERROR*

Values larger than 3.5 times the base value are plotted at the 3.5 mark for readability reasons.

While it is still obvious that the Raspberry Pi values show more jitter (as explained before), the results of the two platforms are very similar and show that gem5 *can* be a feasible and reasonably accurate emulator for the Raspberry Pi 2, removing the constraints put on empirical methods by limited hardware availability and performance.

7.3 Detection Model

For the detection of errors a simple threshold based model is used (figure 7.7). Based on both – the *Original* and the (pseudo-random) *JITTER* measurements (c.p. section 4.5) – BCET/WCET bounds are estimated for both – cycles and instructions and used as lower/upper detection thresholds. The *MUTANTS* are then classified as *detected* if at least one of the values observed is outside the established bounds and as *undetected* otherwise.

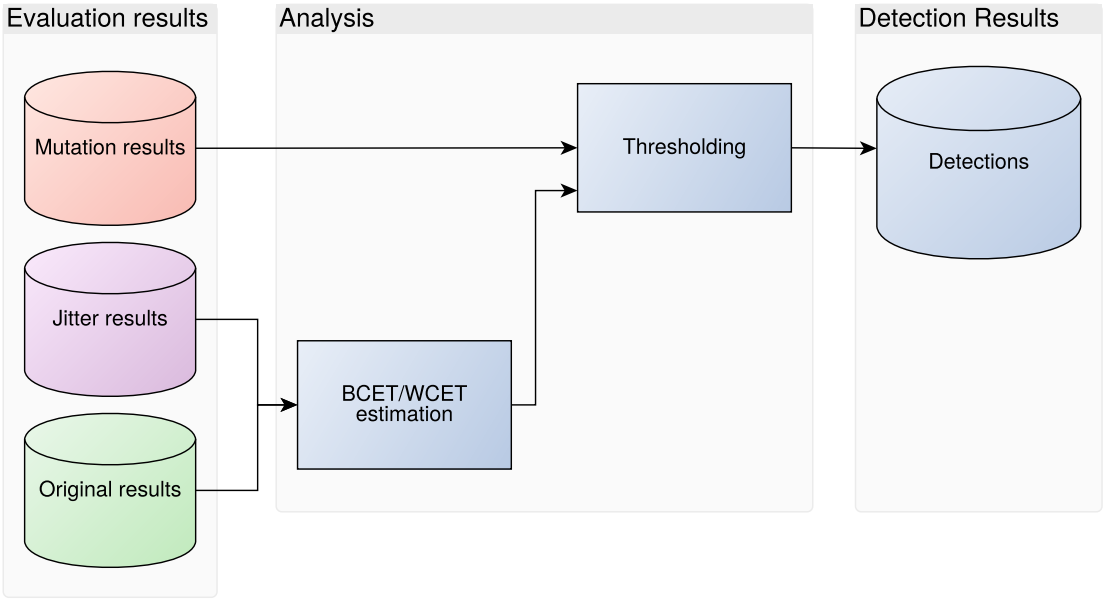

Figure 7.7: Detection model

In figure 7.8 the timing results for *sqrt_micro_wcet* are shown as an example.
The chart is split between *Cycles* (left) and *Instruction count* (right). *Jitter Measurements* (lilac) denotes the collected *JITTER* data – i.e. results of measurement run with (pseudo-)random input data; *Original Binary* (green) are the runs of the unmodified *ORIGINAL* binary with a known input-data set; *Mutants* (red) shows the data collected from the mutated binaries, with the same input-data set. For readability reasons extreme values are binned into a single class (the peaks at the sides of the plot).

It is immediately obvious that this algorithm is WCET optimized, as the instruction-count measurements for the *ORIGINAL* and *JITTER* binaries do not show any jitter, whereas the cycle values still show some deviation due to optimizations inherent to current processors (see above), but are still reasonably close together. The *MUTANT* runs show some clearly distinct behavior – especially when looking at the cycle values. Most prominent is the final peak above 1200 cycles – roughly 50% slower than the slowest non-mutated run.

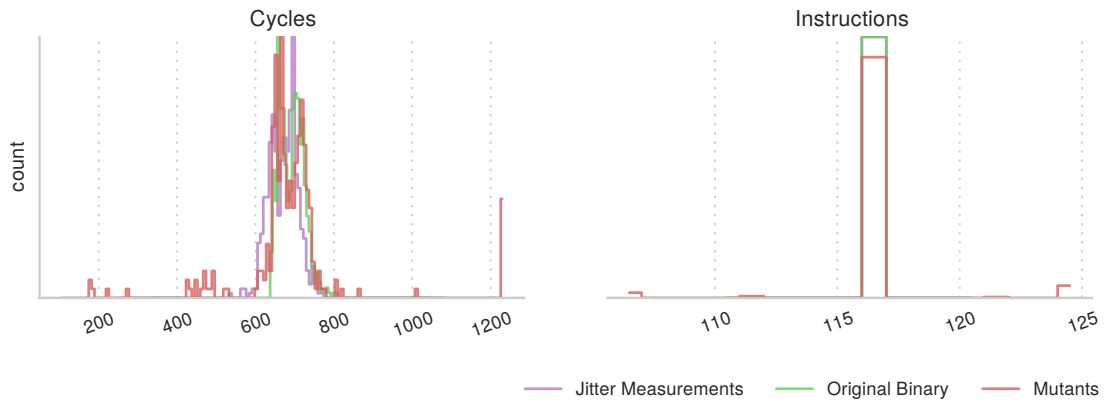

Figure 7.8: Timing distribution for *sqrt_micro_wcet* (Raspberry Pi 2)

The computed thresholds used for detection on gem5 and the Raspberry Pi 2 are shown in tables 7.2 and 7.3, the detailed timing distributions for all benchmarks are listed in appendix A.3. The threshold values clearly show that a) the cycle values have a higher spread than the instructions — this is due to load/store latencies, caching effects and variable-cycles-instructions, and b) the WCET optimized algorithms (*bubblesort_wcet*, *binsearch_wcet*, *prime_wcet*, *sqrt_micro_wcet*) have a much lower spread than their non optimized counterparts. (as expected)

	$Cycles_{min}$	$Cycles_{max}$	$Instructions_{min}$	$Instructions_{max}$
nop	0	276	0	26
long_nop	54	321	0	0
bubblesort	10255	5015489	7974	4002968
bubblesort_wcet	4515404	4515775	3502482	3502507
heapsort	251847	283675	178646	200086
quicksort	155915	1307556	103695	1053022
selectionsort	4018689	4019263	4007985	4008045
shellsort	117866	273377	112603	177513
binsearch	0	1283	182	299
binsearch_wcet	862	2948	341	361
duff	4646	4993	4583	4586
strstr	1516	138007	906	103374
fibcall	37	1288	0	1122
prime	0	370723	49	301254
prime_wcet	17596123	17596475	14188505	14188522
sqrt_micro	188	607	0	168
sqrt_micro_wcet	1024	1162	94	119
sqrt_micro_wcet_neg	37	1026	0	135
matmul	78228	78313	74088	74108
matmul_float	94706	94889	82505	82508
aes	25149	25717	19571	19621
crc32	112537	112732	110567	110649
fft	173955	178633	87646	89120
fir	217909	218084	159111	159146

Table 7.2: Computed timing bounds – gem5

	$Cycles_{min}$	$Cycles_{max}$	$Instructions_{min}$	$Instructions_{max}$
nop	0	191	4	4
long_nop	580	921	103	103
bubblesort	7717	3762326	8008	4003004
bubblesort_wcet	3510792	3511609	3502504	3502508
heapsort	184419	206490	178668	200108
quicksort	104312	1063829	103646	1052975
selectionsort	3013509	3366114	4008011	4008015
shellsort	86507	190508	112554	177464
binsearch	921	1898	133	250
binsearch_wcet	1182	1993	398	398
duff	4743	5371	4620	4621
strstr	1479	105646	943	103411
fibcall	107	1260	21	1119
prime	43	601784	15	301222
prime_wcet	11699716	11732111	14188557	14188566
sqrt_micro	239	766	18	202
sqrt_micro_wcet	533	805	116	116
sqrt_micro_wcet_neg	106	1001	17	119
matmul	74148	86884	74145	74147
matmul_float	114104	114870	82542	82544
aes	24515	25498	19605	19606
crc32	76025	76684	110615	110617
fft	191958	197417	87680	87682
fir	185828	187157	159145	159147

Table 7.3: Computed timing bounds – Raspberry Pi 2

7.4 Detection Results

The detection results (created according to section 7.3) are shown in figure 7.9 and table A.3 for gem5 and figure 7.10 and table A.4 for the Raspberry Pi 2.

Runtime denotes the detected mutants, i.e. the instances showing abnormal timing behavior and violating the bounds established in section 7.3

Timeout denotes the non-terminating mutants, denoted as *TIMEOUT* in section 7.1.

Crash denotes the crashing mutants, denoted as *CRASH* in section 7.1.

Undetected (Ok) denotes the undetected mutants still producing a correct result, part of the *OK* class in section 7.1.

Undetected (Error) denotes the undetected mutants producing an erroneous result, part of the *ERROR* class in section 7.1.

To focus on the improvements made possible by the method proposed in this thesis figure 7.11 and table A.5 (for gem5) and figure 7.12 and table A.6 (for the Raspberry Pi 2) exclude crashing benchmarks (denoted as *crash* detections above). Under the premise that a crash is trivial to detect anyways, these results focus on the actual improvements.

The detailed detection results (tables A.3 and A.4) also contain a detailed break down of the *Runtime* detections:

Both Criteria denotes the percentages of both – the cycle- and instruction count-–thresholds are violated.

Cycles only denotes the cases in which only the cycle threshold was abnormal.

Instructions only denotes the cases in which only the instruction-count threshold was abnormal.

It can be seen that combining the criteria slightly improves the detection rate for most of the benchmarks. It can also be seen that neither cycles nor instruction-count can be called the *better* measure for establishing the thresholds – it depends on the algorithm which of the two yields better results.

Both, *nop* and *long_nop* show unexpected behavior when running in gem5 – the *JITTER* and the *ORIGINAL* benchmark runs differ by a large factor. This seems to be a simulation artifact – gem5 *ignores* `NOP` instructions – so this two benchmarks' results should be ignored in that case.

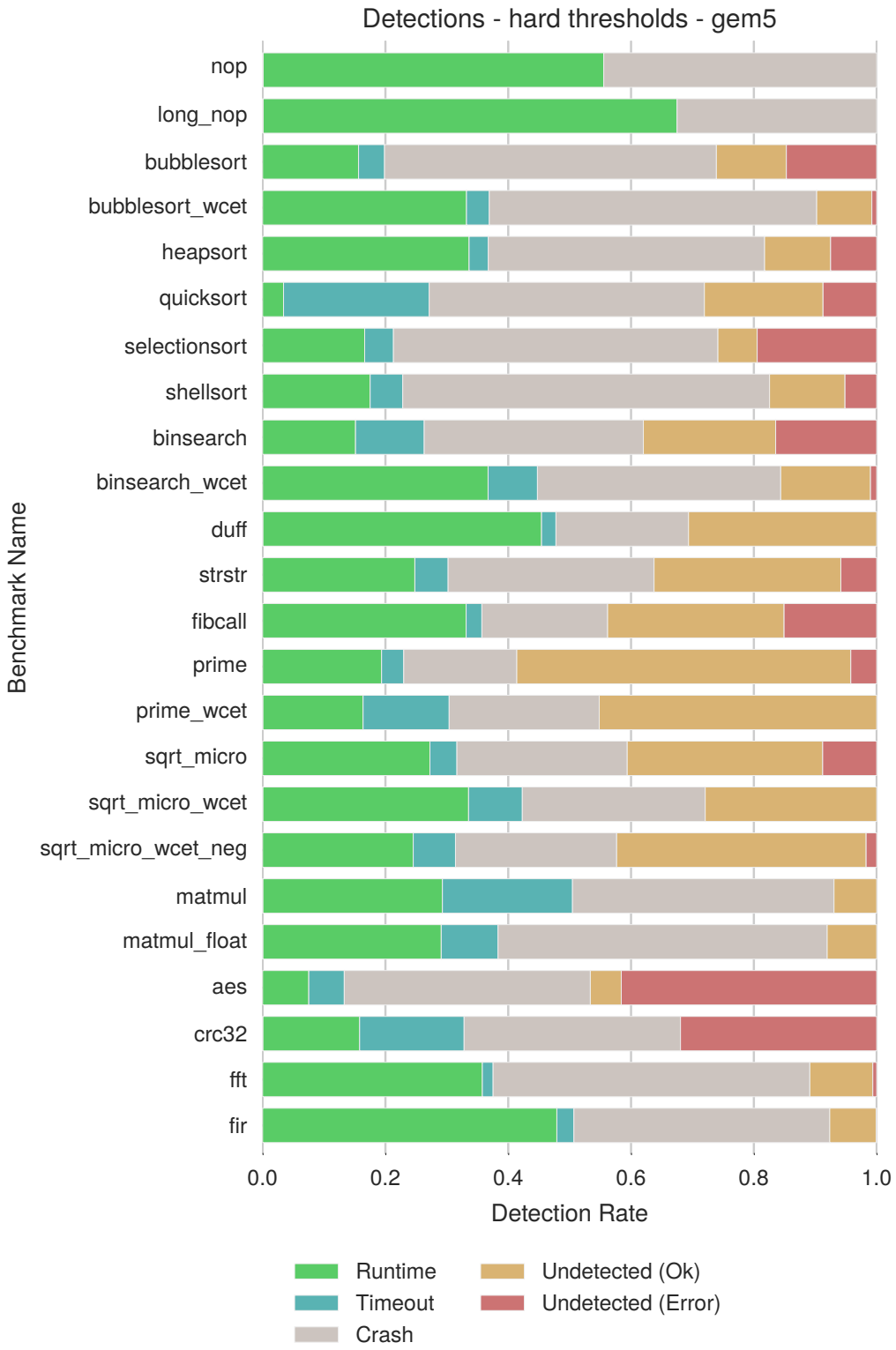

Figure 7.9: Detection results — gem5

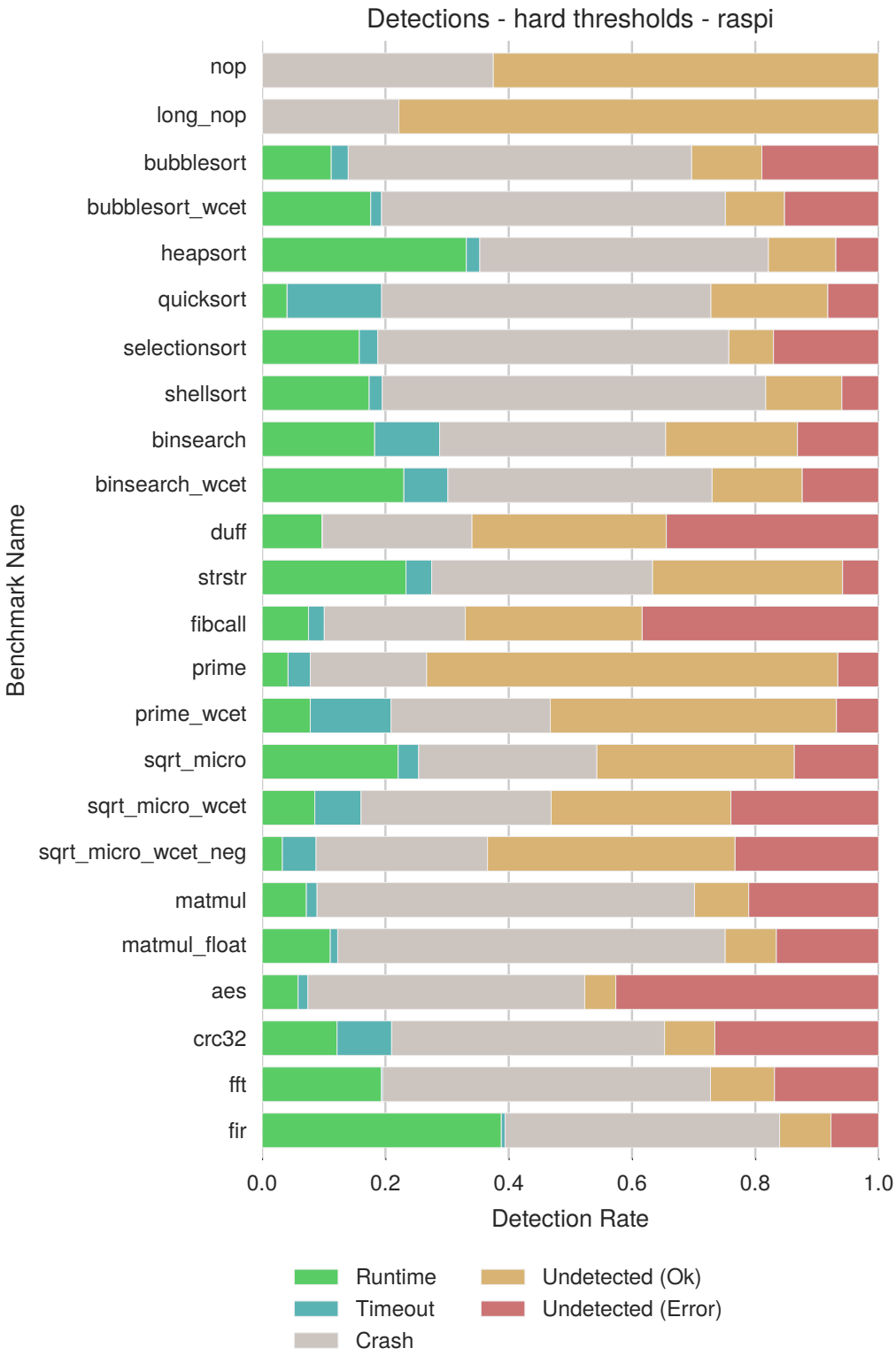

Figure 7.10: Detection results — Raspberry Pi 2

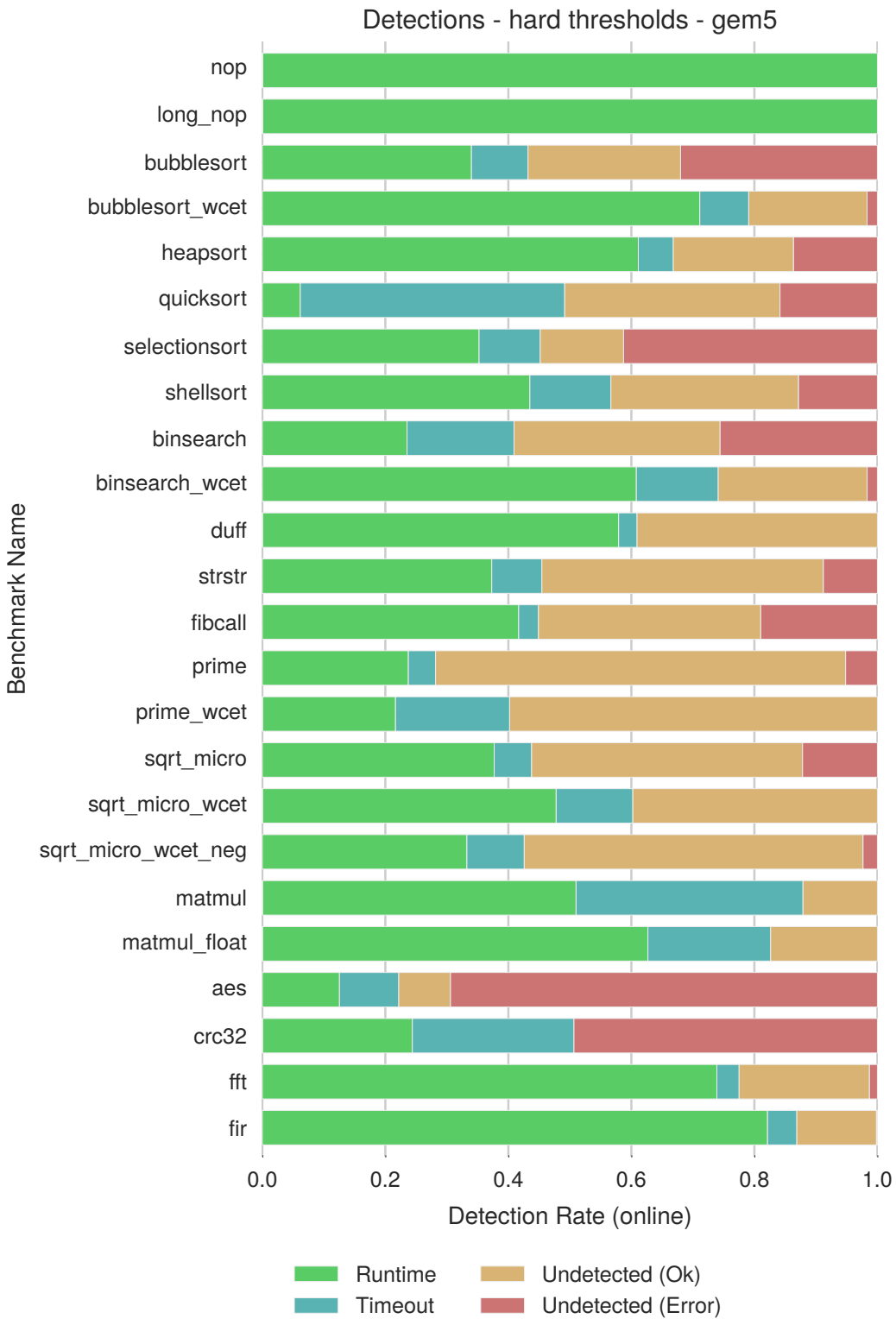

Figure 7.11: Detection results excluding crashes— gem5

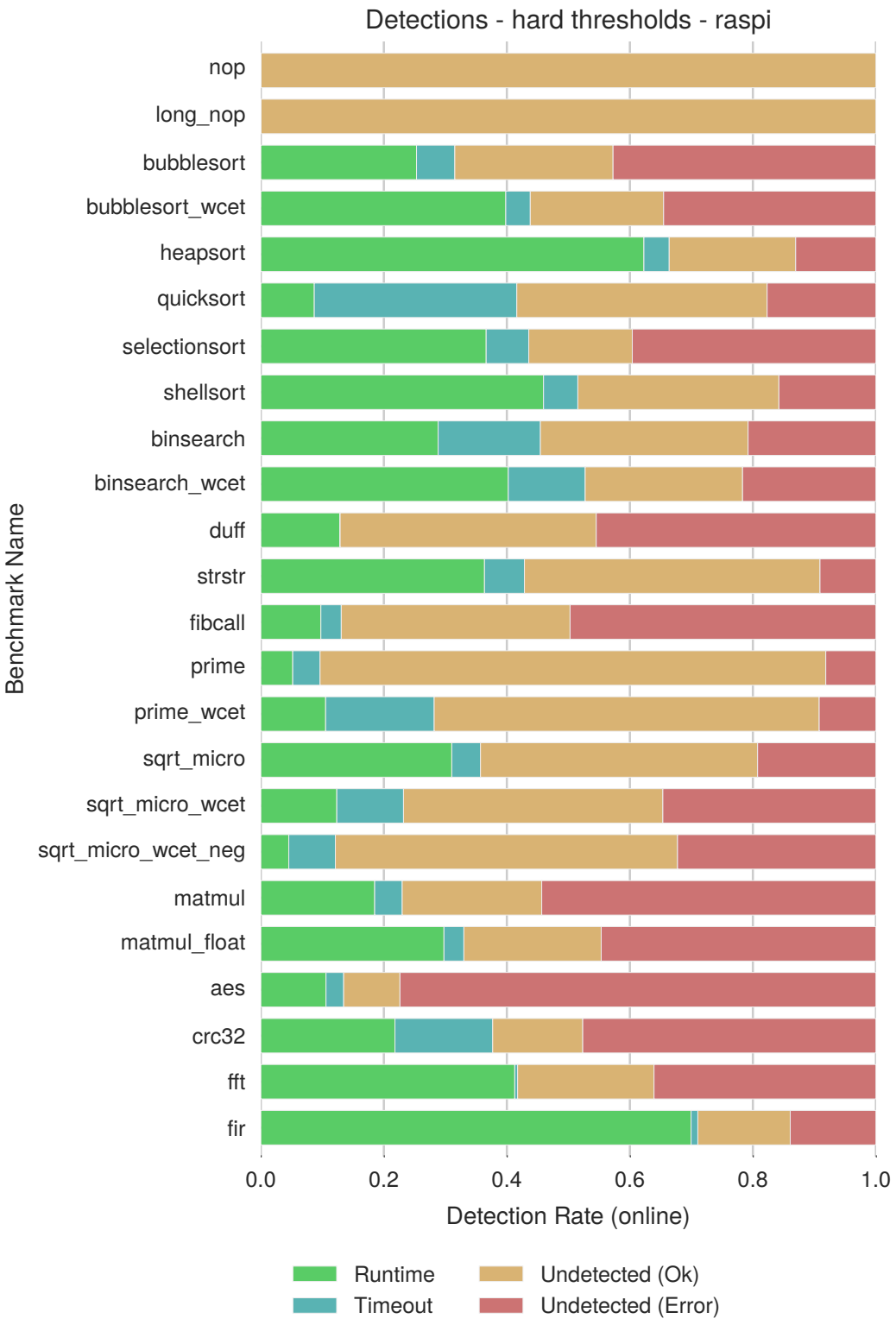

Figure 7.12: Detection results excluding crashes— Raspberry Pi 2

7.5 Lessons Learned

As visible in figure 7.13 the detection rates made possible by the work presented in this thesis are heavily dependent on the algorithm monitored – ranging from less than 10% (*prime*) to over 60% (*heapsort*, *fir*). The cause of those discrepancies becomes quite obvious when looking at the timing distributions and the derived detection thresholds (tables 7.2 and 7.3). *Heapsort* (figure A.53) and *fir* (figure A.72) show a low spread for both, cycles and instruction count, whereas — as expected — *prime* (figure A.62) exhibits an extremely input-data dependent runtime behavior.

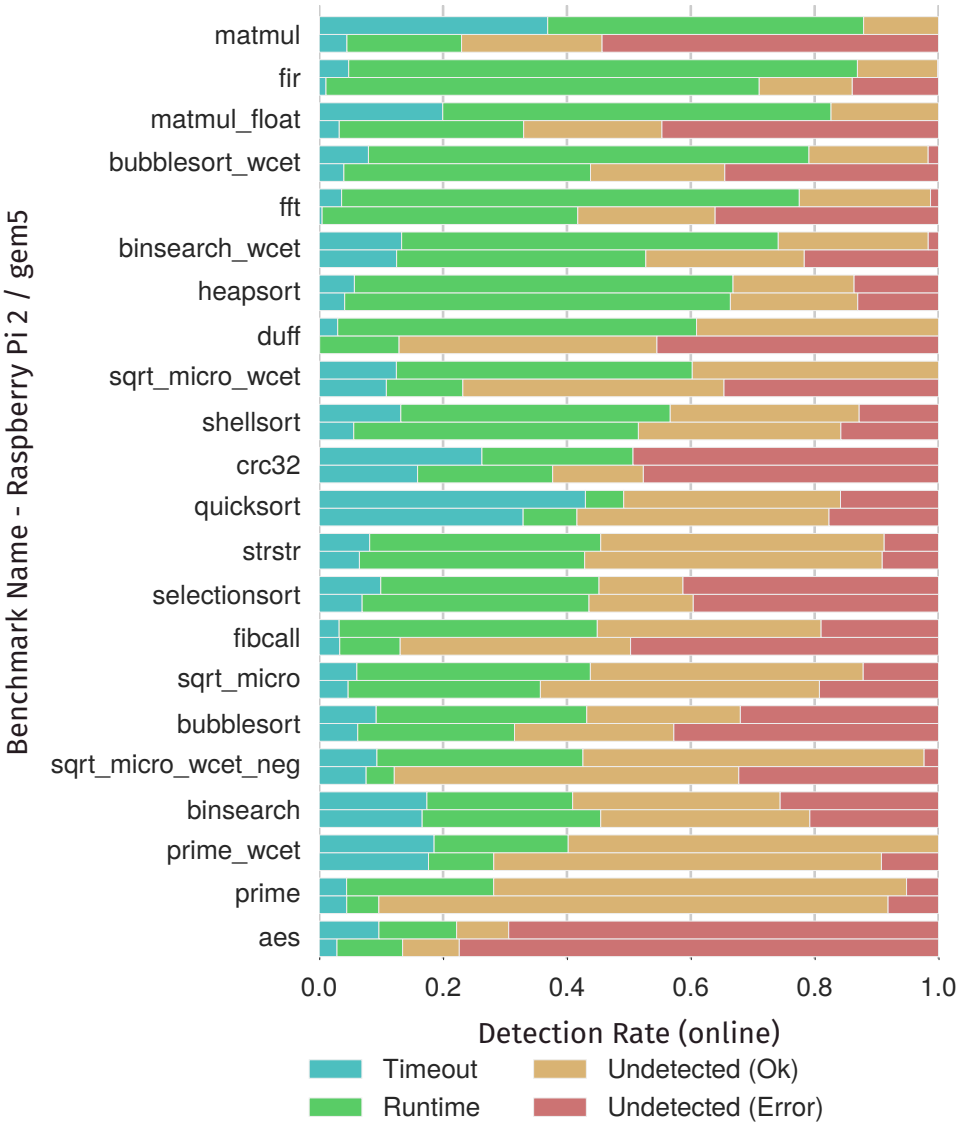

Figure 7.13: Detection results excluding crashes— gem5 (top) vs Raspberry Pi 2 (bottom)

Figure 7.13 also shows the impact of *timing indeterminism* (c.p. section 3.2) in the Raspberry Pi 2 – gem5's detection rate is up to four times higher (*matmul*) – this effect can be explained with caching effects, variable memory latency and variable-cycle-instructions. It also highlights the difference between what is theoretically and practically possible.

Of the sorting algorithms used *quicksort* (figures A.30 and A.54) exhibited one of the worst detection rates. This behavior becomes obvious when looking at the detection thresholds used (tables 7.2 and 7.3), where the $O(n^2)$ worst case behavior of the algorithm is obvious. Also, *quicksort* has the highest percentage of *TIMEOUT*s (infinite loops) of all tested algorithms, probably due to its fragile recursive structure and multiple loops. The WCET optimized bubblesort version *bubblesort_wcet* also shows notably higher detection rates than the unoptimized counterpart, and while it is slower on average, its worst case behavior is better, too. *AES* (figure A.69 and listing B.25) also exhibits a low detection rate. This was not unexpected, as there is no *data dependent branching* – all loops perform a constant number of iterations. The timing results also show a constant number of instructions executed, hinting that the algorithm is optimized for constant runtime.

In general, non-WCET optimized algorithms (with a wide spread for cycles or instruction count) tend to exhibit a lower detection rate (e.g. *bubblesort, prime*). However the converse is not true – while some of the optimized benchmarks (*bubblesort_wcet, heapsort, matmul, fir,...*) show high detection rates – others (*aes, sqrt_micro_wcet*) exhibit average to low detection rates only. Also fixed-iteration loops (*fir,matmul,...*) seem to improve detection rates – presumably due to the amplification provided by the loop.

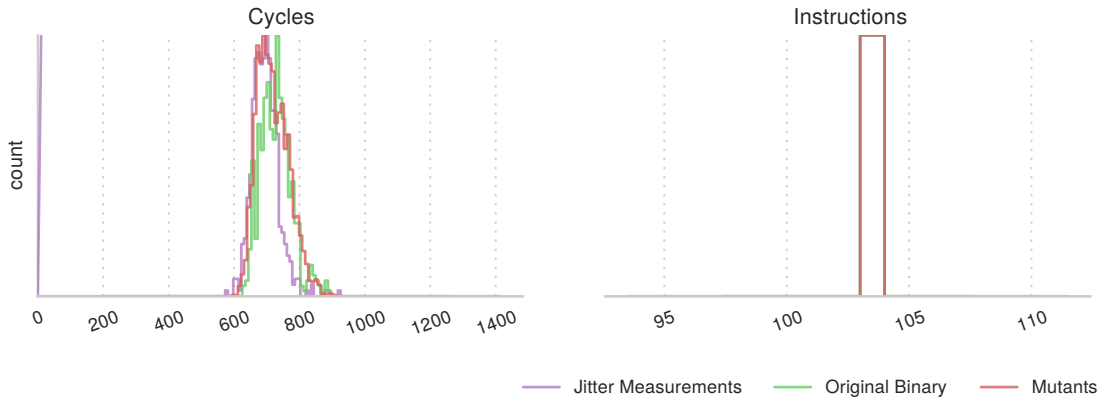

Figure 7.14: Timing distribution (Raspberry Pi 2) – *long_nop*

When looking at the measurement distributions (figures A.65 and A.69) we see just a few mutants deviating from the *normal* behavior. The reason for this behavior seems to be the low percentage of branches/loops in this algorithms – those being the most sensitive instructions.

Figure 7.14 shows the extreme case – the *long_nop* benchmark, containing no branches/loops at all. Here no significant difference between the mutated and the original binaries can be seen.

As noted in section 7.4 monitoring both – cycles and instruction count – is beneficial, as depending on the algorithm, one or the other yields a better detection rate, and in most cases combined monitoring of both values yields an improved result.

It is also noteworthy that WCET optimized algorithms are not necessarily better suited for the proposed detection approach. While the detection rate for the pairs *binsearch/binsearch/_wcet* and *prime/prime_wcet* shows distinct improvements — the converse is true for *sqrt_micro/sqrt_micro_wcet*. For the non-optimized version the cycle distribution of the mutated versions (figure A.64) shows a distinct second peak – so, even as the spread of the detection thresholds is wider, the injected faults have a much more pronounced effect on the timing behavior – facilitating detection. This probably occurs because the WCET optimized algorithm is much simpler in structure and executes fewer data dependent code (cp. listing B.20 and listing B.21) – especially the number of loop iterations is not data dependent in the optimized version. This behavior is even more obvious when comparing the disassembled binaries (listing B.29 and listing B.30).

Experimenting with gem5 (Sections 5.2 and 7.2) showed that, while it is a reliable simulator for the ARM architecture, extensive fine tuning is needed to closely mimic the behavior of a given SoC/system-board. On the other hand, its open architecture and extensible API allow the consistent simulation and analysis of small differences in CPU design – potentially providing helpful insights for choosing suitable processors for real-time workloads.

CHAPTER 8

Conclusion and Future Work

8.1 Conclusion

Concludingly, this thesis shows that the proposed approach of measuring execution-time characteristics is feasible for the detection of runtime errors and that it is possible to implement fine-grained execution-time monitoring. The method shows promising results, reliably detecting up to 70% of non-crash errors when monitoring both – cycles and instruction counts.

The achieved detection rate varies between different algorithms, with WCET-optimized algorithms yielding – on average – higher detection rates than non-optimized ones. Algorithms containing a high percentage of branches/loops are more sensitive to bit flips, and thereby show higher detection rates, too – so some WCET optimizations can actually reduce the achievable detection rate.

The implementation effort needed to add this monitoring approach to an existing algorithm is reasonably low — but a fair amount of time has to be spent on analyzing and identifying safe bounds for the monitored algorithms.

The comparison of the Raspberry Pi 2 single-board computer to the gem5 simulator (section 7.2) shows that further improvements of the detection rate are possible on platforms with temporally predictable instructions/ memory access – an, on average, 20% improved detection rate was achieved on gem5.

With possible future tooling support the whole process can be (semi-)automated by a build system – automatically generating monitoring blocks for critical algorithms.

To further increase the detection rate some future work is still needed (c.p. section 3.2) – extending the proposed method and the analysis tools to cover complex distributions of timing measurements. (c.p. section 8.2)

The tools and libraries needed to implement the proposed method are openly available, easy to use and integrate into existing systems running Linux – provided the SoC used provides a hardware PMU and support for the `perf_event` framework is enabled.

8.2 Future Work

This thesis shows promising results for detecting runtime errors – it also hints at possible future work, improvements, and possible research:

- The next logical step is the extension of the work done to further processors (other ARM cores) and architectures (e.g. x64).
- The CPU-model used in gem5 needs some tweaking to yield more accurate results. Especially low-level modeling of the cache- and memory subsystems could drastically improve the simulation.
- An actual online integration of the `perf_event` framework can be implemented. The framework already provides the functionality for emitting a signal when a certain threshold is crossed, so an online framework needs to implement functionality to wrap a task in the monitoring environment, perform post-task evaluation (for the detection of lower bound violations), and provide a signal handler with appropriate fault management code for handling upper bound violations.
- Depending on the actual hardware support, `perf_event` exposes more performance events than just instruction- and cycle-counters. Those counters could, for example, be used to dynamically adjust the cycle bounds, based on the actual cache- and branch-misses.
- While this thesis uses a dynamic measurement approach with normal distribution estimation, an appropriate static BCET/WCET analysis tool (cp. [KP10]) can be used to (automatically) analyze the algorithms and determine the detection thresholds. This step (and the previous ones) can even be integrated into a compile environment to (semi-)automatically generate monitored tasks.
- The detection models used for determination of the *good/bad* timing values can be improved. While in this thesis used a simple threshold based model (section 7.3), the gathered results already show that improvements in the detection rate are possible when using models more specifically tailored towards the monitored algorithms. (c.p. section 3.2) This is evident when comparing *sqrt_micro_wcet/sqrt_micro_wcet_neg* (cp. listings B.21 and B.22). Those algorithms are identical, except for the addition of a check for negative inputs, resulting in an early return in *sqrt_micro_wcet_neg*. This change results in two distinct peaks in the timing distribution (figure A.66, and

therefore in a very poorly fitting model (cp. table 7.3). In this case, a model generating two distinct *good* ranges (one for each peak) can easily improve the detection rate by a factor of two to three.

- Using input-data dependent detection models is another possibility for future improvements. When looking at the timing distributions for the *fibcall* (figure A.61) and *prime* (figure A.62) benchmarks, we can see that the input data heavily affects the timing behavior of the algorithm. Even a simple, parametrized model can drastically improve the detection rates in those cases.

APPENDIX A

Detailed Results

A.1 Comparison gem5 - Raspberry Pi 2

This section shows the detailed comparison results for the *MUTANT* runs between gem5 and the Raspberry Pi 2. (As explained in section 7.2)

Green marks denote *OK* results, red marks denote *ERROR*. (c.p. section 7.1)

binsearch – figure A.9– is a prime example for memory access jitter visible on the Raspberry Pi 2. (c.p. section 3.2) Looking at *sqrt_micro* – figure A.16 – we can see a similar behavior, but in this case it is caused by variable-cycle-instructions.

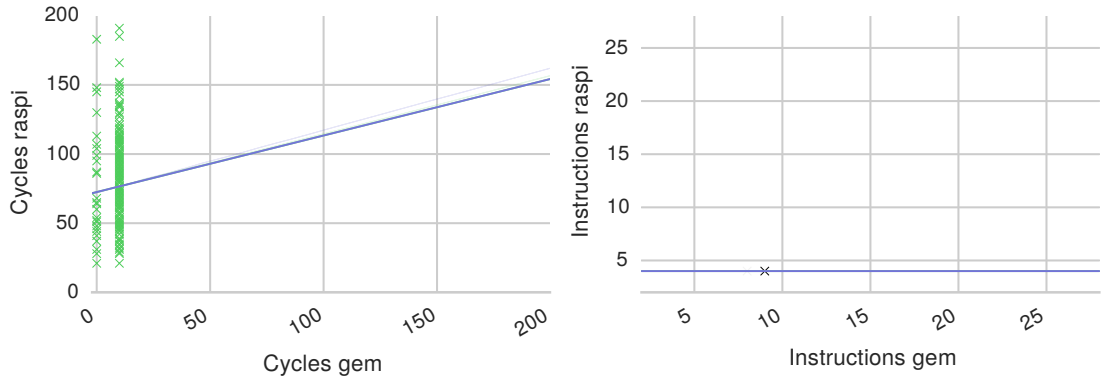

Figure A.1: Regression: gem5 vs raspi — *nop*

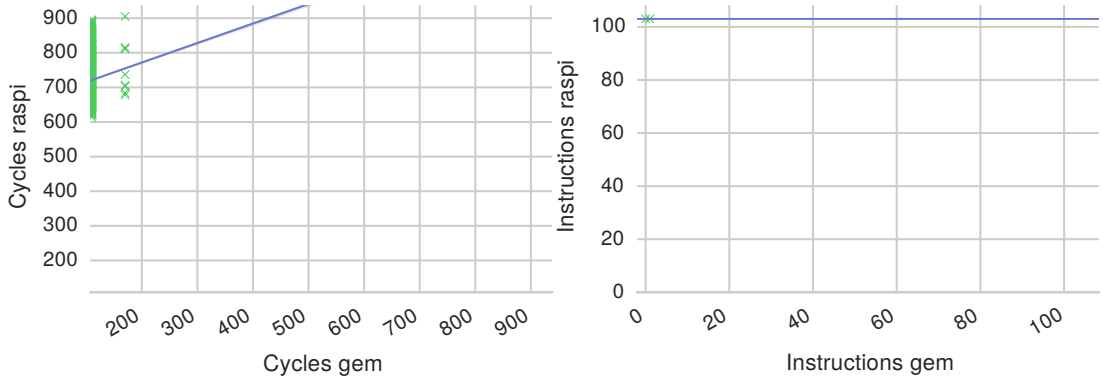

Figure A.2: Regression: gem5 vs raspi — *long_nop*

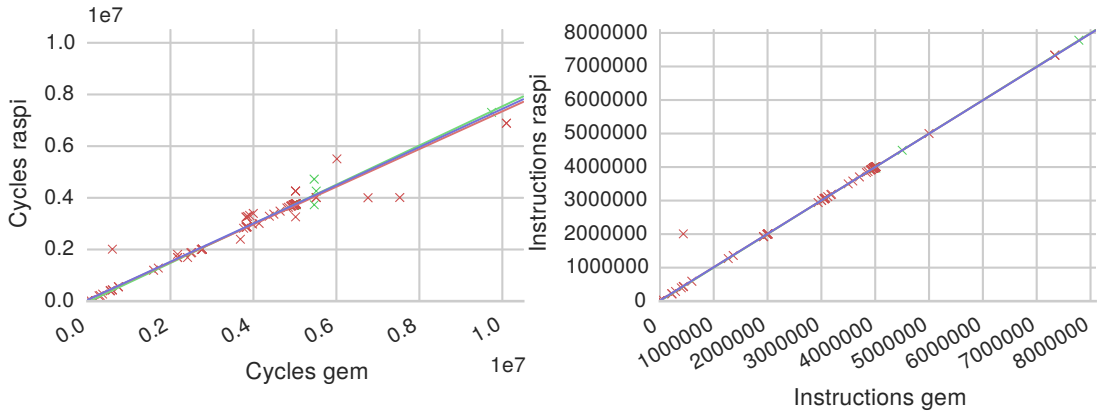

Figure A.3: Regression: gem5 vs raspi — *bubblesort*

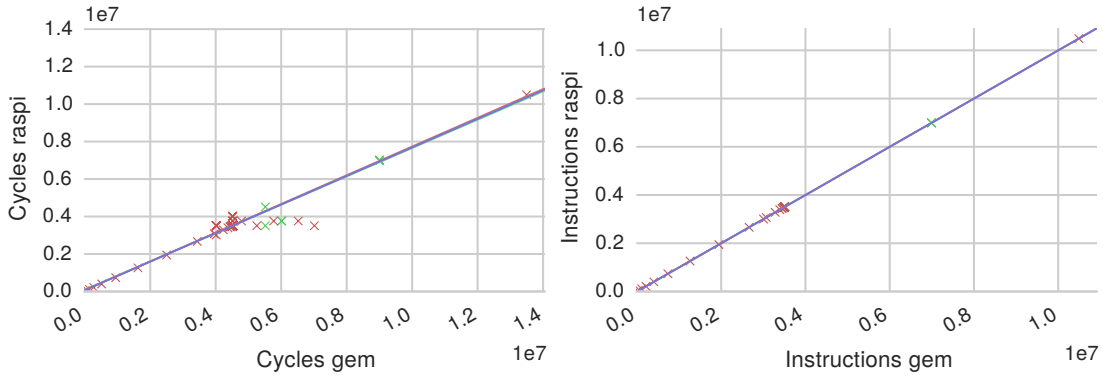

Figure A.4: Regression: gem5 vs raspi — *bubblesort_wcet*

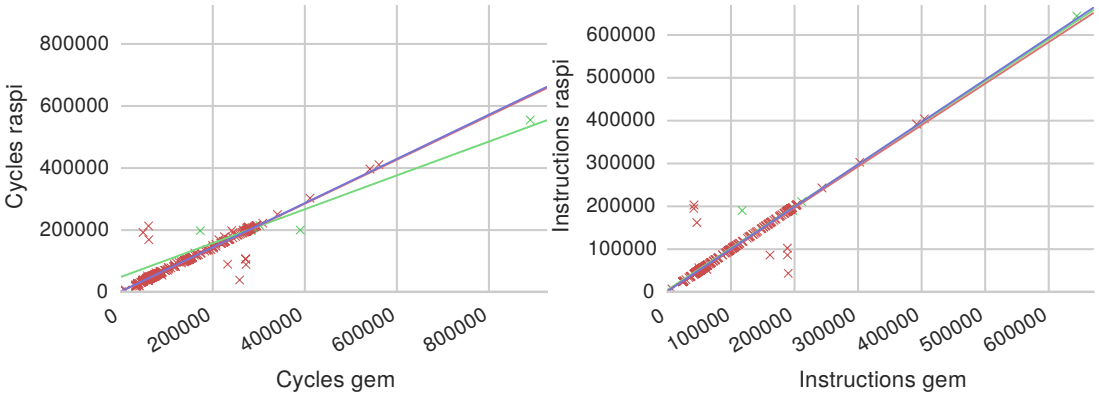

Figure A.5: Regression: gem5 vs raspi — *heapsort*

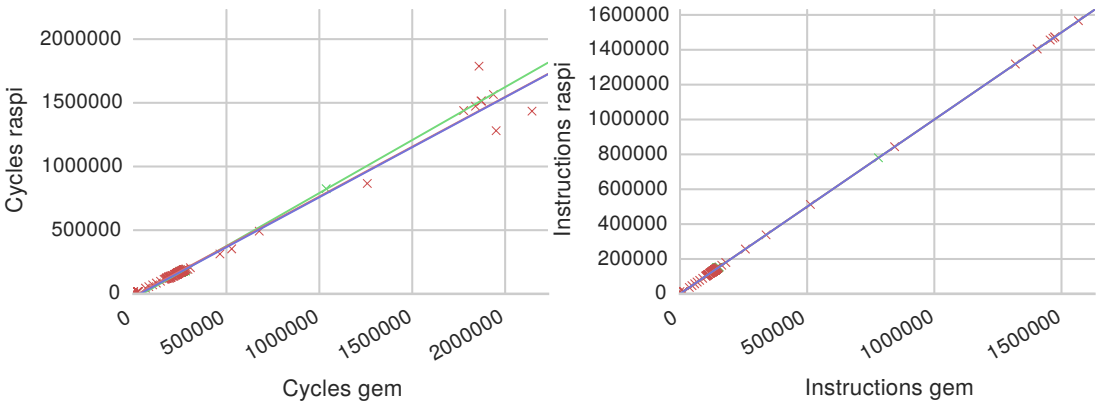

Figure A.6: Regression: gem5 vs raspi — *quicksort*

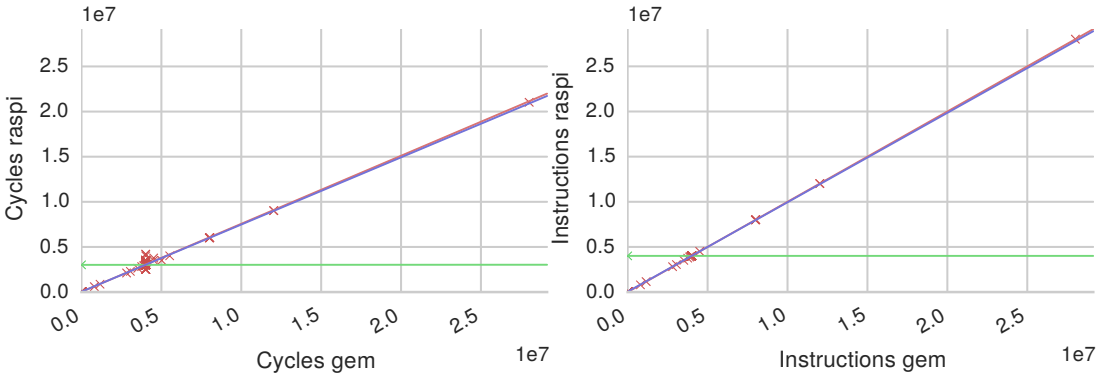

Figure A.7: Regression: gem5 vs raspi — *selectionsort*

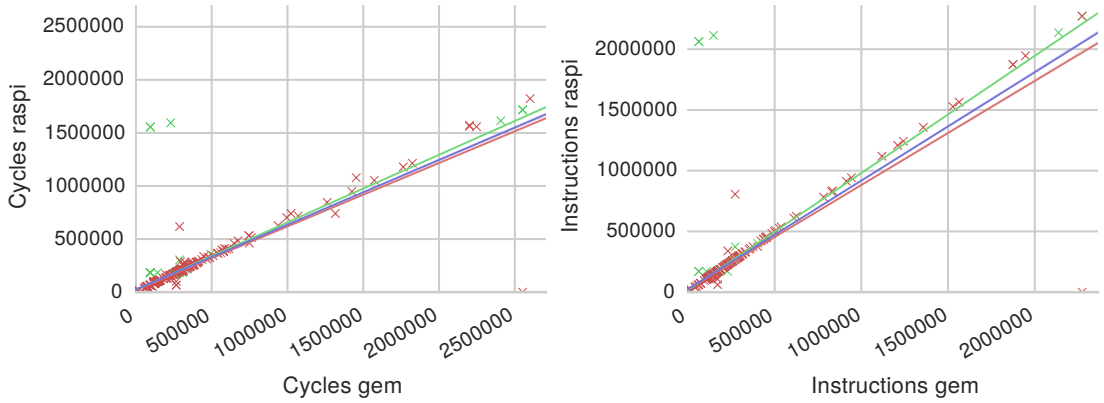

Figure A.8: Regression: gem5 vs raspi — *shellsort*

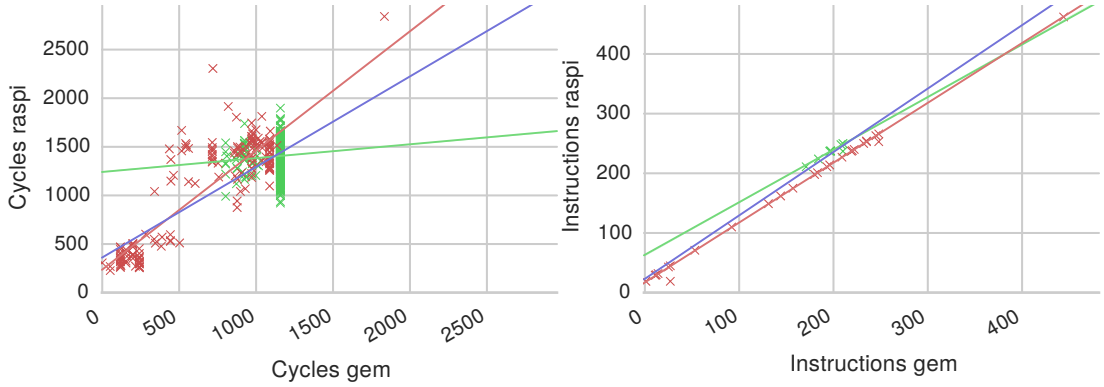

Figure A.9: Regression: gem5 vs raspi — *binsearch*

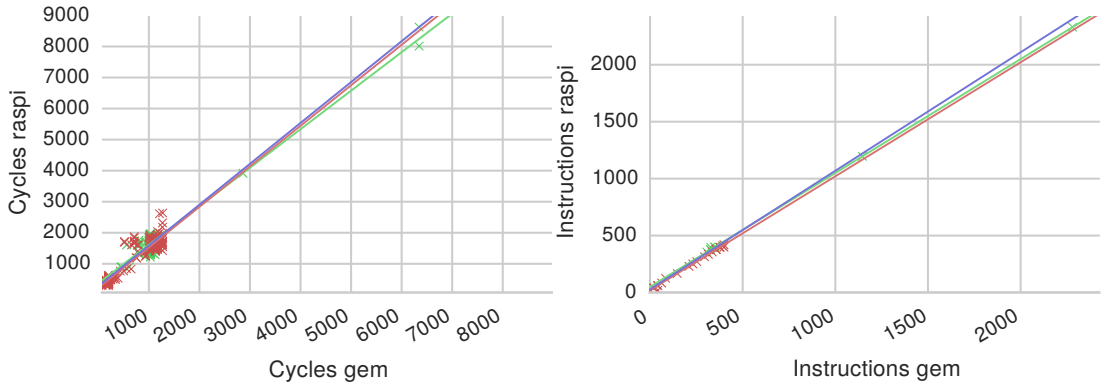

Figure A.10: Regression: gem5 vs raspi — *binsearch_wcet*

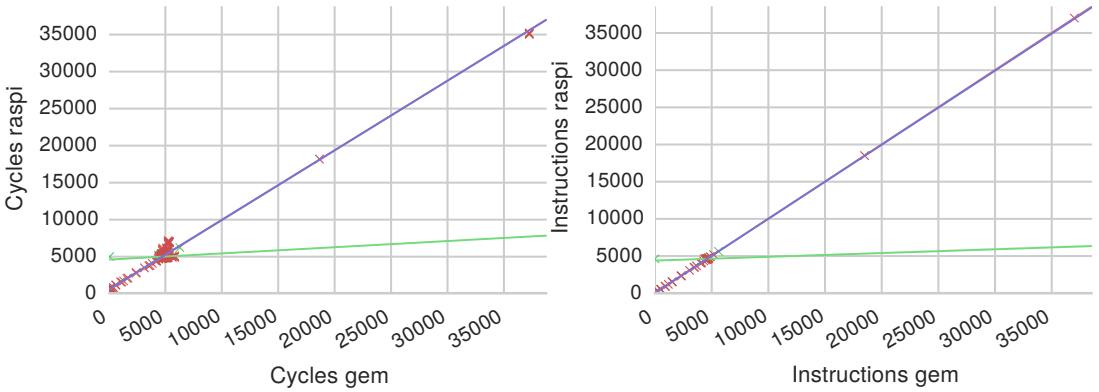

Figure A.11: Regression: gem5 vs raspi — *duff*

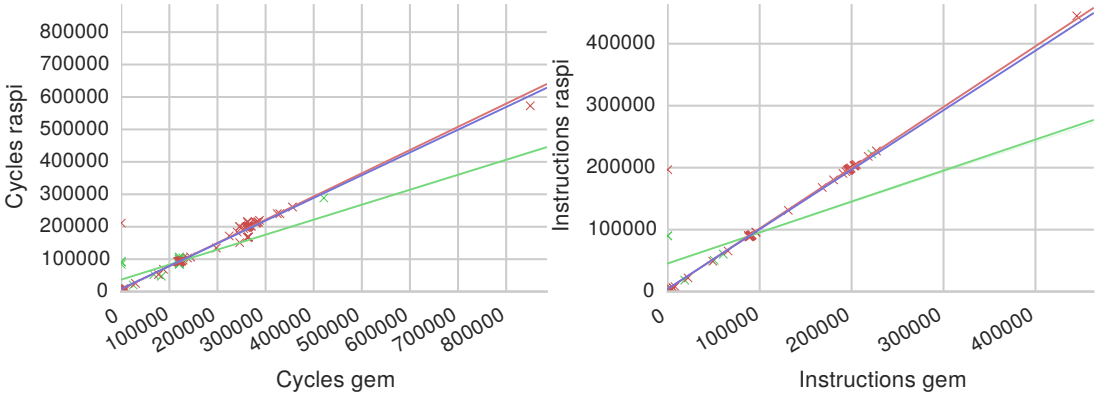

Figure A.12: Regression: gem5 vs raspi — *strstr*

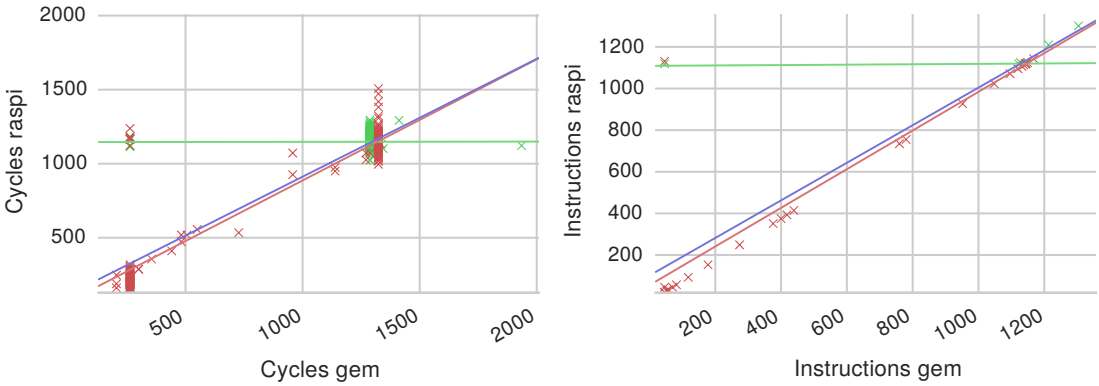

Figure A.13: Regression: gem5 vs raspi — *fibcall*

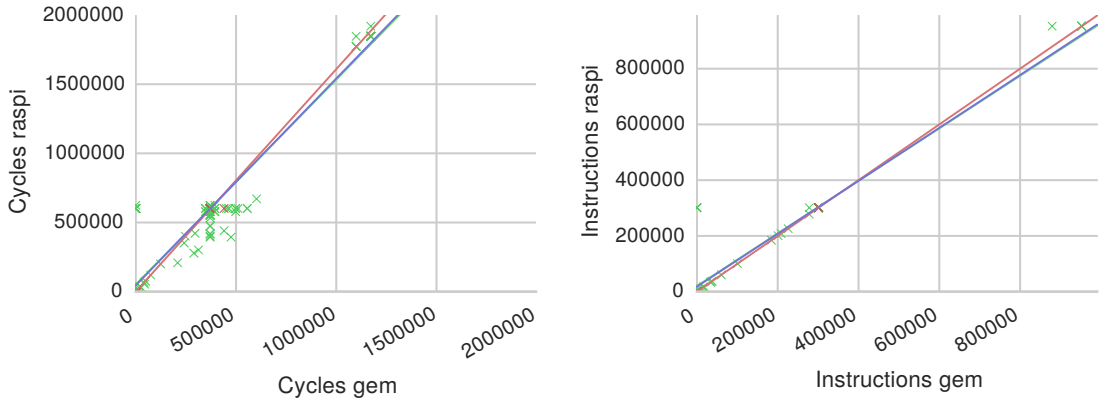

Figure A.14: Regression: gem5 vs raspi — *prime*

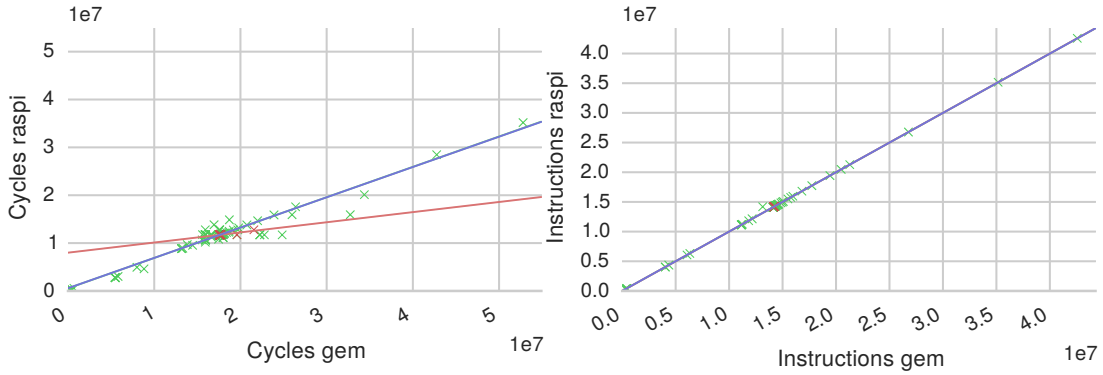

Figure A.15: Regression: gem5 vs raspi — *prime_wcet*

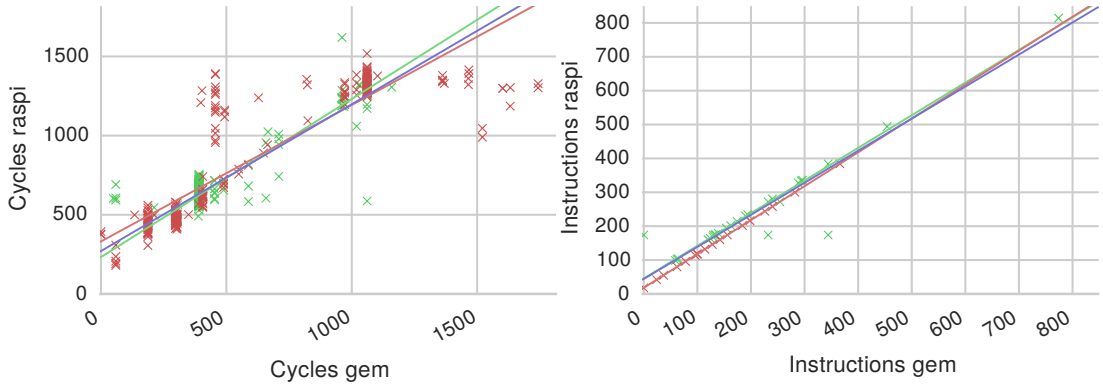

Figure A.16: Regression: gem5 vs raspi — *sqrt_micro*

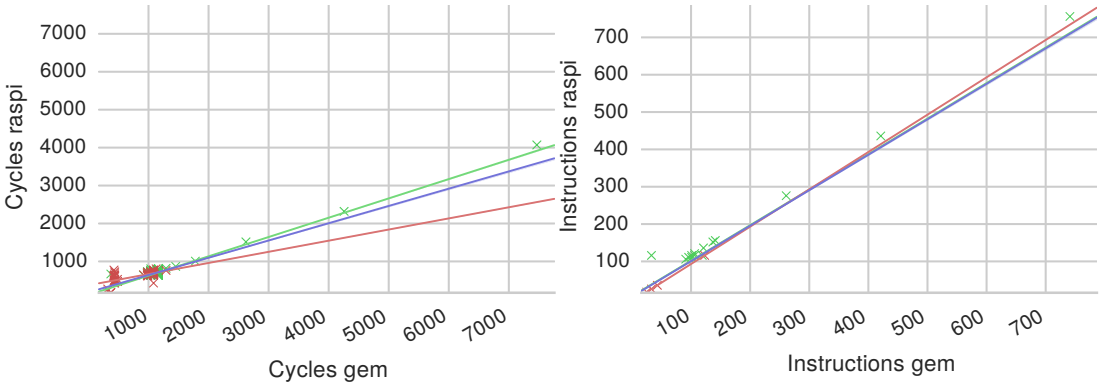

Figure A.17: Regression: gem5 vs raspi — *sqrt_micro_wcet*

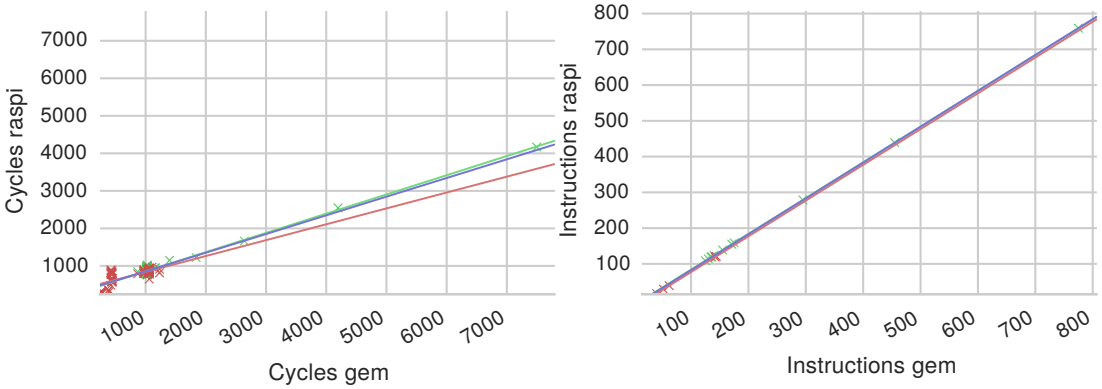

Figure A.18: Regression: gem5 vs raspi — *sqrt_micro_wcet_neg*

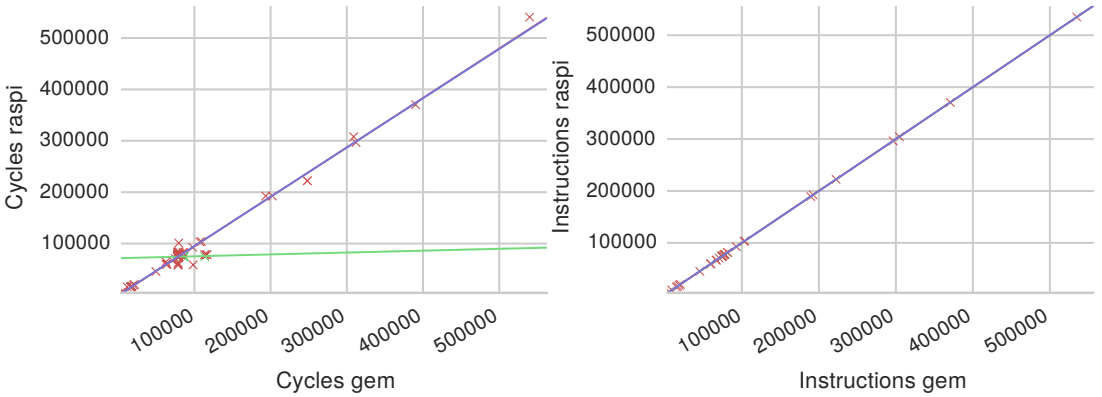

Figure A.19: Regression: gem5 vs raspi — *matmul*

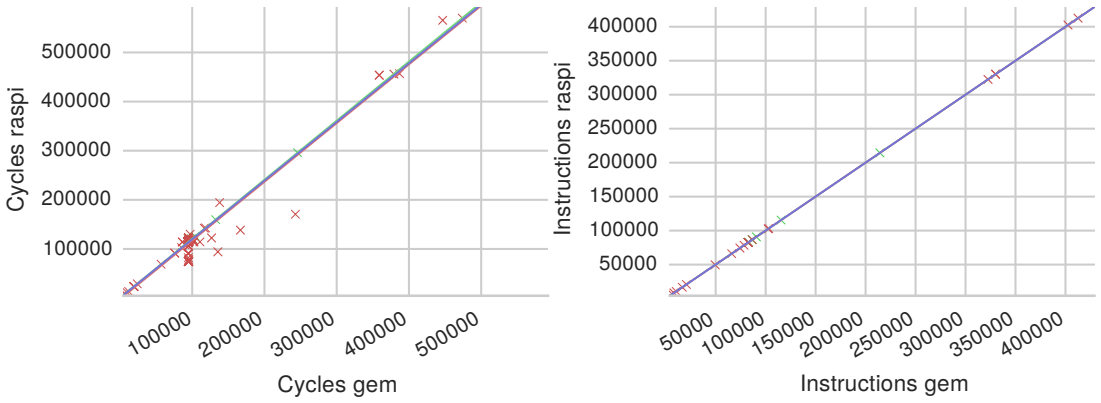

Figure A.20: Regression: gem5 vs raspi — *matmul_float*

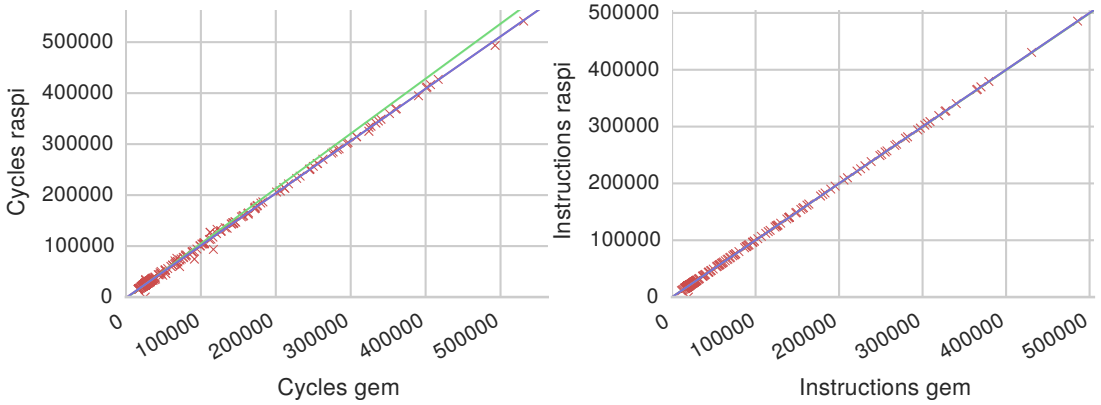

Figure A.21: Regression: gem5 vs raspi — *aes*

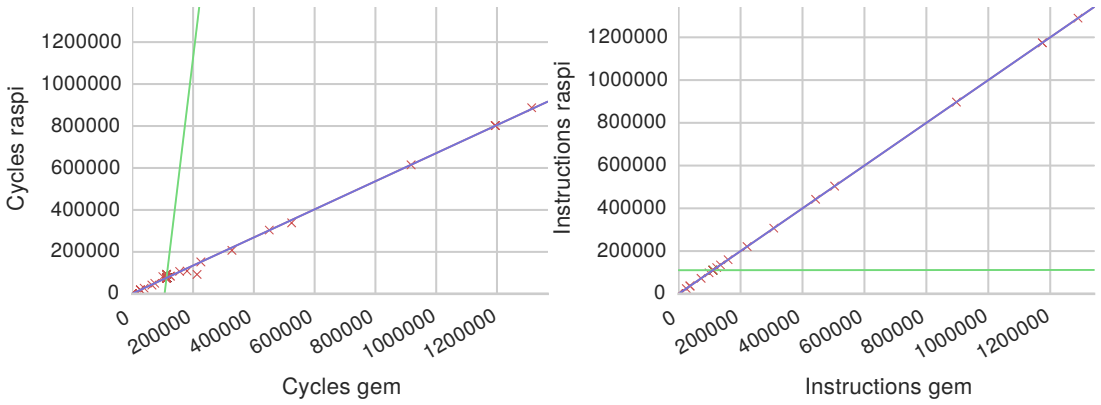

Figure A.22: Regression: gem5 vs raspi — *crc32*

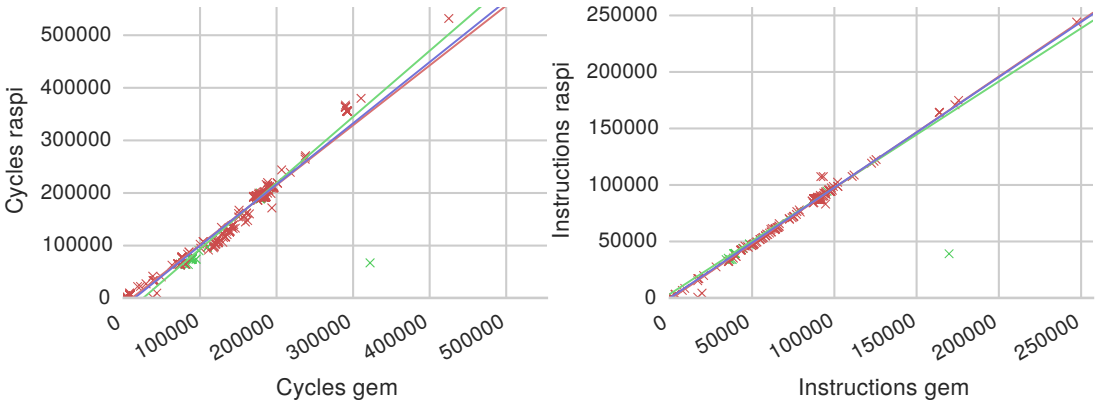

Figure A.23: Regression: gem5 vs raspi — *fft*

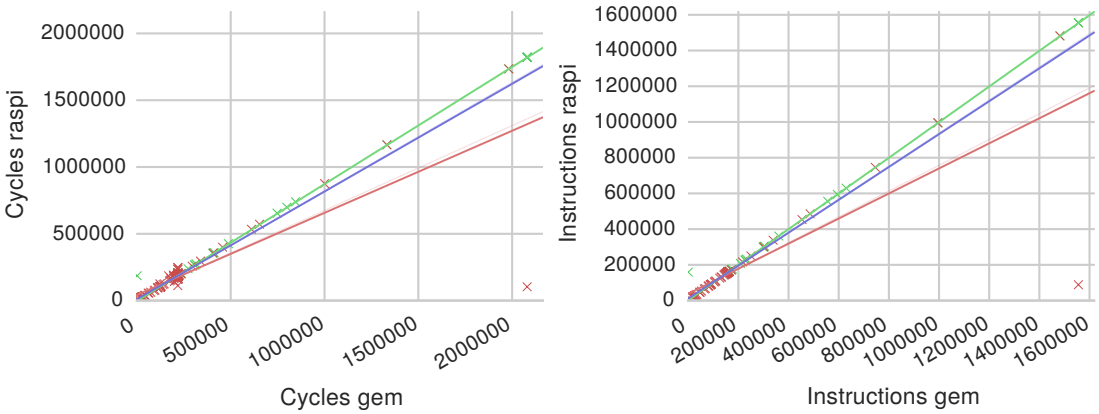

Figure A.24: Regression: gem5 vs raspi — *fir*

A.2 Injection Results

This section shows the detailed injection results for the *MUTANT* runs on gem5 and the Raspberry Pi 2. (c.p. section 7.1)

	OK	ERROR	TIMEOUT	CRASH
nop	55.6	0.0	0.0	44.4
long_nop	67.5	0.0	0.0	32.5
bubblesort	12.3	29.4	4.2	54.0
bubblesort_wcet	12.4	30.4	3.7	53.5
heapsort	11.2	40.7	3.1	45.0
quicksort	19.3	12.1	23.7	44.8
selectionsort	7.0	35.5	4.7	52.8
shellsort	14.2	20.7	5.3	59.8
binsearch	23.0	30.1	11.2	35.7
binsearch_wcet	19.6	32.7	8.0	39.6
duff	31.9	44.1	2.3	21.6
strstr	30.8	30.2	5.4	33.5
fibcall	34.1	42.8	2.6	20.5
prime	70.0	7.9	3.6	18.4
prime_wcet	54.2	7.4	14.0	24.4
sqrt_micro	40.2	27.6	4.4	27.7
sqrt_micro_wcet	31.0	30.4	8.8	29.8
sqrt_micro_wcet_neg	42.7	24.2	6.9	26.2
matmul	8.2	27.9	21.2	42.6
matmul_float	9.1	28.0	9.3	53.6
aes	5.4	48.7	5.8	40.1
crc32	8.2	39.6	17.0	35.2
fft	17.8	28.8	1.8	51.6
fir	10.8	44.8	2.8	41.7

Table A.1: classified execution results – gem5

	OK	ERROR	TIMEOUT	CRASH
nop	62.5	0.0	0.0	37.5
long_nop	77.8	0.0	0.0	22.2
bubblesort	12.3	29.2	2.8	55.7
bubblesort_wcet	12.5	29.8	1.8	55.9
heapsort	11.8	39.2	2.2	46.8
quicksort	19.1	12.1	15.3	53.4
selectionsort	7.2	32.8	3.0	57.0
shellsort	14.5	21.2	2.1	62.3
binsearch	22.7	30.1	10.5	36.7
binsearch_wcet	19.6	30.4	7.1	42.9
duff	32.0	43.7	0.1	24.3
strstr	31.2	28.7	4.2	35.8
fibcall	33.9	40.6	2.6	22.9
prime	70.7	6.9	3.6	18.9
prime_wcet	54.0	7.1	13.1	25.8
sqrt_micro	41.1	26.7	3.3	28.9
sqrt_micro_wcet	33.1	28.5	7.5	30.8
sqrt_micro_wcet_neg	43.1	23.6	5.5	27.8
matmul	9.3	27.7	1.7	61.3
matmul_float	9.6	26.3	1.2	62.9
aes	5.4	48.1	1.6	44.9
crc32	8.9	38.0	8.9	44.3
fft	16.9	29.6	0.2	53.2
fir	11.5	43.4	0.6	44.5

Table A.2: classified execution results – Raspberry Pi 2

A.3 Timing Distributions – Introduction

The detailed measurement results and the fitted distributions for each benchmark, as explained in section 7.3.

- *Jitter Measurements* (lilac) denotes the collected *JITTER* data, and the detection model based on it.
- *Original Binary* (green) different runs of the unmodified *ORIGINAL* binary.
- *Mutants* (red) measurements from the mutated binaries.

A.4 Timing Distributions – gem5

There is some notable differences between the *ORIGINAL*– and the *JITTER* measurements in gem5. (e.g. *nop*, *bubblesort_wcet*, *selectionsort*, . . .) The source of differences could not be pinpointed, it seems to be some variance within gem5 – the exact same binaries run perfectly fine on real hardware. (c.p. appendix A.5). The only visible impact is a slightly reduced detection rate for gem5, as the computed bounds are slightly less tight than possible.

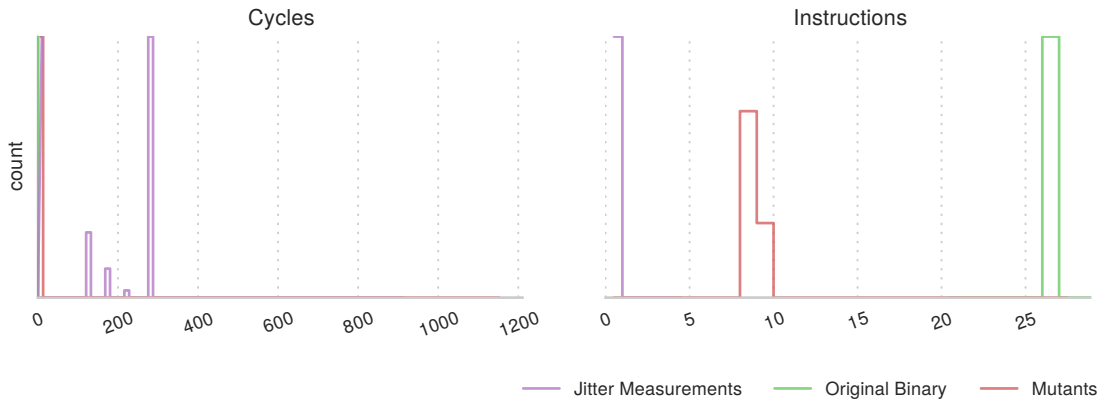

Figure A.25: Timing distribution (gem5) – *nop*

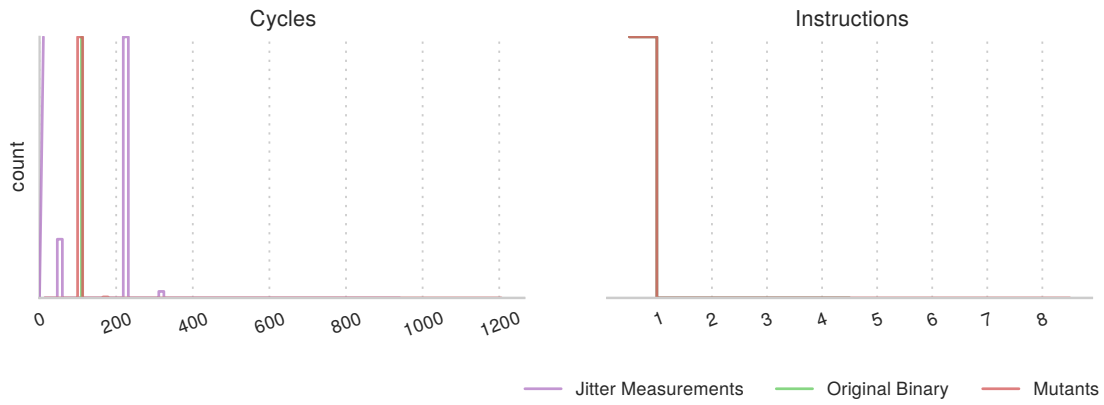

Figure A.26: Timing distribution (gem5) – *long_nop*

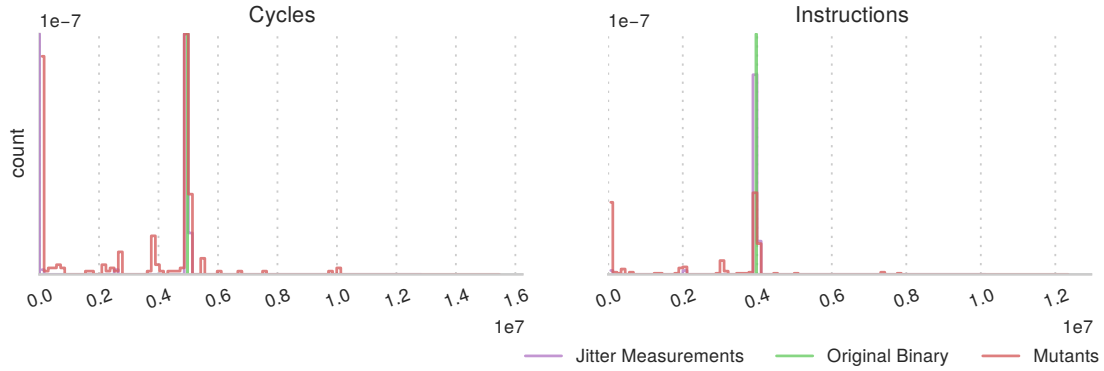

Figure A.27: Timing distribution (gem5) – *bubblesort*

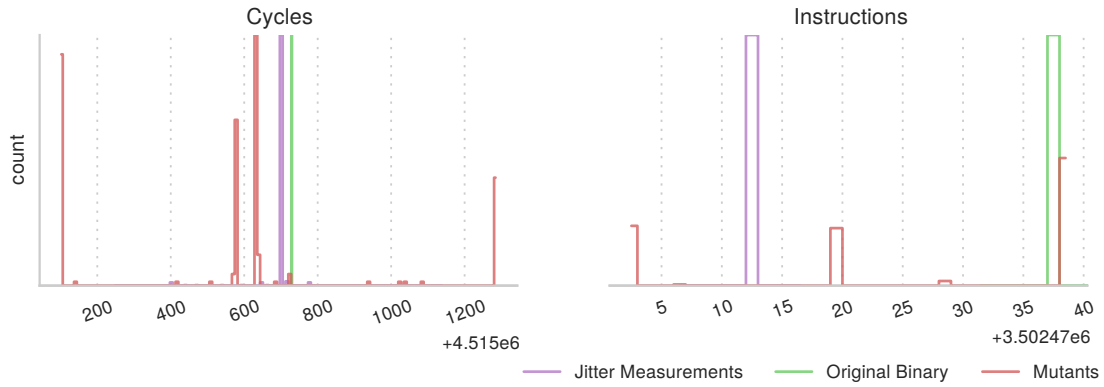

Figure A.28: Timing distribution (gem5) – *bubblesort_wcet*

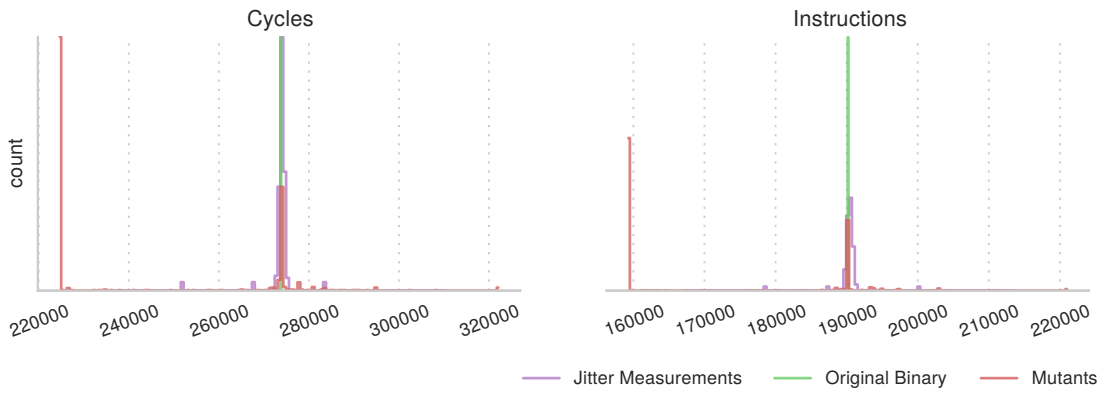

Figure A.29: Timing distribution (gem5) – *heapsort*

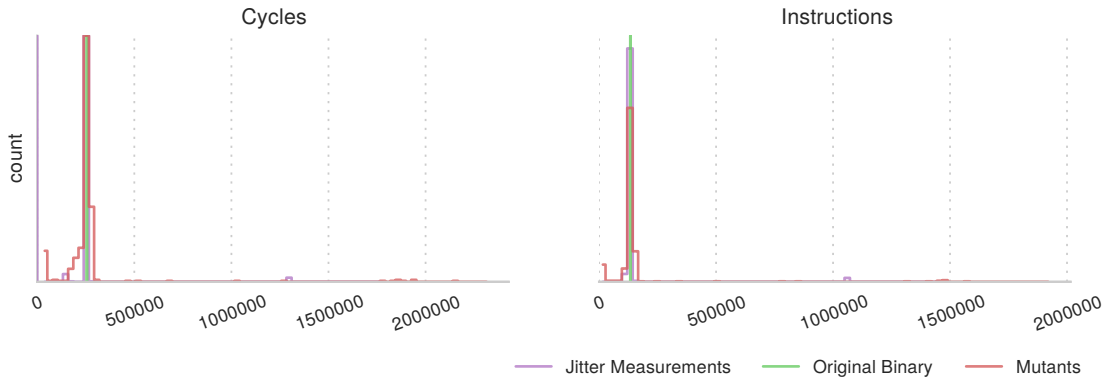

Figure A.30: Timing distribution (gem5) – *quicksort*

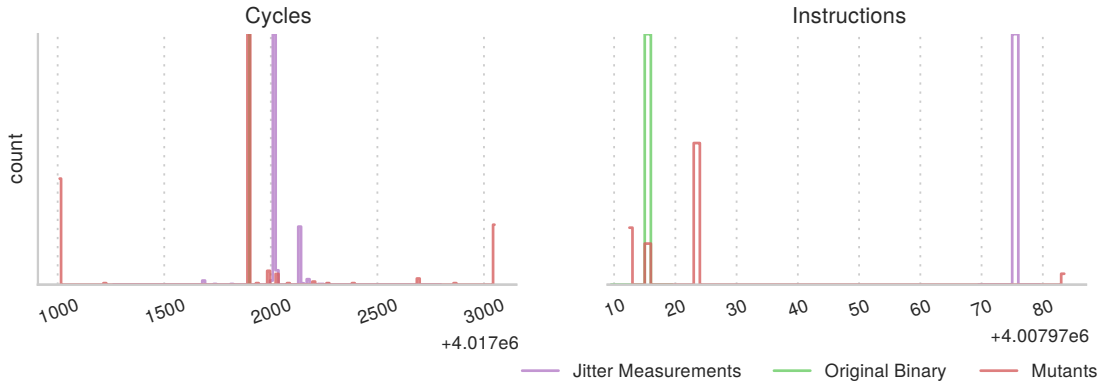

Figure A.31: Timing distribution (gem5) – *selectionsort*

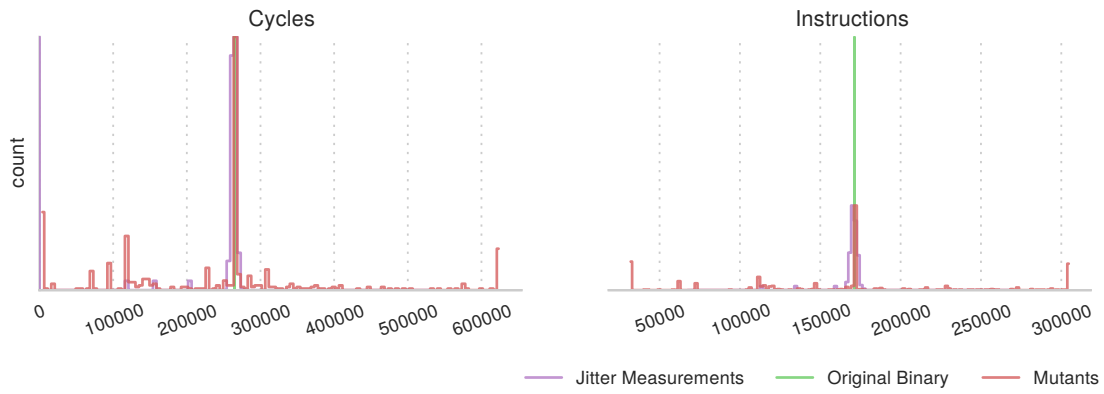

Figure A.32: Timing distribution (gem5) – *shellsort*

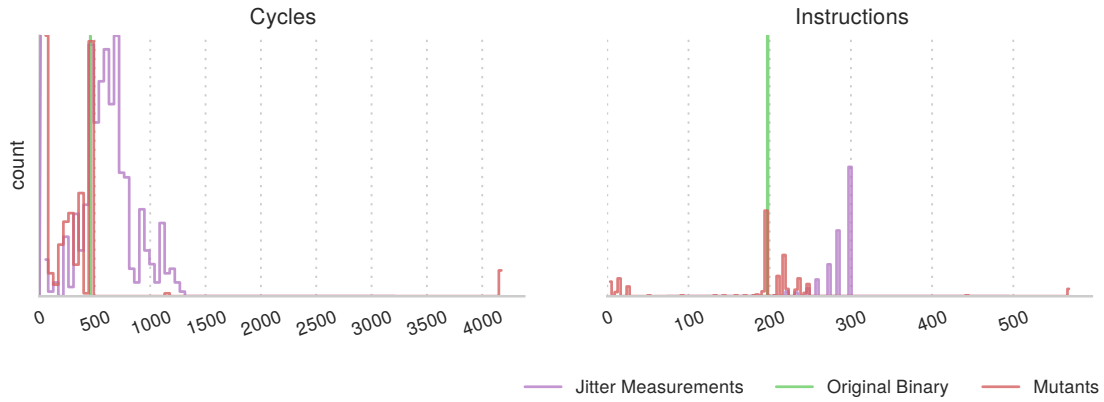

Figure A.33: Timing distribution (gem5) – *binsearch*

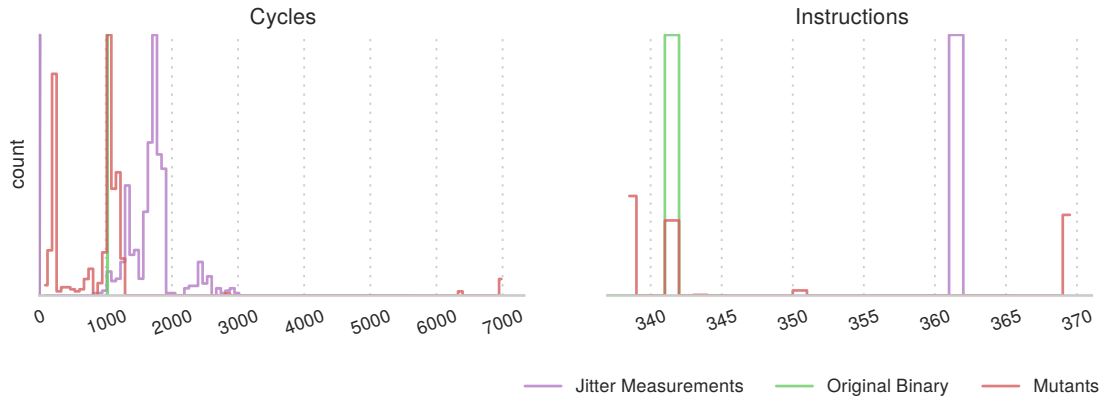

Figure A.34: Timing distribution (gem5) – *binsearch_wcet*

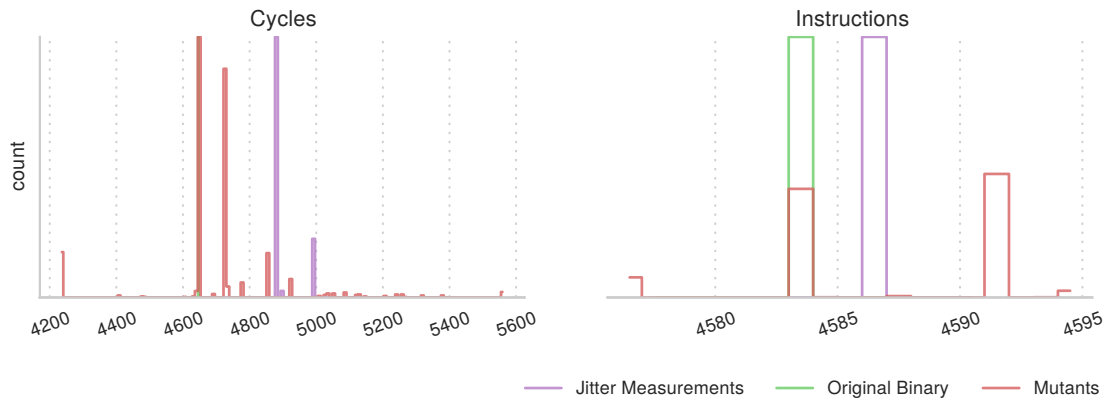

Figure A.35: Timing distribution (gem5) – *duff*

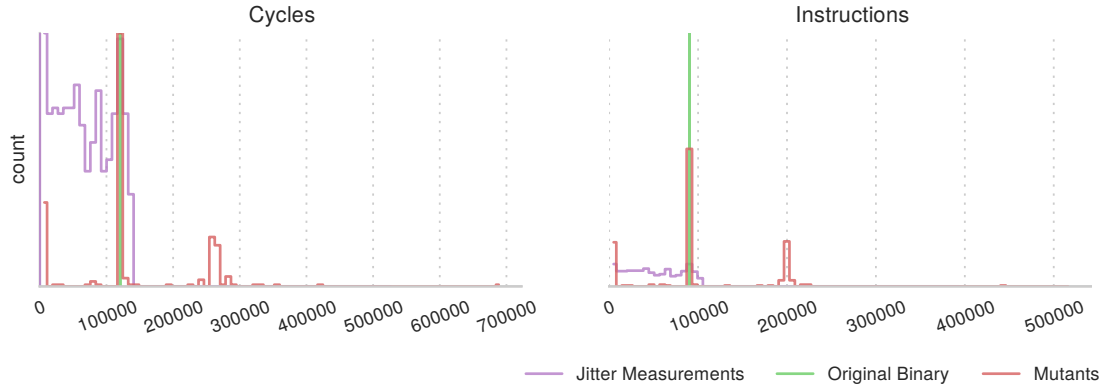

Figure A.36: Timing distribution (gem5) – *strstr*

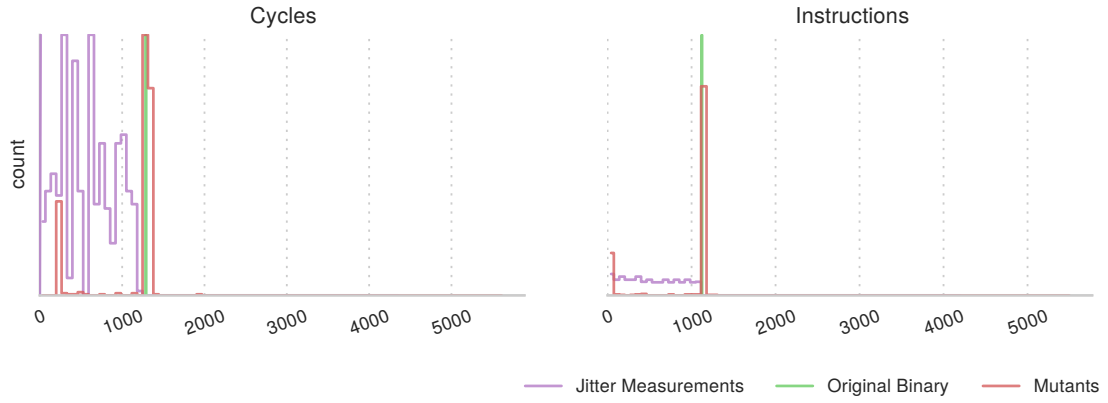

Figure A.37: Timing distribution (gem5) – *fibcall*

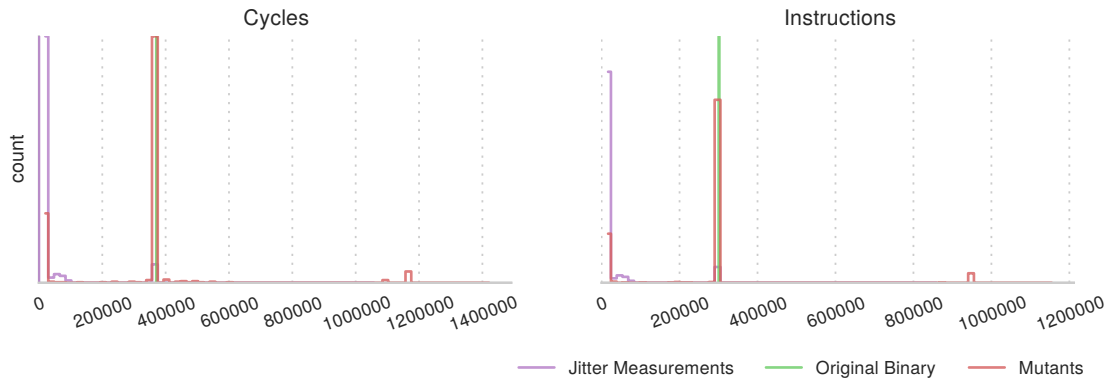

Figure A.38: Timing distribution (gem5) – *prime*

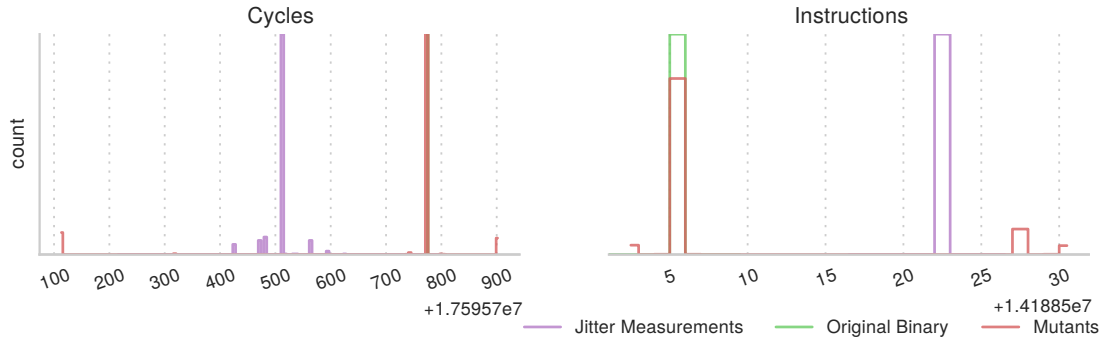

Figure A.39: Timing distribution (gem5) – *prime_wcet*

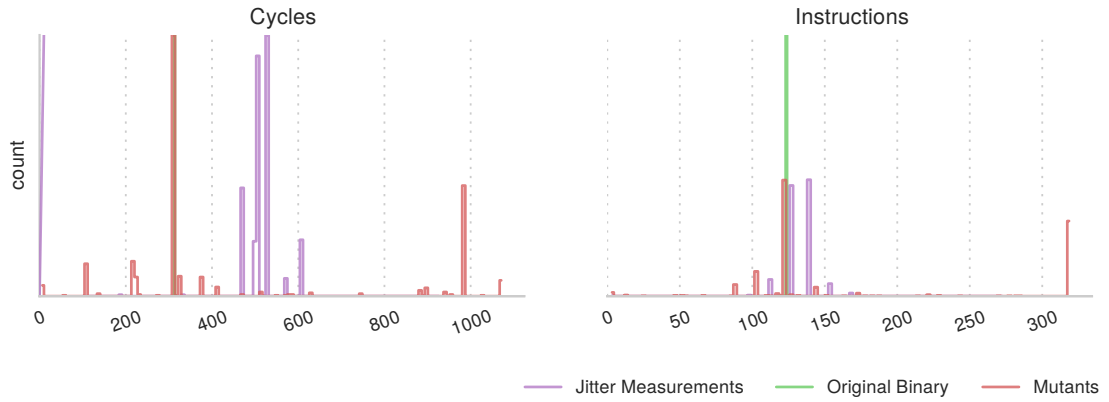

Figure A.40: Timing distribution (gem5) – *sqrt_micro*

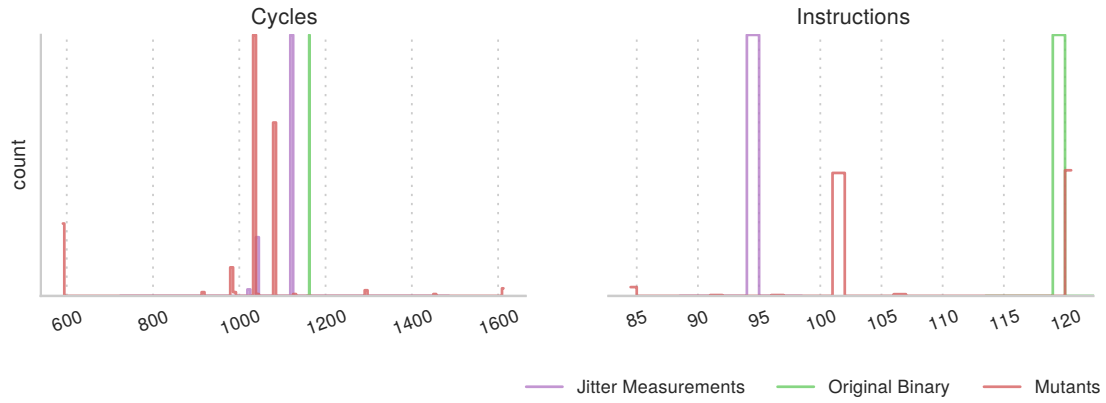

Figure A.41: Timing distribution (gem5) – *sqrt_micro_wcet*

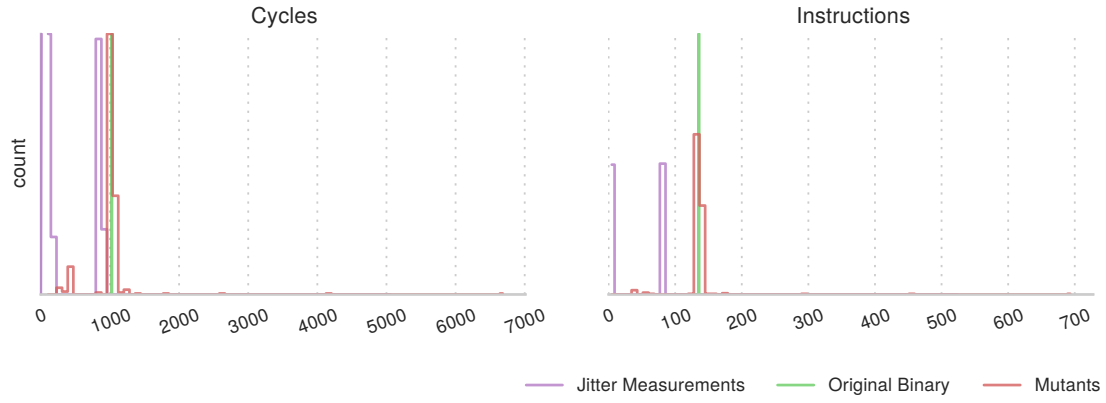

Figure A.42: Timing distribution (gem5) – *sqrt_micro_wcet_neg*

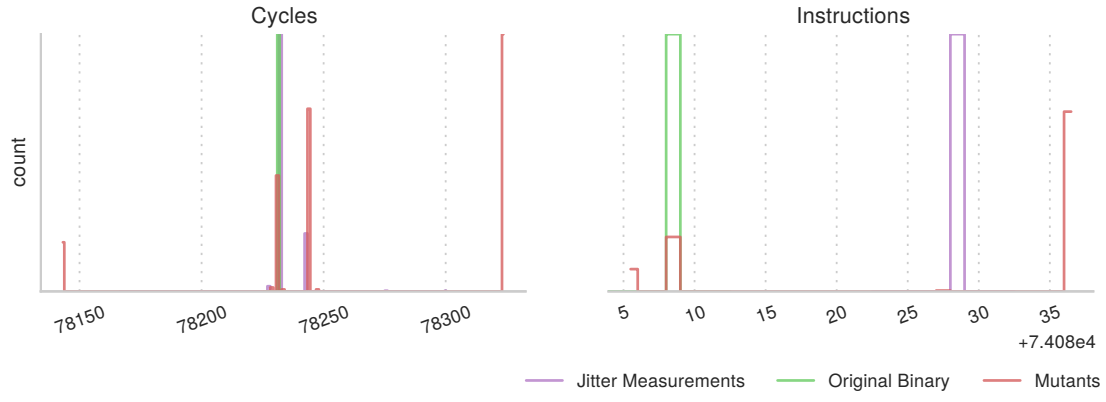

Figure A.43: Timing distribution (gem5) – *matmul*

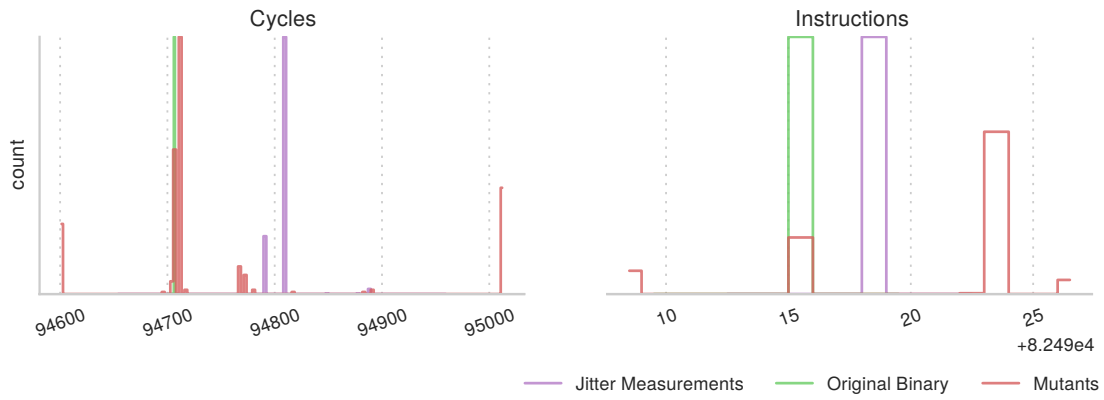

Figure A.44: Timing distribution (gem5) – *matmul_float*

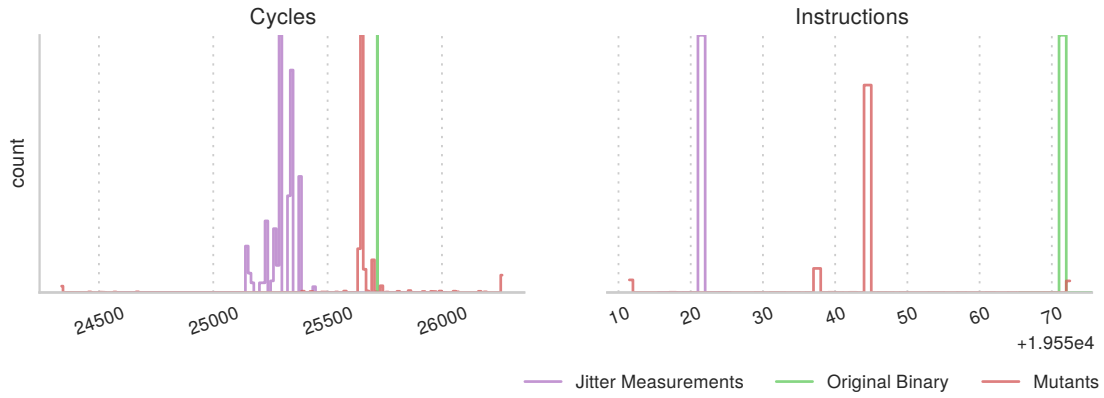

Figure A.45: Timing distribution (gem5) – *aes*

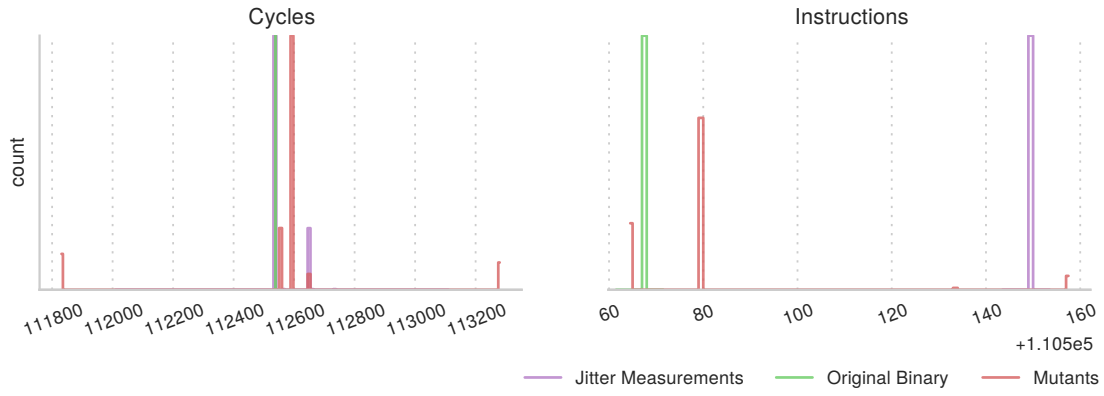

Figure A.46: Timing distribution (gem5) – *crc32*

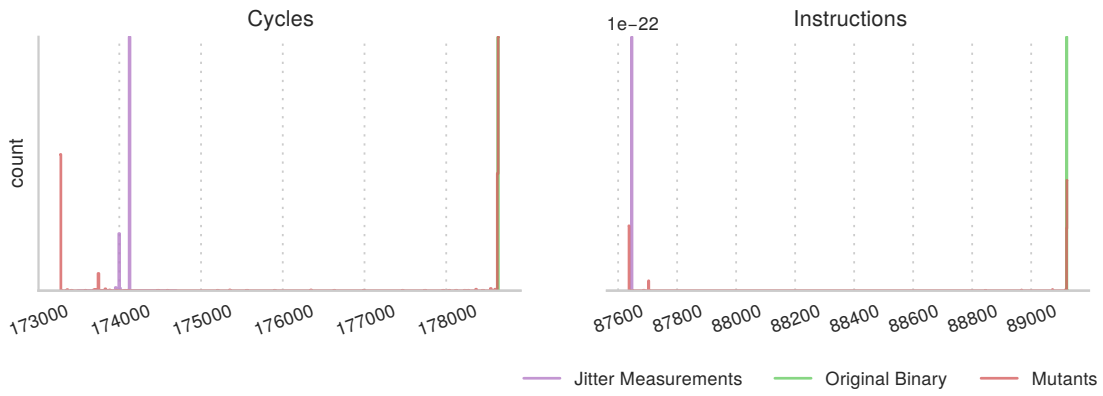

Figure A.47: Timing distribution (gem5) – *fft*

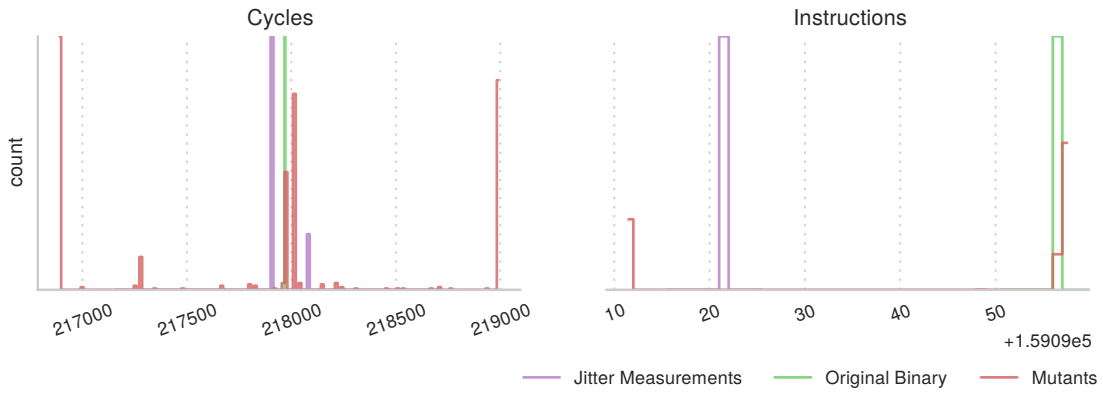

Figure A.48: Timing distribution (gem5) – *fir*

A.5 Timing Distributions – Raspberry Pi 2

The Raspberry Pi 2 results show some interesting (but expected) behavior:

- *nop* (figure A.49) clearly shows some baseline jitter for the cycles counter
- *long_nop* (figure A.50) and *bubblesort_wcet* (figure A.52) show how this baseline jitter accumulates to an approximately normal distributed peak – note that the Instruction value is strictly constant in both cases.
- *binsearch* (figure A.57) and *binsearch_wcet* (figure A.58) exhibit a very similar behavior with their cycle count – with the former being faster on average, and the latter with a tighter peak, whereas in the instruction counter the difference between the two is much more pronounced.

- *sqrt_micro_wcet_neg* (figure A.66) shows two distinct peaks, as predicted in section 3.2.) As expected, some *MUTANTs* lay in-between those peaks – hinting at possible future improvements.

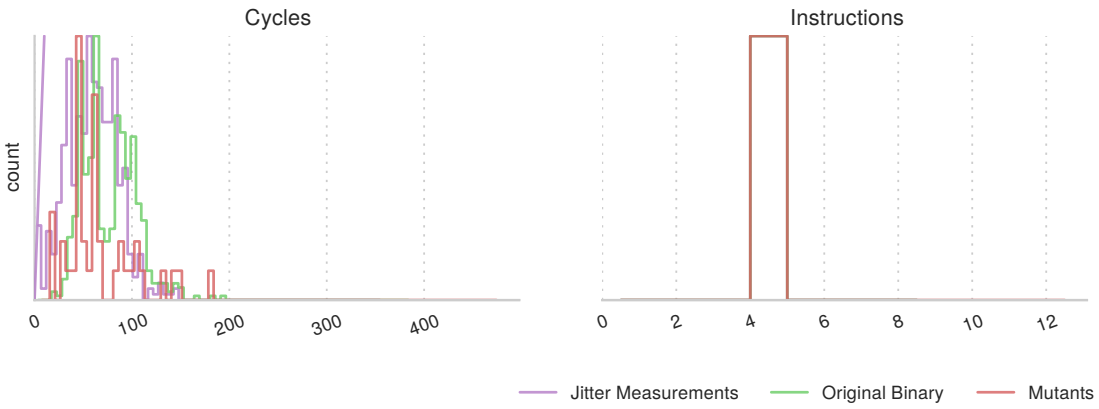

Figure A.49: Timing distribution (Raspberry Pi 2) – *nop*

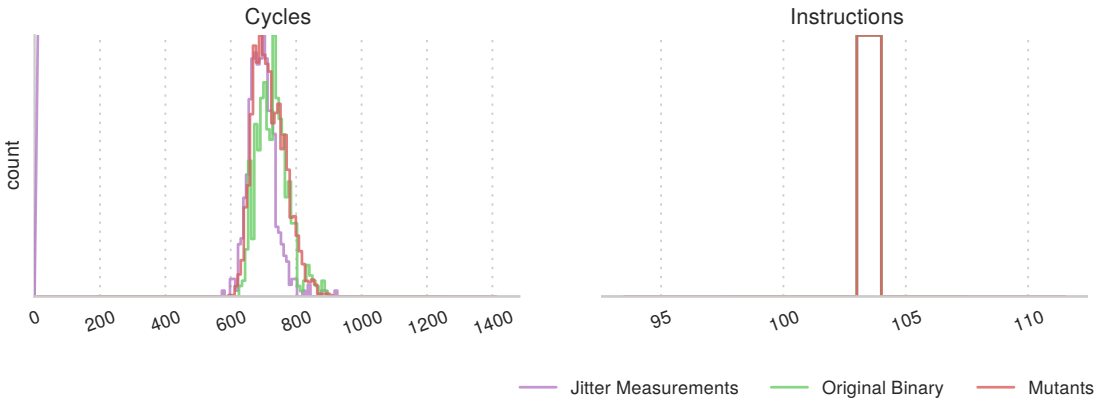

Figure A.50: Timing distribution (Raspberry Pi 2) – *long_nop*

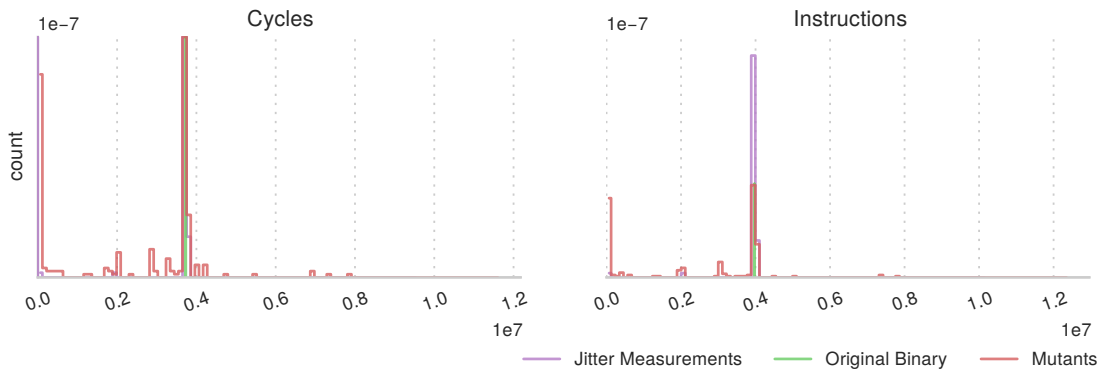

Figure A.51: Timing distribution (Raspberry Pi 2) – *bubblesort*

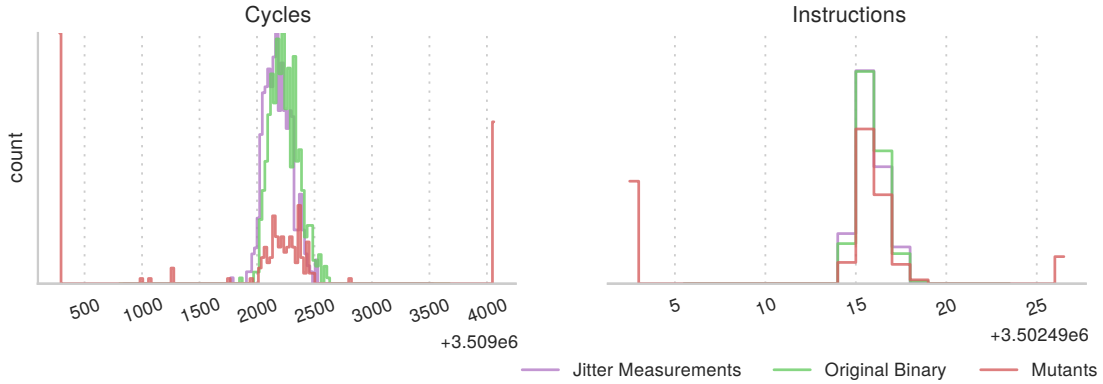

Figure A.52: Timing distribution (Raspberry Pi 2) – *bubblesort_wcet*

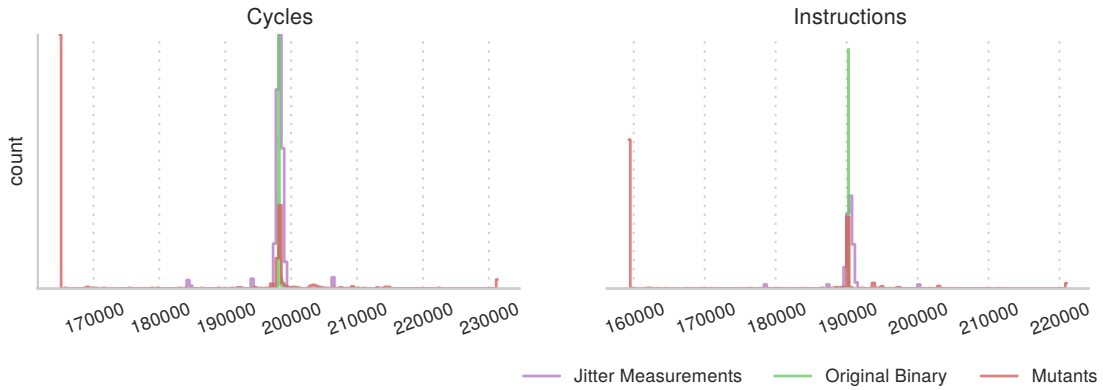

Figure A.53: Timing distribution (Raspberry Pi 2) – *heapsort*

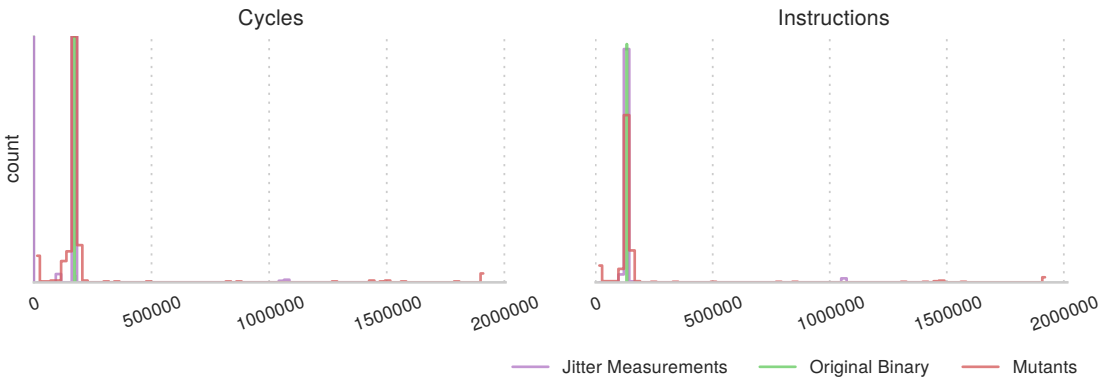

Figure A.54: Timing distribution (Raspberry Pi 2) – *quicksort*

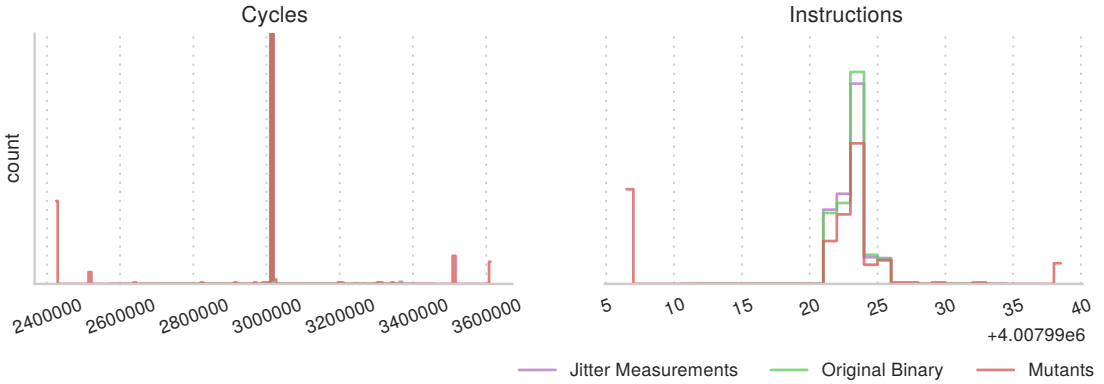

Figure A.55: Timing distribution (Raspberry Pi 2) – *selectionsort*

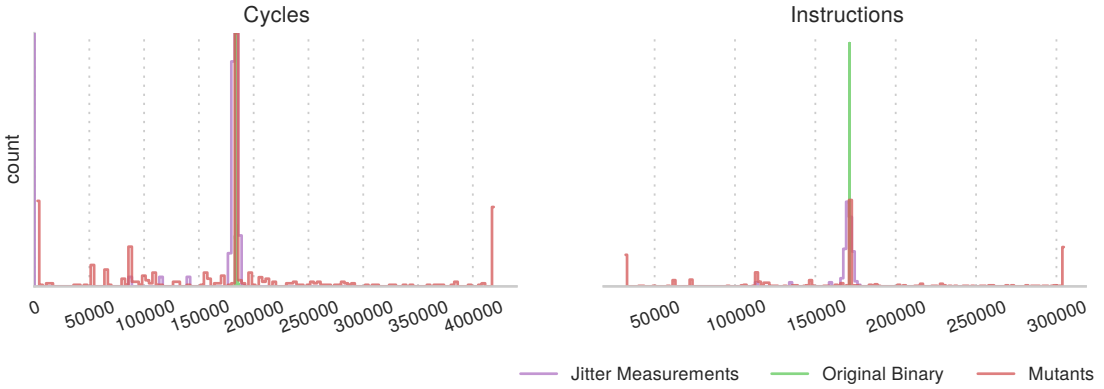

Figure A.56: Timing distribution (Raspberry Pi 2) – *shellsort*

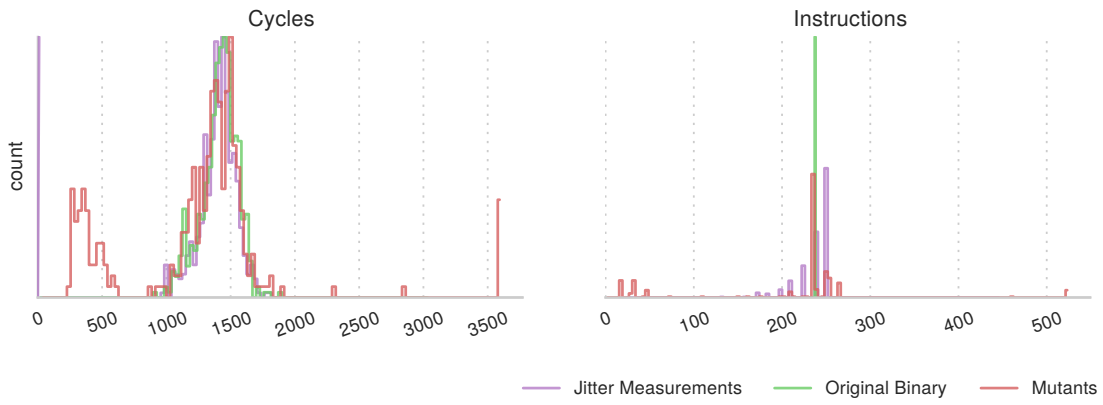

Figure A.57: Timing distribution (Raspberry Pi 2) – *binsearch*

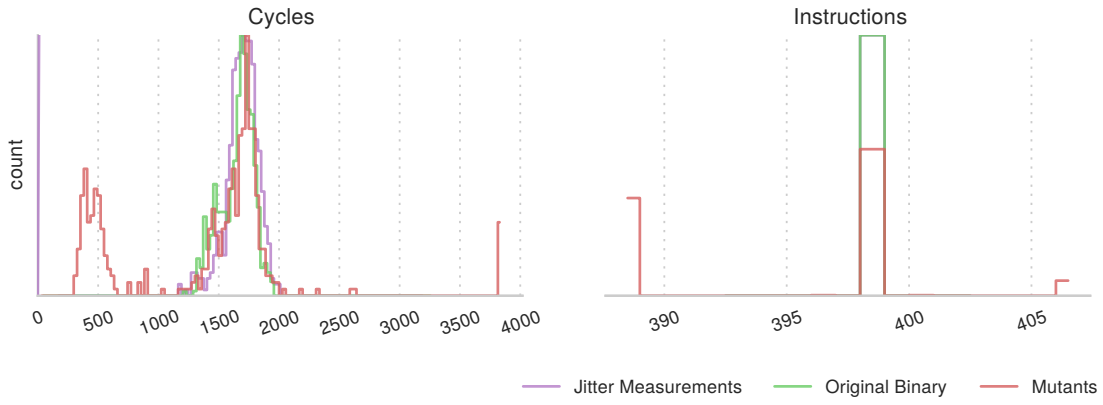

Figure A.58: Timing distribution (Raspberry Pi 2) – *binsearch_wcet*

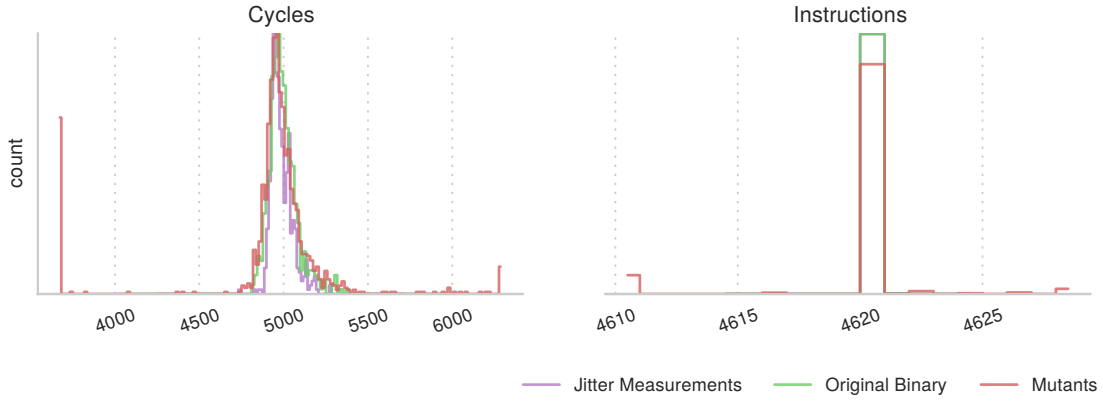

Figure A.59: Timing distribution (Raspberry Pi 2) – *duff*

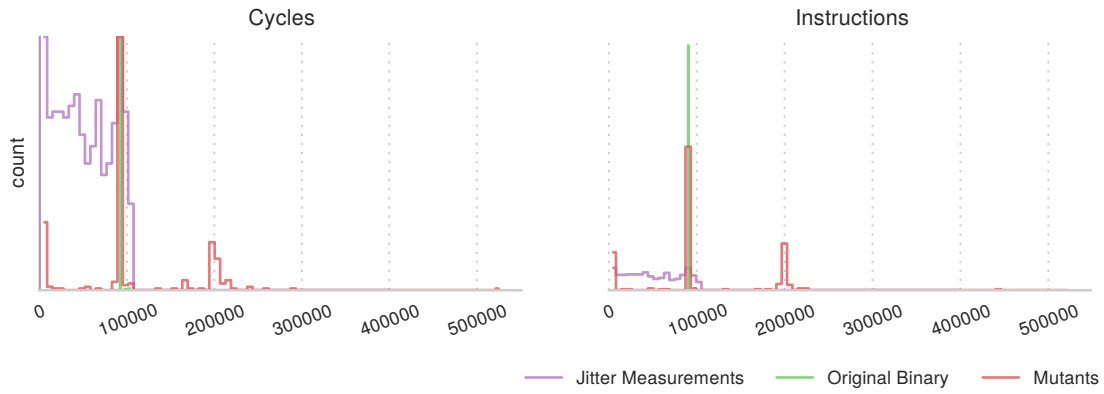

Figure A.60: Timing distribution (Raspberry Pi 2) – *strstr*

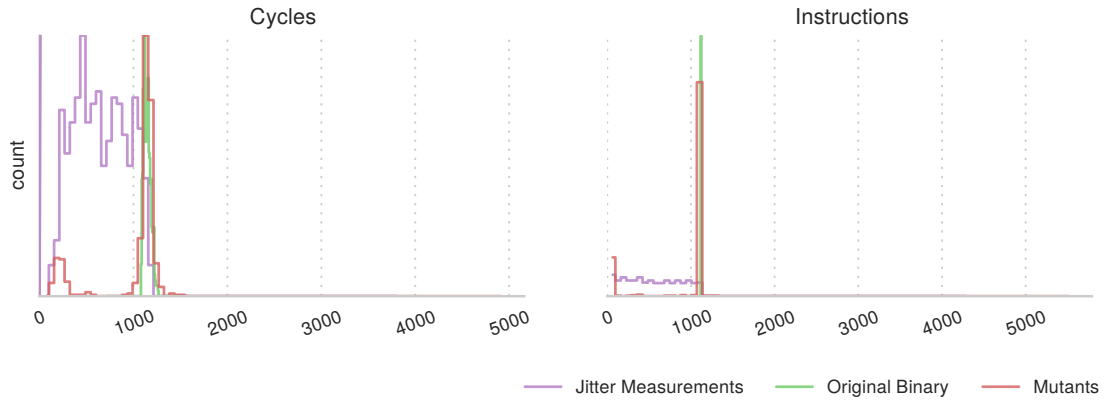

Figure A.61: Timing distribution (Raspberry Pi 2) – *fibcall*

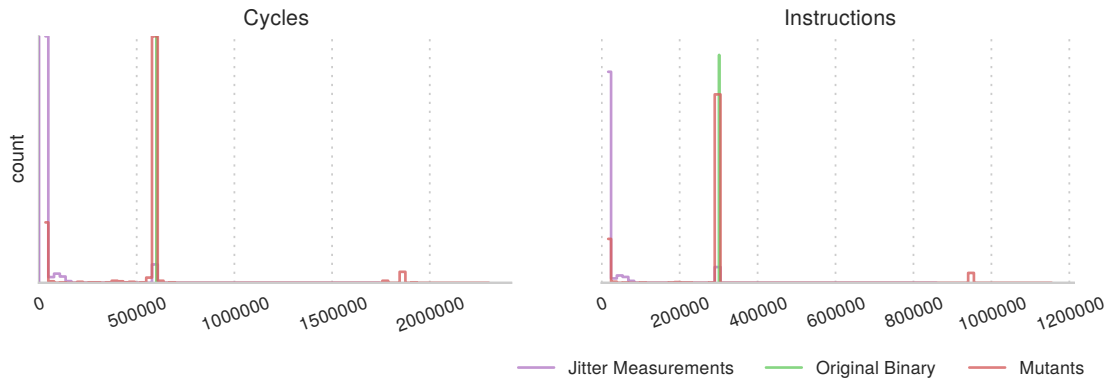

Figure A.62: Timing distribution (Raspberry Pi 2) – *prime*

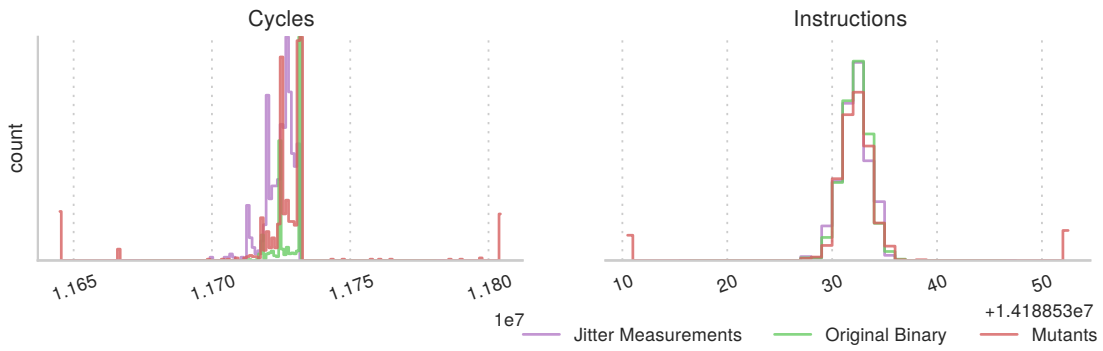

Figure A.63: Timing distribution (Raspberry Pi 2) – *prime_wcet*

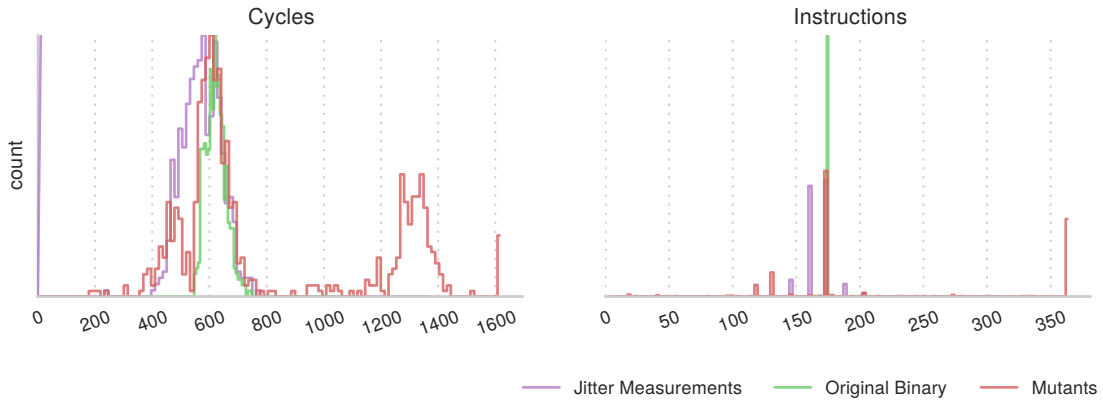

Figure A.64: Timing distribution (Raspberry Pi 2) – *sqrt_micro*

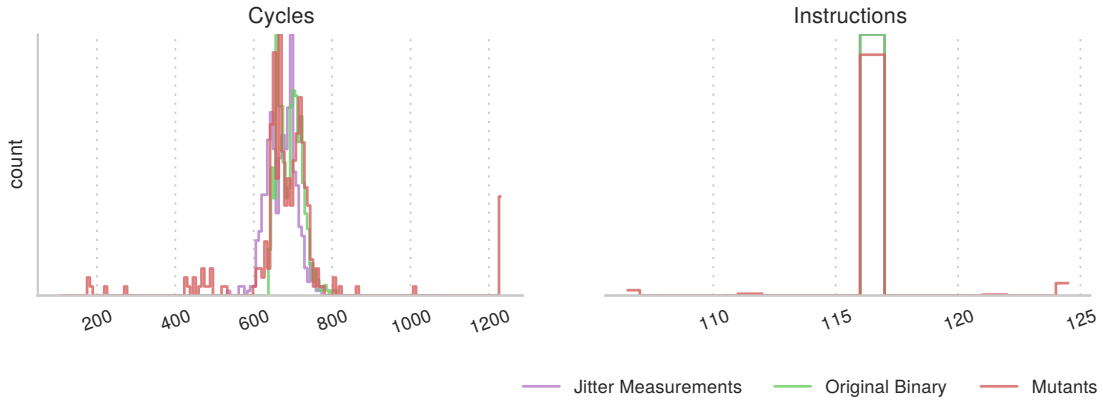

Figure A.65: Timing distribution (Raspberry Pi 2) – *sqrt_micro_wcet*

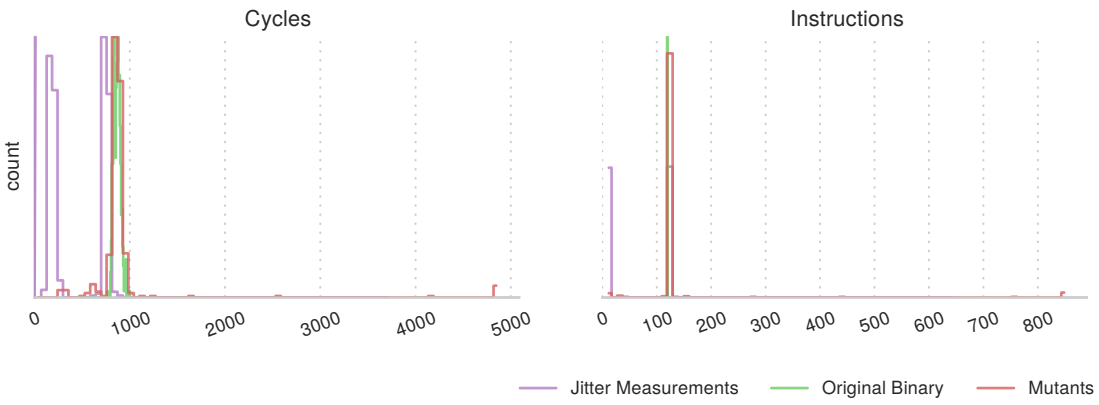

Figure A.66: Timing distribution (Raspberry Pi 2) – *sqrt_micro_wcet_neg*

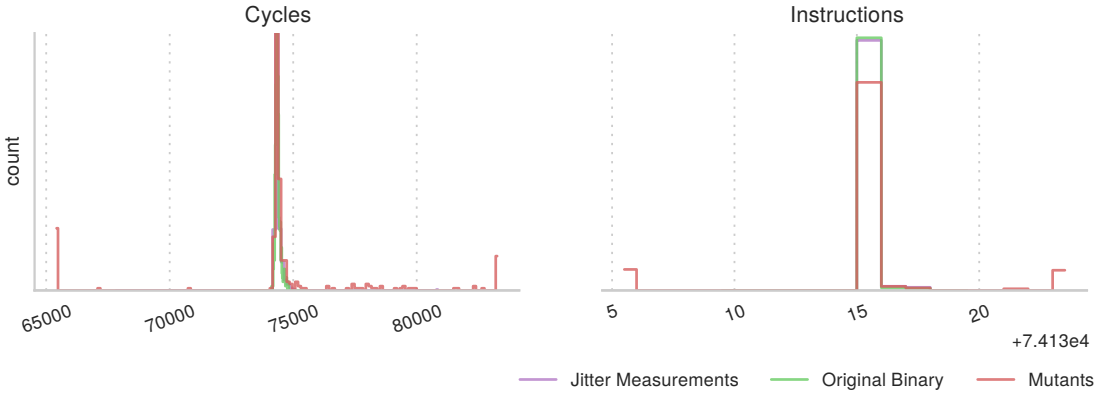

Figure A.67: Timing distribution (Raspberry Pi 2) – *matmul*

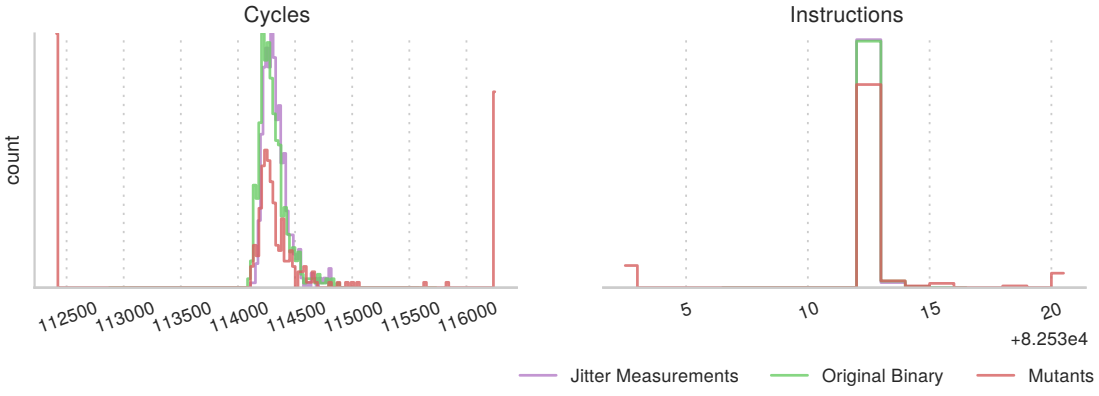

Figure A.68: Timing distribution (Raspberry Pi 2) – *matmul_float*

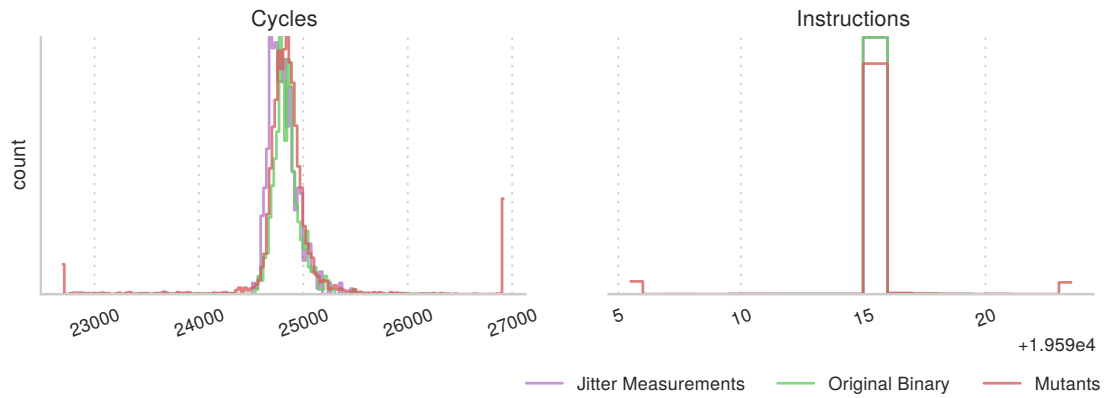

Figure A.69: Timing distribution (Raspberry Pi 2) – *aes*

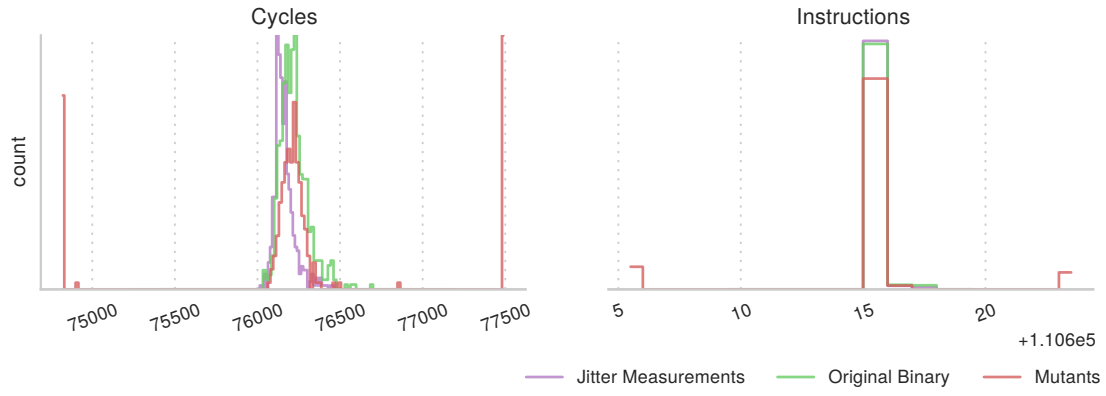

Figure A.70: Timing distribution (Raspberry Pi 2) – *crc32*

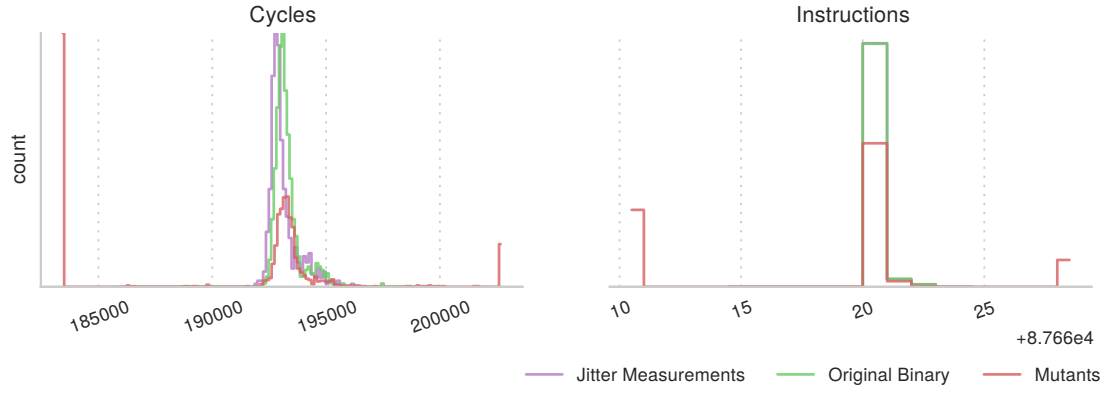

Figure A.71: Timing distribution (Raspberry Pi 2) – *fft*

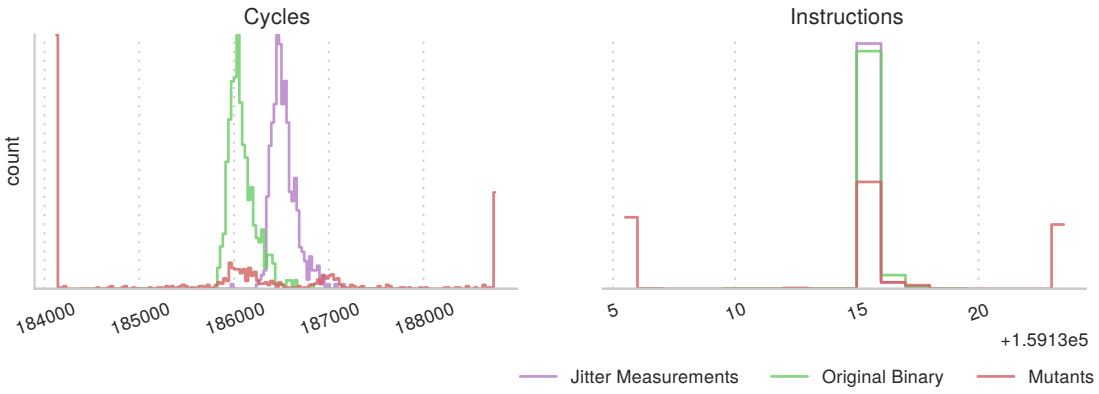

Figure A.72: Timing distribution (Raspberry Pi 2) – *fir*

A.6 Detection Results

This section shows the detailed detection results for the *MUTANT* runs. (c.p. section 7.4)

A.6.1 Complete Results

	Runtime (%)	Runtime (%) Both Criteria	Runtime (%) Cycles only	Runtime (%) Instructions only	Timeout (%)	Crash (%)	Not detected (Ok) (%)	Not detected (Error) (%)
nop	55.6	0.0	55.6	0.0	0.0	44.4	0.0	0.0
long_nop	67.5	0.0	0.0	67.5	0.0	32.5	0.0	0.0
bubblesort	15.6	7.7	1.7	6.2	4.2	54.0	11.4	14.7
bubblesort_wcet	33.2	17.8	0.8	14.6	3.7	53.3	9.0	0.8
heapsort	33.6	32.8	0.6	0.2	3.1	45.0	10.8	7.5
quicksort	3.4	3.1	0.0	0.3	23.7	44.8	19.3	8.7
selectionsort	16.6	11.4	5.1	0.1	4.7	52.8	6.4	19.5
shellsort	17.5	16.0	0.1	1.5	5.3	59.7	12.3	5.2
binsearch	15.1	11.5	3.0	0.7	11.2	35.7	21.5	16.4
binsearch_wcet	36.7	20.0	0.6	16.2	8.0	39.6	14.6	1.0
duff	45.4	10.2	1.1	34.2	2.3	21.6	30.6	0.0
strstr	24.8	24.6	0.2	0.0	5.4	33.5	30.4	5.8
fibcall	33.1	32.1	0.7	0.3	2.6	20.5	28.7	15.1
prime	19.4	16.6	2.8	0.0	3.6	18.4	54.4	4.2
prime_wcet	16.4	5.3	4.0	7.0	14.0	24.4	45.2	0.0
sqrt_micro	27.2	19.0	4.5	3.7	4.4	27.7	31.8	8.8
sqrt_micro_wcet	33.5	11.9	1.7	20.0	8.8	29.8	27.9	0.0
sqrt_micro_wcet_neg	24.5	17.8	0.3	6.4	6.9	26.2	40.6	1.7
matmul	29.3	17.4	0.8	11.1	21.2	42.6	6.9	0.0
matmul_float	29.1	10.9	0.1	18.0	9.3	53.6	8.1	0.0
aes	7.5	3.7	2.4	1.4	5.8	40.1	5.0	41.6
crc32	15.8	6.9	0.7	8.2	17.0	35.2	0.0	31.9
fft	35.8	31.9	3.5	0.3	1.8	51.6	10.3	0.6
fir	47.9	34.5	0.3	13.2	2.8	41.7	7.6	0.1

Table A.3: Detection results — gem5

	Runtime (%)	Runtime (%) Both Criteria	Runtime (%) Cycles only	Runtime (%) Instructions only	Timeout (%)	Crash (%)	Not detected (Ok) (%)	Not detected (Error) (%)
nop	0.0	0.0	0.0	0.0	0.0	37.5	62.5	0.0
long_nop	0.0	0.0	0.0	0.0	0.0	22.2	77.8	0.0
bubblesort	11.2	7.4	3.5	0.4	2.8	55.7	11.4	18.9
bubblesort_wcet	17.6	12.7	4.3	0.6	1.8	55.8	9.6	15.3
heapsort	33.1	32.5	0.6	0.1	2.2	46.8	10.9	6.9
quicksort	4.0	3.7	0.1	0.3	15.3	53.4	19.0	8.2
selectionsort	15.8	10.9	4.1	0.7	3.0	57.0	7.2	17.0
shellsort	17.4	15.9	0.6	0.9	2.1	62.2	12.4	6.0
binsearch	18.3	13.2	0.0	5.1	10.5	36.7	21.4	13.2
binsearch_wcet	23.0	15.4	1.1	6.5	7.1	42.9	14.6	12.4
duff	9.7	6.4	0.8	2.5	0.1	24.3	31.5	34.4
strstr	23.3	22.9	0.4	0.0	4.2	35.8	30.8	5.8
fibcall	7.5	0.3	1.4	5.8	2.6	22.9	28.7	38.4
prime	4.2	3.1	0.6	0.5	3.6	18.9	66.7	6.6
prime_wcet	7.8	4.0	2.5	1.3	13.1	25.8	46.4	6.8
sqrt_micro	22.1	21.2	0.5	0.4	3.3	28.9	32.0	13.7
sqrt_micro_wcet	8.5	4.0	3.8	0.8	7.5	30.8	29.2	24.0
sqrt_micro_wcet_neg	3.3	2.0	0.3	0.9	5.5	27.8	40.2	23.3
matmul	7.2	4.5	0.9	1.7	1.7	61.2	8.8	21.1
matmul_float	11.1	5.8	5.0	0.2	1.2	62.9	8.3	16.6
aes	5.8	3.5	0.5	1.8	1.6	44.9	5.0	42.6
crc32	12.2	7.1	4.9	0.2	8.9	44.3	8.2	26.6
fft	19.3	14.1	0.3	4.9	0.2	53.2	10.4	16.9
fir	38.8	20.7	9.2	8.9	0.6	44.5	8.3	7.7

Table A.4: Detection results — Raspberry Pi 2

A.6.2 Online Results

This tables exclude the *crash* results – i.e. they only contain online detections.

	Runtime (%)	Runtime (%) Both Criteria	Runtime (%) Cycles only	Runtime (%) Instructions only	Timeout (%)	Not detected (Ok) (%)	Not detected (Error) (%)
nop	100.0	0.0	100.0	0.0	0.0	0.0	0.0
long_nop	100.0	0.0	0.0	100.0	0.0	0.0	0.0
bubblesort	34.0	16.8	3.6	13.6	9.2	24.8	32.0
bubblesort_wcet	71.1	38.1	1.7	31.4	7.9	19.2	1.7
heapsort	61.1	59.7	1.1	0.3	5.7	19.5	13.6
quicksort	6.2	5.7	0.0	0.5	43.0	35.0	15.8
selectionsort	35.2	24.1	10.8	0.3	9.9	13.6	41.3
shellsort	43.5	39.7	0.2	3.6	13.2	30.5	12.8
binsearch	23.5	17.9	4.6	1.0	17.4	33.5	25.6
binsearch_wcet	60.8	33.1	0.9	26.8	13.3	24.2	1.7
duff	57.9	13.0	1.4	43.6	3.0	39.1	0.0
strstr	37.3	37.0	0.3	0.0	8.2	45.8	8.8
fibcall	41.7	40.4	0.9	0.4	3.2	36.1	19.0
prime	23.7	20.4	3.4	0.0	4.4	66.7	5.2
prime_wcet	21.7	7.1	5.3	9.2	18.6	59.8	0.0
sqrt_micro	37.7	26.4	6.2	5.1	6.1	44.1	12.2
sqrt_micro_wcet	47.8	16.9	2.4	28.5	12.5	39.8	0.0
sqrt_micro_wcet_neg	33.3	24.2	0.4	8.7	9.3	55.1	2.3
matmul	51.0	30.2	1.4	19.4	36.9	12.1	0.0
matmul_float	62.7	23.6	0.3	38.9	19.9	17.4	0.0
aes	12.5	6.2	4.0	2.3	9.6	8.4	69.4
crc32	24.4	10.7	1.1	12.6	26.3	0.0	49.3
fft	73.9	66.0	7.3	0.6	3.6	21.2	1.3
fir	82.1	59.1	0.4	22.6	4.8	12.9	0.1

Table A.5: Detection results online— gem5

	Runtime (%)	Runtime (%) Both Criteria	Runtime (%) Cycles only	Runtime (%) Instructions only	Timeout (%)	Not detected (Ok) (%)	Not detected (Error) (%)
nop	0.0	0.0	0.0	0.0	0.0	100.0	0.0
long_nop	0.0	0.0	0.0	0.0	0.0	100.0	0.0
bubblesort	25.3	16.6	7.9	0.8	6.2	25.7	42.7
bubblesort_wcet	39.8	28.8	9.7	1.3	4.0	21.7	34.5
heapsort	62.3	61.1	1.1	0.1	4.1	20.6	13.0
quicksort	8.7	7.9	0.2	0.6	32.9	40.7	17.7
selectionsort	36.6	25.4	9.6	1.7	6.9	16.8	39.6
shellsort	46.0	42.1	1.5	2.3	5.6	32.7	15.8
binsearch	28.8	20.8	0.0	8.1	16.6	33.8	20.8
binsearch_wcet	40.2	27.0	2.0	11.3	12.5	25.6	21.7
duff	12.8	8.5	1.1	3.2	0.1	41.6	45.5
strstr	36.4	35.7	0.6	0.0	6.5	48.1	9.1
fibcall	9.8	0.4	1.8	7.6	3.3	37.2	49.7
prime	5.2	3.9	0.7	0.6	4.4	82.2	8.1
prime_wcet	10.5	5.4	3.3	1.8	17.6	62.6	9.2
sqrt_micro	31.0	29.8	0.7	0.5	4.7	45.1	19.2
sqrt_micro_wcet	12.3	5.7	5.4	1.2	10.8	42.2	34.6
sqrt_micro_wcet_neg	4.5	2.8	0.4	1.3	7.6	55.6	32.3
matmul	18.5	11.6	2.4	4.5	4.5	22.7	54.3
matmul_float	29.8	15.5	13.6	0.6	3.2	22.3	44.7
aes	10.6	6.4	0.9	3.3	2.9	9.1	77.4
crc32	21.8	12.8	8.7	0.3	15.9	14.6	47.7
fft	41.3	30.2	0.5	10.6	0.5	22.2	36.1
fir	70.0	37.2	16.6	16.1	1.1	15.0	13.9

Table A.6: Detection results online— Raspberry Pi 2

APPENDIX B

Source Code

B.1 gem5 patches

```
diff -r 629fe6e6c781 src/arch/arm/linux/process.cc
--- a/src/arch/arm/linux/process.cc Thu Jun 25 11:58:28 2015 -0500
+++ b/src/arch/arm/linux/process.cc Fri Nov 06 17:34:41 2015 +0100
@@ -483,7 +483,7 @@
     /* 361 */ SyscallDesc("sys_preadv", unimplementedFunc),
     /* 362 */ SyscallDesc("sys_pwritev", unimplementedFunc),
     /* 363 */ SyscallDesc("sys_rt_tgsigqueueinfo", unimplementedFunc),
-    /* 364 */ SyscallDesc("sys_perf_event_open", unimplementedFunc),
+    /* 364 */ SyscallDesc("sys_perf_event_open", ignoreFunc),
     /* 365 */ SyscallDesc("sys_recvmmsg", unimplementedFunc),
 };
```

Listing B.1: `sys_perf_event_open` patch applied to gem5

```
diff -r 629fe6e6c781 configs/common/CacheConfig.py
--- a/configs/common/CacheConfig.py Thu Jun 25 11:58:28 2015 -0500
+++ b/configs/common/CacheConfig.py Sat Nov 07 19:54:58 2015 +0100
@@ -62,6 +62,15 @@

         dcache_class, icache_class, l2_cache_class = \
             O3_ARM_v7a_DCache, O3_ARM_v7a_ICache, O3_ARM_v7aL2
+    elif options.cpu_type == "raspi":
+        try:
+            from O3_ARM_v7a_raspi import *
+        except:
+            print "raspi is unavailable. Did you compile the O3 model?"
+            sys.exit(1)
+
+        dcache_class, icache_class, l2_cache_class = \
+            O3_ARM_v7a_raspi_DCache, O3_ARM_v7a_raspi_ICache,
  O3_ARM_v7a_raspiL2
     else:
         dcache_class, icache_class, l2_cache_class = \
             L1Cache, L1Cache, L2Cache
diff -r 629fe6e6c781 configs/common/CpuConfig.py
```

```
--- a/configs/common/CpuConfig.py Thu Jun 25 11:58:28 2015 -0500
+++ b/configs/common/CpuConfig.py Sat Nov 07 19:54:58 2015 +0100
@@ -116,6 +116,14 @@
 except:
     pass

+# The raspi detailed CPU is special in the sense that it doesn't exist
+# in the normal object hierarchy, so we have to add it manually.
+try:
+    from O3_ARM_v7a_raspi import O3_ARM_v7a_raspi_3
+    _cpu_classes["raspi"] = O3_ARM_v7a_raspi_3
+except:
+    pass
+
 # Add all CPUs in the object hierarchy.
 for name, cls in inspect.getmembers(m5.objects, is_cpu_class):
     _cpu_classes[name] = cls
diff -r 629fe6e6c781 configs/common/O3_ARM_v7a_raspi.py
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/configs/common/O3_ARM_v7a_raspi.py  Sat Nov 07 19:54:58 2015 +0100
@@ -0,0 +1,198 @@
+# Copyright (c) 2012 The Regents of The University of Michigan
+# All rights reserved.
+#
+# Redistribution and use in source and binary forms, with or without
+# modification, are permitted provided that the following conditions are
+# met: redistributions of source code must retain the above copyright
+# notice, this list of conditions and the following disclaimer;
+# redistributions in binary form must reproduce the above copyright
+# notice, this list of conditions and the following disclaimer in the
+# documentation and/or other materials provided with the distribution;
+# neither the name of the copyright holders nor the names of its
+# contributors may be used to endorse or promote products derived from
+# this software without specific prior written permission.
+#
+# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
+# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
+# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
+# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
+# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
+# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
+# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
+# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
+# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
+# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
+# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
+#
+# Authors: Ron Dreslinski
+
+# modified according to Micro-architectural Simulation of In-order and
+# Out-of-order ARM Microprocessors with gem5
+# by Endo et al
+# and
+# Micro-architectural simulation of embedded core
+# heterogeneity with gem5 and McPAT
+# by Endo et al
+# to closely emulate a raspberry pi2 core
+from m5.objects import *
+
+# Simple ALU Instructions have a latency of 1
+class O3_ARM_v7a_raspi_Simple_Int(FUDesc):
```

```
82 +    opList = [ OpDesc(opClass='IntAlu', opLat=1) ]
83 +    count = 1
84 +
85 +# Complex ALU instructions have a variable latencies
86 +class O3_ARM_v7a_raspi_Complex_Int(FUDesc):
87 +    opList = [ OpDesc(opClass='IntMult', opLat=4, pipelined=True),
88 +               OpDesc(opClass='IntDiv', opLat=12, pipelined=False),
89 +               OpDesc(opClass='IprAccess', opLat=3, pipelined=True) ]
90 +    count = 1
91 +
92 +
93 +# Floating point and SIMD instructions
94 +class O3_ARM_v7a_raspi_FP(FUDesc):
95 +    opList = [ OpDesc(opClass='SimdAdd', opLat=4),
96 +               OpDesc(opClass='SimdAddAcc', opLat=4),
97 +               OpDesc(opClass='SimdAlu', opLat=4),
98 +               OpDesc(opClass='SimdCmp', opLat=4),
99 +               OpDesc(opClass='SimdCvt', opLat=3),
100 +               OpDesc(opClass='SimdMisc', opLat=3),
101 +               OpDesc(opClass='SimdMult',opLat=5),
102 +               OpDesc(opClass='SimdMultAcc',opLat=5),
103 +               OpDesc(opClass='SimdShift',opLat=3),
104 +               OpDesc(opClass='SimdShiftAcc', opLat=3),
105 +               OpDesc(opClass='SimdSqrt', opLat=9),
106 +               OpDesc(opClass='SimdFloatAdd',opLat=3),
107 +               OpDesc(opClass='SimdFloatAlu',opLat=5),
108 +               OpDesc(opClass='SimdFloatCmp', opLat=6),
109 +               OpDesc(opClass='SimdFloatCvt', opLat=7),
110 +               OpDesc(opClass='SimdFloatDiv', opLat=43),
111 +               OpDesc(opClass='SimdFloatMisc', opLat=4),
112 +               OpDesc(opClass='SimdFloatMult', opLat=4),
113 +               OpDesc(opClass='SimdFloatMultAcc',opLat=6),
114 +               OpDesc(opClass='SimdFloatSqrt', opLat=40),
115 +               OpDesc(opClass='FloatAdd', opLat=3),
116 +               OpDesc(opClass='FloatCmp', opLat=6),
117 +               OpDesc(opClass='FloatCvt', opLat=7),
118 +               OpDesc(opClass='FloatDiv', opLat=43),
119 +               OpDesc(opClass='FloatSqrt', opLat=40),
120 +               OpDesc(opClass='FloatMult', opLat=4)]
121 +    count = 2
122 +
123 +
124 +# Load/Store Units
125 +class O3_ARM_v7a_raspi_Load(FUDesc):
126 +    opList = [ OpDesc(opClass='MemRead',opLat=1) ]
127 +    count = 1
128 +
129 +class O3_ARM_v7a_raspi_Store(FUDesc):
130 +    opList = [OpDesc(opClass='MemWrite',opLat=1) ]
131 +    count = 1
132 +
133 +# Functional Units for this CPU
134 +class O3_ARM_v7a_raspi_FUP(FUPool):
135 +    FUList = [O3_ARM_v7a_raspi_Simple_Int(), O3_ARM_v7a_raspi_Complex_Int(),
136 +              O3_ARM_v7a_raspi_Load(), O3_ARM_v7a_raspi_Store(),
    O3_ARM_v7a_raspi_FP()]
137 +
138 +# Bi-Mode Branch Predictor
139 +class O3_ARM_v7a_raspi_BP(BiModeBP):
140 +    globalPredictorSize = 256
141 +    globalCtrBits = 2
```

```
142 +    choicePredictorSize = 256
143 +    choiceCtrBits = 2
144 +    BTBEntries = 256
145 +    BTBTagSize = 18
146 +    RASSize = 8
147 +    instShiftAmt = 2
148 +
149 +class O3_ARM_v7a_raspi_3(DerivO3CPU):
150 +    LQEntries = 8
151 +    SQEntries = 8
152 +    LSQDepCheckShift = 0
153 +    LFSTSize = 1024
154 +    SSITSize = 1024
155 +    decodeToFetchDelay = 1
156 +    renameToFetchDelay = 1
157 +    iewToFetchDelay = 1
158 +    commitToFetchDelay = 1
159 +    renameToDecodeDelay = 1
160 +    iewToDecodeDelay = 1
161 +    commitToDecodeDelay = 1
162 +    iewToRenameDelay = 1
163 +    commitToRenameDelay = 1
164 +    commitToIEWDelay = 1
165 +    fetchWidth = 3
166 +    fetchBufferSize = 16
167 +    fetchToDecodeDelay = 3
168 +    decodeWidth = 1 #3
169 +    decodeToRenameDelay = 2
170 +    renameWidth = 512
171 +    renameToIEWDelay = 1
172 +    issueToExecuteDelay = 1
173 +    dispatchWidth = 512
174 +    issueWidth = 2
175 +    wbWidth = 2 #6
176 +    fuPool = O3_ARM_v7a_raspi_FUP()
177 +    iewToCommitDelay = 1
178 +    renameToROBDelay = 1
179 +    commitWidth = 512
180 +    squashWidth = 512
181 +    trapLatency = 13
182 +    backComSize = 5
183 +    forwardComSize = 5
184 +    numPhysIntRegs = 512+44
185 +    numPhysFloatRegs = 512+72
186 +    numIQEntries = 16
187 +    numROBEntries = 512
188 +
189 +    switched_out = False
190 +    branchPred = O3_ARM_v7a_raspi_BP()
191 +
192 +# Instruction Cache
193 +class O3_ARM_v7a_raspi_ICache(BaseCache):
194 +    hit_latency = 1
195 +    response_latency = 1
196 +    mshrs = 2
197 +    tgts_per_mshr = 8
198 +    size = '32kB'
199 +    assoc = 2
200 +    is_top_level = True
201 +    forward_snoops = False
202 +
```

```
+# Data Cache
+class O3_ARM_v7a_raspi_DCache(BaseCache):
+    hit_latency = 1
+    response_latency = 1
+    mshrs = 4
+    tgts_per_mshr = 8
+    size = '32kB'
+    assoc = 4
+    write_buffers = 4
+    is_top_level = True
+
+# TLB Cache
+# Use a cache as a L2 TLB
+class O3_ARM_v7a_raspiWalkCache(BaseCache):
+    hit_latency = 4
+    response_latency = 4
+    mshrs = 6
+    tgts_per_mshr = 8
+    size = '1kB'
+    assoc = 8
+    write_buffers = 16
+    is_top_level = True
+    forward_snoops = False
+
+# L2 Cache
+class O3_ARM_v7a_raspiL2(BaseCache):
+    hit_latency = 3
+    response_latency = 3
+    mshrs = 8
+    tgts_per_mshr = 8
+    size = '512kB'
+    assoc = 8
+    write_buffers = 16
+    prefetch_on_access = True
+    # Simple stride prefetcher
+    prefetcher = StridePrefetcher(degree=1, latency = 1)
+    tags = RandomRepl()
diff -r 629fe6e6c781 src/cpu/o3/impl.hh
--- a/src/cpu/o3/impl.hh  Thu Jun 25 11:58:28 2015 -0500
+++ b/src/cpu/o3/impl.hh  Sat Nov 07 19:54:58 2015 +0100
@@ -76,7 +76,7 @@
     typedef O3CPU CPUType;

     enum {
-      MaxWidth = 8,
+      MaxWidth = 512,
       MaxThreads = 4
     };
 };
```

Listing B.2: Patch to add the `raspi` CPU to gem5

B.2 perflib

```
// Perflib
// A C++ wrapper around perf_event.h
// This helps in using the performance counters, provided by the linux kernel
// It is not a complete abstraction, but it exposes the most important parts
// as needed by the rest of this thesis
//
```

```
7  // Dieter Steiner - 2015-2016
8  // e0326004@student.tuwien.ac.at
9  // spoilerhead@gmail.com
10 //
11 // References:
12 // https://perf.wiki.kernel.org/index.php/Main_Page
13 // http://man7.org/linux/man-pages/man2/perf_event_open.2.html
14 //-----------------------------------------------------------------------------
15 #ifndef PERFLIB_H
16 #define PERFLIB_H
17
18 //#define DEBUG_PERFLIB //define to enable debugging mode
19 #include <cstdint>
20 #include <unistd.h>
21
22 #include <linux/perf_event.h>
23
24 //define this macro to disable performance events
25 #ifndef NO_PERF
26
27 #include <cstdlib>
28 #include <cstring>
29 #include <exception>
30 #include <sys/ioctl.h>
31
32 #ifdef DEBUG_PERFLIB
33 #include <cerrno>
34 #include <cstring>
35 #include <iostream>
36 #endif //DEBUG_PERFLIB
37
38 /// The actual perf_event_open call
39 static inline long perf_event_open(struct perf_event_attr *hw_event,
40         pid_t pid,
41         int cpu,
42         int group_fd,
43         unsigned long flags) {
44    int ret;
45    ret = syscall(__NR_perf_event_open, hw_event, pid, cpu, group_fd, flags);
46    return ret;
47 }
48
49 /// exception class, used when the syscall fails
50 class PerfEventException : public std::exception {};
51
52 class PerfEvent {
53     public:
54         // No event group specified
55         static const int NO_GROUP = -1;
56
57         PerfEvent(const uint64_t  p_event,
58                 const int group = NO_GROUP,
59                 const pid_t pid = 0,
60                 const int cpu = -1) {
61             fd = 0; //initialize file descriptor as zero
62             memset(&pe, 0, sizeof(struct perf_event_attr)); //clear structure
63
64             pe.type = PERF_TYPE_HARDWARE;                   //we want hardware
65             pe.size = sizeof(struct perf_event_attr);
66             pe.config = p_event;
67             if( group != NO_GROUP) { //SLAVE
```

```
68                  // slave events are always enabled, they get controled by the
69                  // master event
70                  pe.disabled = 0;
71              } else { //GROUP LEADER
72                  // master calls are disabled by default and enabled by Start()
73                  pe.disabled = 1;
74                  //can only be set for group leader
75                  pe.pinned  =1;   // must always be on PMU
76                  //pe.exclusive = 1;  // only group on PMU
77              }
78              pe.exclude_kernel = 1;                          //no kernel times
79              pe.exclude_hv = 1;                              //no hypervisor
80
81              fd = perf_event_open(&pe, pid, cpu, group, 0); // syscall
82              if (fd == -1) {
83                  // syscall failed
84                  #ifdef DEBUG_PERFLIB
85                  std::cerr<< std::strerror(errno) <<std::endl;
86                  #endif
87                  throw PerfEventException();
88              }
89              //if we're running under gem5 (arm emulator) fd will be 0
90              //on real machines this is not possible, as fd 0 is stdin
91          }
92
93          ~PerfEvent() {
94              if(fd > 0) { //don't do anything when emulating (gem5) or on errors
95                  close(fd);
96              }
97          }
98
99          inline void Start() {
100             if(fd > 0) { //don't do anything when emulating (gem5) or on errors
101                 ioctl(fd, PERF_EVENT_IOC_RESET, PERF_IOC_FLAG_GROUP);
102                 ioctl(fd, PERF_EVENT_IOC_ENABLE, PERF_IOC_FLAG_GROUP);
103             }
104         }
105
106         inline void Stop() {
107             if(fd > 0) { //don't do anything when emulating (gem5) or on errors
108                 ioctl(fd, PERF_EVENT_IOC_DISABLE, PERF_IOC_FLAG_GROUP);
109             }
110         }
111
112         uint64_t Count() {
113             if(fd > 0) { //don't do anything when emulating (gem5) or on errors
114                 uint64_t count;
115                 read(fd, &count, sizeof(long long));
116                 return count;
117             } else {
118                 return 0;
119             }
120         }
121
122         int Group() const {
123             return fd;
124         }
125
126     private:
127         int fd; // file descriptor of event, also group id, always > 2
128                 // (0,1,2 are specfial FDs)
```

```
129         struct perf_event_attr pe;
130 };
131
132 #else // NO_PERF ---------------------------------------------------
133 //empty class stub so perf monitoring can be disabled alltogether
134 class PerfEvent {
135     public:
136         PerfEvent(const uint64_t  p_event, const int group= -1,
137             const pid_t pid = 0, const int cpu = -1) {}
138
139         void Start() {}
140         void Stop() {}
141         uint64_t Count() {
142             return 0;
143         }
144         int Group() const {
145             return -1;
146         }
147
148         ~PerfEvent() {}
149 };
150 #endif //!NO_PERF
151 #endif //PERFLIB_H
```

Listing B.3: perflib.h

B.3 flipbit

```
1  // (C) 2015 Dieter Steiner
2  // <spoilerhead@gmail.com>
3  //
4  // flipbit flips (negates) a single bit in a file
5  // dependencies:
6  //  boost::program_options
7  //  C++11 (tested with g++ 4.8.3)
8
9  #include <fstream>
10 #include <iostream>
11 #include <iterator>
12 #include <boost/program_options.hpp>
13 #include "flipbit_config.h"
14
15 namespace po = boost::program_options;
16
17 int main(int argc, char *argv[]) {
18     // Parameters ---------------------------------------------------------
19     std::string filename_input("");
20     std::string filename_output("");
21     size_t bitnumber = 0;
22     bool verbose = false;
23
24     // Parse parameters ---------------------------------------------------
25     // Declare the supported options.
26     po::options_description desc("Allowed options");
27     desc.add_options()
28         ("help,h", "produce help message")
29         ("input,i", po::value<std::string>(), "input file")
30         ("output,o", po::value<std::string>(), "output file")
31         ("bit,b", po::value<size_t>(), "bit to modify")
32         ("verbose,v", "verbose")
```

```
33      ;
34
35      po::positional_options_description p;
36      p.add("input", 1);
37      p.add("output", 1);
38      p.add("bit", 1);
39
40      po::variables_map vm;
41      po::store(po::command_line_parser(argc, argv).
42            options(desc).positional(p).run(), vm);
43      po::notify(vm);
44
45      if((vm.count("help")) || argc == 1) {
46          std::cout << "flipbit "
47                  << FLIPBIT_VERSION_MAJOR << "."
48                  << FLIPBIT_VERSION_MINOR << "."
49                  << FLIPBIT_VERSION_PATCH
50                  << std::endl;
51          std::cout << "Flips a single bit in a file" << std::endl;
52          std::cout << desc << std::endl;
53          return 1;
54      }
55      if(vm.count("verbose")) {
56          verbose = true;
57      }
58
59      if(vm.count("input")) {
60          filename_input = vm["input"].as<std::string>();
61      } else {
62          std::cerr << "needs input file" << std::endl;
63          return -2;
64      }
65      if(vm.count("output")) {
66          filename_output = vm["output"].as<std::string>();
67      } else {
68          std::cerr << "needs output file" << std::endl;
69          return -3;
70      }
71
72      //options
73      if(vm.count("bit")) {
74          bitnumber = vm["bit"].as<size_t>();
75      } else {
76          std::cerr << "needs bit number" << std::endl;
77          return -4;
78      }
79      // done parsing options ------------------------------------------------------
80      // compute the needed byte and the bit within the byte
81      const size_t byte_to_modify = bitnumber / 8;
82      const size_t bit_to_modify = bitnumber % 8;
83      const uint8_t modifyMask = 1 << bit_to_modify;
84
85      // Print Verbose for diagnostic options (if requested)
86      if(verbose) {
87          std::cout << "Input File:  " << filename_input << std::endl;
88          std::cout << "Output File: " << filename_output << std::endl;
89          std::cout << "Bit to flip: " << bitnumber << std::endl;
90          std::cout << "Byte to mod: " << byte_to_modify << std::endl;
91          std::cout << "bit to mod:  " << bit_to_modify << std::endl;
92          std::cout << "modifymask:  " << (int)modifyMask <<  std::endl;
93      }
```

```
94
95      // actual code ------------------------------------------------------------
96      std::ifstream infile;
97      std::ofstream outfile;
98
99      // open filestreams
100     infile.open(filename_input.c_str(), std::ios::in | std::ios::binary);
101     outfile.open(filename_output.c_str(), std::ios::out | std::ios::binary);
102
103     if(infile.is_open() && outfile.is_open()) {
104         // get iterators on the filestreams
105         std::istreambuf_iterator<char> inIter(infile);
106         std::ostreambuf_iterator<char> outIter(outfile);
107         std::istreambuf_iterator<char> endIter; //EOF
108
109         size_t byteCount = 0;
110         // while not EOF
111         while(inIter != endIter) {
112             uint8_t curbyte = *inIter;  // read a byte
113             // if it is the byte we want, XOR it with the modify mask
114             if(byteCount == byte_to_modify) {
115                 curbyte ^= modifyMask;
116             }
117             *outIter = curbyte;         // write back the file
118             byteCount++;                // increment byte count
119             ++inIter; ++outIter;        // increment iterators
120         }
121         //not enough bits
122         if(byteCount <= byte_to_modify) {
123             if(verbose) {
124                 std::cout << "WARNING: Less bits in file than needed"
125                     << std::endl;
126             }
127         }
128
129         // Close files
130         infile.close();
131         outfile.close();
132     }
133     return 0;
134 }
```

Listing B.4: flipbit.cpp

B.4 Benchmarks

```
1 void nopfcn(void) {
2     asm("nop");
3 }
```

Listing B.5: nop.c

```
1 #define doNOP10 \
2     asm("nop"); asm("nop"); asm("nop"); asm("nop"); asm("nop");\
3     asm("nop"); asm("nop"); asm("nop"); asm("nop"); asm("nop")
4
5 #define doNOP100 \
6         doNOP10; doNOP10; doNOP10; doNOP10; \
7         doNOP10; doNOP10; doNOP10; doNOP10; doNOP10
```

```
8
9  void nopfcn(void) {
10     doNOP100;
11 }
```

Listing B.6: long_nop.c

```
1  /*
2      bubble sort algorithm
3      http://rosettacode.org/wiki/Sorting_algorithms/Bubble_sort#C
4      2015 - Dieter Steiner
5      e0326004@student.tuwien.ac.at
6  */
7  #include "bubblesort.h"
8
9  void bubblesort(int data[], const int size) {
10 /*    int i, t, hasChanged = 1;
11     while (hasChanged) {
12         hasChanged = 0;
13         for (i = 1; i < size; i++) {
14             if (data[i] < data[i - 1]) {
15                 t = data[i];
16                 data[i] = data[i - 1];
17                 data[i - 1] = t;
18                 hasChanged = 1;
19             }
20         }
21     }
22 }*/
23 //http://www.algorithmist.com/index.php/Bubble_sort#Optimizations - bubblesort4
24 //+ fixed to add earely abort (as describes in pseudo code but not in the C
     version)
25     int bound = size - 1;
26     //for( int i = 0 ; i < size - 2b ; i++ ) {
27     while(bound > 0) {
28         int newbound = 0;
29         for(int j = 0 ; j < bound ; j++ ) {
30             if( data[j] > data[j + 1] ) {
31                 int temp = data[j];
32                 data[j] = data[j + 1];
33                 data[j + 1] = temp;
34                 newbound = j;
35             }
36         }
37         bound = newbound;
38     }
39 }
```

Listing B.7: bubblesort.c

```
1  /*
2      Single Path version of the bubble sort algorithm
3      2015 - Dieter Steiner
4      e0326004@student.tuwien.ac.at
5  */
6  #include "bubblesort.h"
7
8  void bubblesort(int data[], const int size) {
9      for(int i = (size - 1); i > 0; i--) {
10         for(int j = 1; j <= i; j++) {
```

```
            /*if(data[j-1] > data[j]) swap(data[j-1], data[j]);*/
            int s = data[j - 1];
            int t = data[j];

            if(s <= t) data[j - 1] = s;
            if(s <= t) data[j] = t;

            if(s > t) data[j - 1] = t;
            if(s > t) data[j] = s;
        }
    }
}
```

Listing B.8: bubblesort_wcet.c

```
/*
    heapsort
    Adapted from http://rosettacode.org/wiki/Sorting_algorithms/Heapsort#C
    2015 - Dieter Steiner
    e0326004@student.tuwien.ac.at
    *
*/
#include "heapsort.h"

#define ValType int
#define IS_LESS(v1, v2)  (v1 < v2)

void siftDown( ValType *a, int start, int count);
#define SWAP(r,s)  do{ValType t=r; r=s; s=t; } while(0)

void heapsort(int data[], const int size) {
    /* heapify */
    for(int start = (size-2)>>1/*/2*/; start >=0; start--) {
        siftDown( data, start, size);
    }

    for(int end=size-1; end > 0; end--) {
        SWAP(data[end],data[0]);
        siftDown(data, 0, end);
    }
}

void siftDown( ValType *a, int start, int end) {
    int root = start;
    while ( root*2+1 < end ) {
        int child = 2*root + 1;
        if ((child + 1 < end) && IS_LESS(a[child],a[child+1])) {
            child += 1;
        }
        if (IS_LESS(a[root], a[child])) {
            SWAP( a[child], a[root] );
            root = child;
        } else
            return;
    }
}
```

Listing B.9: heapsort.c

```
/*
```

```
2     quicksort, recursive version
3     Adapted from http://rosettacode.org/wiki/Sorting_algorithms/Quicksort#C
4     2015 - Dieter Steiner
5     e0326004@student.tuwien.ac.at
6     *
7     * Added Median of 3 optimization
8 */
9 #include "quicksort.h"
10 #ifndef max
11     #define max(a,b) ((a) > (b) ? (a) : (b))
12 #endif
13 #ifndef min
14     #define min(a,b) ((a) < (b) ? (a) : (b))
15 #endif
16
17 void quicksort(int data[], const int size) {
18     int i, j, p, t;
19     if (size < 2)
20         return;
21     //use median of three as pievot
22     const int a = data[0];          //first
23     const int b = data[size >> 1];  //middle
24     const int c = data[size - 1];   //last
25     p = max(min(a,b), min(max(a,b),c));
26
27     for (i = 0, j = size - 1;/*we use break;*/; i++, j--) {
28         while (data[i] < p)
29             i++;
30         while (p < data[j])
31             j--;
32         if (i >= j)
33             break;
34         t = data[i];
35         data[i] = data[j];
36         data[j] = t;
37     }
38     quicksort(data, i);
39     quicksort(data + i, size - i);
40 }
```

Listing B.10: quicksort.c

```
1 /*
2     SelectionSort
3     adapted from https://thilinasameera.wordpress.com/2011/06/01/sorting-
    algorithms-sample-codes-on-java-c-and-matlab/
4     An implementation following Sedgewicks version in Algorithms in C
5     2015 - Dieter Steiner
6     e0326004@student.tuwien.ac.at
7 */
8 #include "selectionsort.h"
9
10 void selectionsort(int data[], const int size) {
11     for(int i = 0; i < size; ++i) {
12         int idx_min = i;
13         int data_cur = data[i];
14         int data_min = data[i];
15
16         //find minimum
17         for(int k = i+1; k < size; ++k) {
18             if(data_min> data[k]) {
```

```
19                 idx_min = k;
20                 data_min = data[k];
21             }
22         }
23         //swap minimum to the front of current subset
24         data[i] = data_min;
25         data[idx_min] = data_cur;
26     }
27 }
```

Listing B.11: selectionsort.c

```
1  /*
2      Shellsort
3      adapted from http://www.cs.princeton.edu/~rs/shell/shell.c - Robert
     Sedgewick
4      2015 - Dieter Steiner
5      e0326004@student.tuwien.ac.at
6  */
7  #include "shellsort.h"
8
9  void shellsort(int data[], const int size) {
10     const int incs[16] = { 1391376, 463792, 198768, 86961, 33936,
11                            13776, 4592, 1968, 861, 336,
12                            112, 48, 21, 7, 3, 1 };
13     for(int k = 0; k < 16; ++k) {
14         const int h = incs[k];
15         for(int i = h; i < size; ++i) {
16             int v = data[i];
17             int j = i;
18             while((j >= h) && (data[j-h] > v)) {
19                 data[j] = data[j-h]; j -= h;
20             }
21             data[j] = v;
22         }
23     }
24 }
```

Listing B.12: shellsort.c

```
1  /*************************************************************************/
2  /*                                                                       */
3  /*   SNU-RT Benchmark Suite for Worst Case Timing Analysis               */
4  /*   =====================================================               */
5  /*                                Collected and Modified by S.-S. Lim     */
6  /*                                          sslim@archi.snu.ac.kr        */
7  /*                                       Real-Time Research Group        */
8  /*                                      Seoul National University        */
9  /*                                                                       */
10 /*                                                                       */
11 /*        < Features > - restrictions for our experimental environment   */
12 /*                                                                       */
13 /*          1. Completely structured.                                    */
14 /*               - There are no unconditional jumps.                     */
15 /*               - There are no exit from loop bodies.                   */
16 /*                 (There are no 'break' or 'return' in loop bodies)     */
17 /*          2. No 'switch' statements.                                   */
18 /*          3. No 'do..while' statements.                                */
19 /*          4. Expressions are restricted.                               */
20 /*               - There are no multiple expressions joined by 'or',     */
```

```
/*                'and' operations.                                        */
/*          5. No library calls.                                           */
/*               - All the functions needed are implemented in the         */
/*                 source file.                                            */
/*                                                                         */
/*                                                                         */
/***************************************************************************/
/*                                                                         */
/*  FILE: bs.c                                                             */
/*  SOURCE : Public Domain Code                                            */
/*                                                                         */
/*  DESCRIPTION :                                                          */
/*                                                                         */
/*     Binary search for the array of 15 integer elements.                 */
/*                                                                         */
/*  REMARK :                                                               */
/*                                                                         */
/*  EXECUTION TIME :                                                       */
/*                                                                         */
/*                                                                         */
/***************************************************************************/

/*
    Modified for use in my thesis
    2015 - Dieter Steiner
    e0326004@student.tuwien.ac.at
    *
*/

#include "binsearch.h"

/* returns value @ key*/
int binary_search(struct DATA data[], const int len, const int key) {
  int fvalue, mid, up, low ;

  low = 0;
  up = len - 1;
  fvalue = -1 /* all data are positive */ ;
  while (low <= up) {
    mid = (low + up) >> 1;
    if ( data[mid].key == key ) {  /*  found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else  /* not found */
      if ( data[mid].key > key )    {
        up = mid - 1;
      }
      else   {
        low = mid + 1;
      }
  }
  return fvalue;
}
```

Listing B.13: binsearch.c

```
/***************************************************************************/
/*                                                                         */
/*   SNU-RT Benchmark Suite for Worst Case Timing Analysis                 */
/*   =====================================================                 */
```

```
5  /*                              Collected and Modified by S.-S. Lim      */
6  /*                                        sslim@archi.snu.ac.kr          */
7  /*                                    Real-Time Research Group           */
8  /*                                   Seoul National University           */
9  /*                                                                       */
10 /*                                                                       */
11 /*        < Features > - restrictions for our experimental environment   */
12 /*                                                                       */
13 /*          1. Completely structured.                                    */
14 /*               - There are no unconditional jumps.                     */
15 /*               - There are no exit from loop bodies.                   */
16 /*                 (There are no 'break' or 'return' in loop bodies)     */
17 /*          2. No 'switch' statements.                                   */
18 /*          3. No 'do..while' statements.                                */
19 /*          4. Expressions are restricted.                               */
20 /*               - There are no multiple expressions joined by 'or',     */
21 /*                 'and' operations.                                     */
22 /*          5. No library calls.                                         */
23 /*               - All the functions needed are implemented in the       */
24 /*                 source file.                                          */
25 /*                                                                       */
26 /*                                                                       */
27 /*************************************************************************/
28 /*                                                                       */
29 /*  FILE: bs.c                                                           */
30 /*  SOURCE : Public Domain Code                                          */
31 /*                                                                       */
32 /*  DESCRIPTION :                                                        */
33 /*                                                                       */
34 /*     Binary search for the array of 15 integer elements.               */
35 /*                                                                       */
36 /*  REMARK :                                                             */
37 /*                                                                       */
38 /*  EXECUTION TIME :                                                     */
39 /*                                                                       */
40 /*                                                                       */
41 /*************************************************************************/
42
43 /*
44     Modified for use in my thesis
45     2015 - Dieter Steiner
46     e0326004@student.tuwien.ac.at
47     *
48     * WCETified
49 */
50
51 #include "binsearch_wcet.h"
52
53 /* returns value @ key*/
54 int binary_search(struct DATA data[], const int len, const int key) {
55   int fvalue, up, low ;
56
57   low = 0;
58   up = len - 1;
59   fvalue = -1 /* all data are positive */ ;
60   while (low <= up) {
61     const int mid = (low + up) >> 1;
62
63     //0.. lower half (data.key > key)
64     //1.. upper half (otherwise)
65     const int newup[2] = {mid - 1, up};
```

```
66      const int newlow[2] = {low, mid + 1};
67
68      if (data[mid].key == key) { //found
69        fvalue = data[mid].value;
70      }
71      /* not found, or after finding */
72      const int whichhalf = (data[mid].key >= key) ? 0 : 1;
73
74      up = newup[whichhalf];
75      low = newlow[whichhalf];
76    }
77    return fvalue;
78  }
```

Listing B.14: binsearch_wcet.c

```
1  /*--------------------------------------------------------------------
2   *  WCET Benchmark created by Jakob Engblom, Uppsala university,
3   *  February 2000.
4   *
5   *  The purpose of this benchmark is to force the compiler to emit an
6   *  unstructured loop, which is usually problematic for WCET tools to
7   *  handle.
8   *
9   *  The execution time should be constant.
10  *
11  *  The original code is "Duff's Device", see the Jargon File, e.g. at
12  *  http://www.tf.hut.fi/cgi-bin/jargon.  Created in the early 1980s
13  *  as a way to express loop unrolling in C.
14  *
15  *--------------------------------------------------------------------*/
16 /*
17     Modified for use in my thesis
18     2015 - Dieter Steiner
19     e0326004@student.tuwien.ac.at
20 */
21 #include "duff.h"
22
23 void duffcopy( char *to, char *from, const int count) {
24     int n = (count + 7) / 8;
25     switch(count % 8) {
26         case 0: do{     *to++ = *from++;
27         case 7:         *to++ = *from++;
28         case 6:         *to++ = *from++;
29         case 5:         *to++ = *from++;
30         case 4:         *to++ = *from++;
31         case 3:         *to++ = *from++;
32         case 2:         *to++ = *from++;
33         case 1:         *to++ = *from++;
34             } while(--n > 0);
35     }
36 }
```

Listing B.15: duff.c

```
1 #include <stddef.h>
2 #include "strstr.h"
3
4 // Source: http://codereview.stackexchange.com/questions/35396/strstr-
    implementation
```

```
5  // Adapted for use in my thesis by
6  // Dieter Steiner <e0326004@student.tuwien.ac.at>
7
8  char * strstr_iterative(const char *haystack, const char *needle) {
9      const char *a = haystack, *b = needle;
10     for (;;)
11         if      (!*b)          return (char *)haystack;  //end of needle, means
       it matches
12         else if (!*a)          return NULL;  //end of haystack
13         else if (*a++ != *b++) { a = ++haystack; b = needle;} //current char
       doesn't mactch
14 }
```

Listing B.16: strstr.c

```
1  /* $Id: fibcall.c,v 1.2 2005/04/04 11:34:58 csg Exp $ */
2
3  /*****************************************************************************/
4  /*                                                                           */
5  /*   SNU-RT Benchmark Suite for Worst Case Timing Analysis                   */
6  /*   =====================================================                   */
7  /*                                Collected and Modified by S.-S. Lim        */
8  /*                                          sslim@archi.snu.ac.kr            */
9  /*                                       Real-Time Research Group            */
10 /*                                      Seoul National University            */
11 /*                                                                           */
12 /*                                                                           */
13 /*        < Features > - restrictions for our experimental environment       */
14 /*                                                                           */
15 /*          1. Completely structured.                                        */
16 /*               - There are no unconditional jumps.                         */
17 /*               - There are no exit from loop bodies.                       */
18 /*                 (There are no 'break' or 'return' in loop bodies)         */
19 /*          2. No 'switch' statements.                                       */
20 /*          3. No 'do..while' statements.                                    */
21 /*          4. Expressions are restricted.                                   */
22 /*               - There are no multiple expressions joined by 'or',         */
23 /*                 'and' operations.                                         */
24 /*          5. No library calls.                                             */
25 /*               - All the functions needed are implemented in the           */
26 /*                 source file.                                              */
27 /*                                                                           */
28 /*                                                                           */
29 /*****************************************************************************/
30 /*                                                                           */
31 /*  FILE: fibcall.c                                                          */
32 /*  SOURCE : Public Domain Code                                              */
33 /*                                                                           */
34 /*  DESCRIPTION :                                                            */
35 /*                                                                           */
36 /*     Summing the Fibonacci series.                                         */
37 /*                                                                           */
38 /*  REMARK :                                                                 */
39 /*                                                                           */
40 /*  EXECUTION TIME :                                                         */
41 /*                                                                           */
42 /*                                                                           */
43 /*****************************************************************************/
44
45 /*
46     Modified for use in my thesis
```

```
47      2015 - Dieter Steiner
48      e0326004@student.tuwien.ac.at
49  */
50
51  #include "fibcall.h"
52
53  uint64_t fib(const int n) {
54    uint64_t i, Fnew, Fold, temp,ans;
55
56      Fnew = 1;  Fold = 0;
57      for ( i = 2;
58        //Fib_93 is the largest 64bit fibonacci number
59        i <= 93 && i <= n;          /* apsim_loop 1 0 */
60        i++ )
61      {
62        temp = Fnew;
63        Fnew = Fnew + Fold;
64        Fold = temp;
65      }
66      ans = Fnew;
67    return ans;
68  }
```

Listing B.17: fibcall.c

```
1  // Source: http://www.mrtc.mdh.se/projects/wcet/wcet_bench/prime/prime.c
2  // Adapted for use in my thesis by
3  // Dieter Steiner <e0326004@student.tuwien.ac.at>
4  //
5  // - removed unneeded functions
6  // - Added loop check && ((i * i) > i) - otherwise i*i might overflow (e.g. when
       computing 2^32-1)
7
8  #include "prime.h"
9
10 static inline int divides(const uint32_t n, const uint32_t m) {
11   return (m % n == 0);
12 }
13
14 static inline int even(const uint32_t n) {
15   return (divides (2, n));
16 }
17
18 int prime(const uint32_t n) {
19   if(even (n))
20       return (n == 2);
21   for(uint32_t i = 3; ((i * i) <= n) && ((i * i) > i); i += 2) {
22       if(divides (i, n))
23           return 0;
24   }
25   return (n > 1);
26 }
```

Listing B.18: prime.c

```
1  // Source: http://www.mrtc.mdh.se/projects/wcet/wcet_bench/prime/prime.c
2  // Adapted for use in my thesis by
3  // Dieter Steiner <e0326004@student.tuwien.ac.at>
4  //
5  // - removed unneeded functions
```

```
6  // - Added loop check && ((i * i) > i) - otherwise i*i might overflow (e.g. when
       computing 2^32-1)
7  // - optimized for wcet (no early abort)
8  #include "prime.h"
9
10 //wcet optimized modulo,
11 //runtime of modulo is value dependent on arm7
12 static inline int UImod(const uint32_t a, const uint32_t b) {
13     uint32_t s = b; //need to have that extra bit
14
15     //scale up s
16     for(uint32_t i = 0; i<32; ++i) {
17         uint32_t s_new = s <<= 1;
18         if (s <= a) {
19             s = s_new;
20         }
21     }
22
23     //fit in
24     uint32_t r = a;
25     for(uint32_t i = 0; i<32; ++i) {
26         uint32_t r_new = r;
27         s >>= 1;
28         r_new = r - s;
29         if((r >= b) && (s <= r) ) {
30             r = r_new;
31         }
32     }
33     return r;
34 }
35
36 static int divides(const uint32_t n, const uint32_t m) {
37     return (UImod(m, n) == 0);
38 }
39
40 static inline int even(const uint32_t n) {
41     return (divides (2, n));
42 }
43
44 int prime(const uint32_t n) {
45     //0,1 are no primes
46     int isprime = (n > 1);
47     //even numbers only prime if 2
48     isprime &= !(even(n) && !(n == 2));
49
50     for(uint32_t i = 3; i <= 0xFFFF; i += 2) {
51         //only prime if not divisible by other numbers than itself
52         isprime &= (!(divides (i, n)) || (i == n));
53     }
54     return isprime;
55 }
```

Listing B.19: prime_wcet.c

```
1  /* MDH WCET BENCHMARK SUITE. */
2
3  /* 2012/09/28, Jan Gustafsson <jan.gustafsson@mdh.se>
4   * Changes:
5   *  - This program redefines the C standard function sqrt. Therefore, this
     function has been renamed to sqrtfcn.
```

```
 *  - qrt.c:79:15: warning: explicitly assigning a variable of type 'float' to
  itself: fixed
 */

/*****************************************************************************/
/*                                                                           */
/*   SNU-RT Benchmark Suite for Worst Case Timing Analysis                   */
/*   =====================================================                   */
/*                              Collected and Modified by S.-S. Lim           */
/*                                                sslim@archi.snu.ac.kr      */
/*                                           Real-Time Research Group        */
/*                                          Seoul National University        */
/*                                                                           */
/*                                                                           */
/*        < Features > - restrictions for our experimental environment       */
/*                                                                           */
/*          1. Completely structured.                                        */
/*               - There are no unconditional jumps.                         */
/*               - There are no exit from loop bodies.                       */
/*                 (There are no 'break' or 'return' in loop bodies)         */
/*          2. No 'switch' statements.                                       */
/*          3. No 'do..while' statements.                                    */
/*          4. Expressions are restricted.                                   */
/*               - There are no multiple expressions joined by 'or',         */
/*                 'and' operations.                                         */
/*          5. No library calls.                                             */
/*               - All the functions needed are implemented in the           */
/*                 source file.                                              */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*  FILE: sqrt.c                                                            */
/*  SOURCE : Public Domain Code                                              */
/*                                                                           */
/*  DESCRIPTION :                                                            */
/*                                                                           */
/*     Square root function implemented by Taylor series.                    */
/*                                                                           */
/*  REMARK :                                                                 */
/*                                                                           */
/*  EXECUTION TIME :                                                         */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/

/*
    Modified for use in my thesis
    2015 - Dieter Steiner
    e0326004@student.tuwien.ac.at
*/
#include "sqrt.h"

static float f_abs(float x) {
   if (x < 0)
      return -x;
   else
      return x;
}

float sqrtfcn(const float val) {
```

```
66    float x = val/10.f;
67    int flag = 0;
68
69    if(val == 0.f ) {
70       x = 0.f;
71    } else {
72       for(int i=1; i<20; ++i) {
73          if (!flag) {
74             x += (val - (x * x)) / (2.0f * x);
75             const float diff = val - (x * x);
76             if (f_abs(diff) <= SQRT_MIN_TOL) {
77                flag = 1;
78             }
79          } else {} /* JG */
80 /*             x =x; */
81       }
82    }
83    return (x);
84 }
```

Listing B.20: sqrt_micro.c

```
1  /*
2      Square root computation - WCET optimized version
3      2015 - Dieter Steiner
4      e0326004@student.tuwien.ac.at
5      *
6  */
7  #include "sqrt.h"
8
9  //fully wcet optimized
10 float sqrtfcn(const float val) {
11     float x = val/10.f; //initial guess
12
13     for(int i=0; i<19; ++i) {
14          //x = (val + (x * x)) / (2.0f * x);
15          x = 0.5f * ((val / x) + x);
16     }
17     if(val == 0.f) {
18         return 0.f;
19     } else {
20          return x;
21     }
22 }
```

Listing B.21: sqrt_micro_wcet.c

```
1  /*
2      Square root computation - WCET optimized version
3      2015 - Dieter Steiner
4      e0326004@student.tuwien.ac.at
5      *
6      * Added early abort for negative numbers
7  */
8  #include "sqrt.h"
9
10 //fully wcet optimized
11 float sqrtfcn(const float val) {
12     if(val < 0.f) return -1; //no square roots of negative numbers
13     float x = val/10.f; //initial guess
```

```
14
15      for(int i=0; i<19; ++i) {
16          //x = (val + (x * x)) / (2.0f * x);
17          x = 0.5f * ((val / x) + x);
18      }
19      if(val == 0.f) {
20          return 0.f;
21      } else {
22          return x;
23      }
24  }
```

Listing B.22: sqrt_micro_wcet_neg.c

```
1  #include "matmul.h"
2
3  // Source: http://www.mrtc.mdh.se/projects/wcet/wcet_bench/matmult/matmult.c
4  // Adapted for use in my thesis by
5  // Dieter Steiner <e0326004@student.tuwien.ac.at>
6  //
7  // - removed unneeded functions
8  // - Updated to modern coding standards
9
10
11 // Multiplies matrices A and B and stores the result in ResultArray.
12 void Multiply(const matrix A, const matrix B, matrix Res) {
13     for(size_t Outer = 0; Outer < UPPERLIMIT; ++Outer) {
14         for(size_t Inner = 0; Inner < UPPERLIMIT; ++Inner) {
15             Res[Outer][Inner] = 0;
16             for(size_t Index = 0; Index < UPPERLIMIT; ++Index) {
17                 Res[Outer][Inner] += A[Outer][Index] * B[Index][Inner];
18             }
19         }
20     }
21 }
```

Listing B.23: matmul.c

```
1  #include "matmul_float.h"
2
3  // Source: http://www.mrtc.mdh.se/projects/wcet/wcet_bench/matmult/matmult.c
4  // Adapted for use in my thesis by
5  // Dieter Steiner <e0326004@student.tuwien.ac.at>
6  //
7  // - removed unneeded functions
8  // - Updated to modern coding standards
9  // - Converted to float
10
11
12 // Multiplies matrices A and B and stores the result in ResultArray.
13 void MultiplyFloat(const matrix A, const matrix B, matrix Res) {
14     for(size_t Outer = 0; Outer < UPPERLIMIT; ++Outer) {
15         for(size_t Inner = 0; Inner < UPPERLIMIT; ++Inner) {
16             Res[Outer][Inner] = 0.f;
17             for(size_t Index = 0; Index < UPPERLIMIT; ++Index) {
18                 Res[Outer][Inner] += A[Outer][Index] * B[Index][Inner];
19             }
20         }
21     }
22 }
```

Listing B.24: matmul_float.c

```
/*
This is an implementation of the AES128 algorithm, specifically ECB and CBC mode.

The implementation is verified against the test vectors in:
  National Institute of Standards and Technology Special Publication 800-38A 2001 ED

ECB-AES128
----------

  plain-text:
    6bc1bee22e409f96e93d7e117393172a
    ae2d8a571e03ac9c9eb76fac45af8e51
    30c81c46a35ce411e5fbc1191a0a52ef
    f69f2445df4f9b17ad2b417be66c3710

  key:
    2b7e151628aed2a6abf7158809cf4f3c

  resulting cipher
    3ad77bb40d7a3660a89ecaf32466ef97
    f5d3d58503b9699de785895a96fdbaaf
    43b1cd7f598ece23881b00e3ed030688
    7b0c785e27e8ad3f8223207104725dd4


NOTE:   String length must be evenly divisible by 16byte (str_len % 16 == 0)
        You should pad the end of the string with zeros if this is not the case.
*/

/*****************************************************************************/
/* Includes:                                                                 */
/*****************************************************************************/
#include <stdint.h>
#include <string.h> // CBC mode, for memset
#include "aes.h"

/*****************************************************************************/
/* Defines:                                                                  */
/*****************************************************************************/
// The number of columns comprising a state in AES. This is a constant in AES. Value=4
#define Nb 4
// The number of 32 bit words in a key.
#define Nk 4
// Key length in bytes [128 bit]
#define KEYLEN 16
// The number of rounds in AES Cipher.
#define Nr 10

// jcallan@github points out that declaring Multiply as a function
// reduces code size considerably with the Keil ARM compiler.
// See this link for more information: https://github.com/kokke/tiny-AES128-C/pull/3
#ifndef MULTIPLY_AS_A_FUNCTION
  #define MULTIPLY_AS_A_FUNCTION 0
```

```
54 #endif
55
56 /*****************************************************************************/
57 /* Private variables:                                                        */
58 /*****************************************************************************/
59 // state - array holding the intermediate results during decryption.
60 typedef uint8_t state_t[4][4];
61 static state_t* state;
62
63 // The array that stores the round keys.
64 static uint8_t RoundKey[176];
65
66 // The Key input to the AES Program
67 static const uint8_t* Key;
68
69 #if defined(CBC) && CBC
70   // Initial Vector used only for CBC mode
71   static uint8_t* Iv;
72 #endif
73
74 // The lookup-tables are marked const so they can be placed in read-only storage
     instead of RAM
75 // The numbers below can be computed dynamically trading ROM for RAM -
76 // This can be useful in (embedded) bootloader applications, where ROM is often
     limited.
77 static const uint8_t sbox[256] =   {
78   //0     1    2      3     4    5     6     7      8    9     A      B    C
      D     E     F
79   0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
     0xd7, 0xab, 0x76,
80   0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
     0xa4, 0x72, 0xc0,
81   0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
     0xd8, 0x31, 0x15,
82   0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
     0x27, 0xb2, 0x75,
83   0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
     0xe3, 0x2f, 0x84,
84   0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
     0x4c, 0x58, 0xcf,
85   0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
     0x3c, 0x9f, 0xa8,
86   0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
     0xff, 0xf3, 0xd2,
87   0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
     0x5d, 0x19, 0x73,
88   0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
     0x5e, 0x0b, 0xdb,
89   0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
     0x95, 0xe4, 0x79,
90   0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
     0x7a, 0xae, 0x08,
91   0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
     0xbd, 0x8b, 0x8a,
92   0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
     0xc1, 0x1d, 0x9e,
93   0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
     0x55, 0x28, 0xdf,
94   0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
     0x54, 0xbb, 0x16 };
95
```

```
96  static const uint8_t rsbox[256] =
97  { 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81,
      0xf3, 0xd7, 0xfb,
98    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4,
      0xde, 0xe9, 0xcb,
99    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42,
      0xfa, 0xc3, 0x4e,
100   0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d,
      0x8b, 0xd1, 0x25,
101   0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d,
      0x65, 0xb6, 0x92,
102   0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7,
      0x8d, 0x9d, 0x84,
103   0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8,
      0xb3, 0x45, 0x06,
104   0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01,
      0x13, 0x8a, 0x6b,
105   0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0,
      0xb4, 0xe6, 0x73,
106   0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c,
      0x75, 0xdf, 0x6e,
107   0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa,
      0x18, 0xbe, 0x1b,
108   0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78,
      0xcd, 0x5a, 0xf4,
109   0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27,
      0x80, 0xec, 0x5f,
110   0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93,
      0xc9, 0x9c, 0xef,
111   0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83,
      0x53, 0x99, 0x61,
112   0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55,
      0x21, 0x0c, 0x7d };
113
114
115 // The round constant word array, Rcon[i], contains the values given by
116 // x to th e power (i-1) being powers of x (x is denoted as {02}) in the field
      GF(2^8)
117 // Note that i starts at 1, not 0).
118 static const uint8_t Rcon[255] = {
119   0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
      0xab, 0x4d, 0x9a,
120   0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
      0xc5, 0x91, 0x39,
121   0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
      0x83, 0x1d, 0x3a,
122   0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
      0x36, 0x6c, 0xd8,
123   0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
      0x7d, 0xfa, 0xef,
124   0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
      0x33, 0x66, 0xcc,
125   0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
      0x40, 0x80, 0x1b,
126   0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
      0x6a, 0xd4, 0xb3,
127   0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
      0x25, 0x4a, 0x94,
128   0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
      0x08, 0x10, 0x20,
```

```
129   0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
      0xc6, 0x97, 0x35,
130   0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
      0x61, 0xc2, 0x9f,
131   0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
      0x01, 0x02, 0x04,
132   0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
      0x5e, 0xbc, 0x63,
133   0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
      0xe4, 0xd3, 0xbd,
134   0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74,
      0xe8, 0xcb  };
135
136 /*****************************************************************************/
137 /* Private functions:                                                        */
138 /*****************************************************************************/
139 static uint8_t getSBoxValue(uint8_t num)
140 {
141   return sbox[num];
142 }
143
144 static uint8_t getSBoxInvert(uint8_t num)
145 {
146   return rsbox[num];
147 }
148
149 // This function produces Nb(Nr+1) round keys. The round keys are used in each
      round to decrypt the states.
150 static void KeyExpansion(void)
151 {
152   uint32_t i, j, k;
153   uint8_t tempa[4]; // Used for the column/row operations
154
155   // The first round key is the key itself.
156   for(i = 0; i < Nk; ++i)
157   {
158     RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
159     RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
160     RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
161     RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
162   }
163
164   // All other round keys are found from the previous round keys.
165   for(; (i < (Nb * (Nr + 1))); ++i)
166   {
167     for(j = 0; j < 4; ++j)
168     {
169       tempa[j]=RoundKey[(i-1) * 4 + j];
170     }
171     if (i % Nk == 0)
172     {
173       // This function rotates the 4 bytes in a word to the left once.
174       // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]
175
176       // Function RotWord()
177       {
178         k = tempa[0];
179         tempa[0] = tempa[1];
180         tempa[1] = tempa[2];
181         tempa[2] = tempa[3];
182         tempa[3] = k;
```

```
183       }
184
185       // SubWord() is a function that takes a four-byte input word and
186       // applies the S-box to each of the four bytes to produce an output word.
187
188       // Function Subword()
189       {
190         tempa[0] = getSBoxValue(tempa[0]);
191         tempa[1] = getSBoxValue(tempa[1]);
192         tempa[2] = getSBoxValue(tempa[2]);
193         tempa[3] = getSBoxValue(tempa[3]);
194       }
195
196       tempa[0] =  tempa[0] ^ Rcon[i/Nk];
197     }
198     else if (Nk > 6 && i % Nk == 4)
199     {
200       // Function Subword()
201       {
202         tempa[0] = getSBoxValue(tempa[0]);
203         tempa[1] = getSBoxValue(tempa[1]);
204         tempa[2] = getSBoxValue(tempa[2]);
205         tempa[3] = getSBoxValue(tempa[3]);
206       }
207     }
208     RoundKey[i * 4 + 0] = RoundKey[(i - Nk) * 4 + 0] ^ tempa[0];
209     RoundKey[i * 4 + 1] = RoundKey[(i - Nk) * 4 + 1] ^ tempa[1];
210     RoundKey[i * 4 + 2] = RoundKey[(i - Nk) * 4 + 2] ^ tempa[2];
211     RoundKey[i * 4 + 3] = RoundKey[(i - Nk) * 4 + 3] ^ tempa[3];
212   }
213 }
214
215 // This function adds the round key to state.
216 // The round key is added to the state by an XOR function.
217 static void AddRoundKey(uint8_t round)
218 {
219   uint8_t i,j;
220   for(i=0;i<4;++i)
221   {
222     for(j = 0; j < 4; ++j)
223     {
224       (*state)[i][j] ^= RoundKey[round * Nb * 4 + i * Nb + j];
225     }
226   }
227 }
228
229 // The SubBytes Function Substitutes the values in the
230 // state matrix with values in an S-box.
231 static void SubBytes(void)
232 {
233   uint8_t i, j;
234   for(i = 0; i < 4; ++i)
235   {
236     for(j = 0; j < 4; ++j)
237     {
238       (*state)[j][i] = getSBoxValue((*state)[j][i]);
239     }
240   }
241 }
242
243 // The ShiftRows() function shifts the rows in the state to the left.
```

```
244 // Each row is shifted with different offset.
245 // Offset = Row number. So the first row is not shifted.
246 static void ShiftRows(void)
247 {
248   uint8_t temp;
249 
250   // Rotate first row 1 columns to left
251   temp           = (*state)[0][1];
252   (*state)[0][1] = (*state)[1][1];
253   (*state)[1][1] = (*state)[2][1];
254   (*state)[2][1] = (*state)[3][1];
255   (*state)[3][1] = temp;
256 
257   // Rotate second row 2 columns to left
258   temp           = (*state)[0][2];
259   (*state)[0][2] = (*state)[2][2];
260   (*state)[2][2] = temp;
261 
262   temp       = (*state)[1][2];
263   (*state)[1][2] = (*state)[3][2];
264   (*state)[3][2] = temp;
265 
266   // Rotate third row 3 columns to left
267   temp       = (*state)[0][3];
268   (*state)[0][3] = (*state)[3][3];
269   (*state)[3][3] = (*state)[2][3];
270   (*state)[2][3] = (*state)[1][3];
271   (*state)[1][3] = temp;
272 }
273 
274 static uint8_t xtime(uint8_t x)
275 {
276   return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
277 }
278 
279 // MixColumns function mixes the columns of the state matrix
280 static void MixColumns(void)
281 {
282   uint8_t i;
283   uint8_t Tmp,Tm,t;
284   for(i = 0; i < 4; ++i)
285   {
286     t   = (*state)[i][0];
287     Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3] ;
288     Tm  = (*state)[i][0] ^ (*state)[i][1] ; Tm = xtime(Tm);  (*state)[i][0] ^=
    Tm ^ Tmp ;
289     Tm  = (*state)[i][1] ^ (*state)[i][2] ; Tm = xtime(Tm);  (*state)[i][1] ^=
    Tm ^ Tmp ;
290     Tm  = (*state)[i][2] ^ (*state)[i][3] ; Tm = xtime(Tm);  (*state)[i][2] ^=
    Tm ^ Tmp ;
291     Tm  = (*state)[i][3] ^ t ;        Tm = xtime(Tm);  (*state)[i][3] ^= Tm ^
    Tmp ;
292   }
293 }
294 
295 // Multiply is used to multiply numbers in the field GF(2^8)
296 #if MULTIPLY_AS_A_FUNCTION
297 static uint8_t Multiply(uint8_t x, uint8_t y)
298 {
299   return (((y & 1) * x) ^
300        ((y>>1 & 1) * xtime(x)) ^
```

```
301         ((y>>2 & 1) * xtime(xtime(x))) ^
302         ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^
303         ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))));
304   }
305 #else
306 #define Multiply(x, y)                                \
307       (  ((y & 1) * x) ^                              \
308       ((y>>1 & 1) * xtime(x)) ^                       \
309       ((y>>2 & 1) * xtime(xtime(x))) ^                \
310       ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
311       ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))   \
312
313 #endif
314
315 // MixColumns function mixes the columns of the state matrix.
316 // The method used to multiply may be difficult to understand for the
    inexperienced.
317 // Please use the references to gain more information.
318 static void InvMixColumns(void)
319 {
320   int i;
321   uint8_t a,b,c,d;
322   for(i=0;i<4;++i)
323   {
324     a = (*state)[i][0];
325     b = (*state)[i][1];
326     c = (*state)[i][2];
327     d = (*state)[i][3];
328
329     (*state)[i][0] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^
     Multiply(d, 0x09);
330     (*state)[i][1] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^
     Multiply(d, 0x0d);
331     (*state)[i][2] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^
     Multiply(d, 0x0b);
332     (*state)[i][3] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^
     Multiply(d, 0x0e);
333   }
334 }
335
336 // The SubBytes Function Substitutes the values in the
337 // state matrix with values in an S-box.
338 static void InvSubBytes(void)
339 {
340   uint8_t i,j;
341   for(i=0;i<4;++i)
342   {
343     for(j=0;j<4;++j)
344     {
345       (*state)[j][i] = getSBoxInvert((*state)[j][i]);
346     }
347   }
348 }
349
350 static void InvShiftRows(void)
351 {
352   uint8_t temp;
353
354   // Rotate first row 1 columns to right
355   temp=(*state)[3][1];
356   (*state)[3][1]=(*state)[2][1];
```

```
357   (*state)[2][1]=(*state)[1][1];
358   (*state)[1][1]=(*state)[0][1];
359   (*state)[0][1]=temp;
360
361   // Rotate second row 2 columns to right
362   temp=(*state)[0][2];
363   (*state)[0][2]=(*state)[2][2];
364   (*state)[2][2]=temp;
365
366   temp=(*state)[1][2];
367   (*state)[1][2]=(*state)[3][2];
368   (*state)[3][2]=temp;
369
370   // Rotate third row 3 columns to right
371   temp=(*state)[0][3];
372   (*state)[0][3]=(*state)[1][3];
373   (*state)[1][3]=(*state)[2][3];
374   (*state)[2][3]=(*state)[3][3];
375   (*state)[3][3]=temp;
376 }
377
378 // Cipher is the main function that encrypts the PlainText.
379 static void Cipher(void)
380 {
381   uint8_t round = 0;
382
383   // Add the First round key to the state before starting the rounds.
384   AddRoundKey(0);
385
386   // There will be Nr rounds.
387   // The first Nr-1 rounds are identical.
388   // These Nr-1 rounds are executed in the loop below.
389   for(round = 1; round < Nr; ++round)
390   {
391     SubBytes();
392     ShiftRows();
393     MixColumns();
394     AddRoundKey(round);
395   }
396
397   // The last round is given below.
398   // The MixColumns function is not here in the last round.
399   SubBytes();
400   ShiftRows();
401   AddRoundKey(Nr);
402 }
403
404 static void InvCipher(void)
405 {
406   uint8_t round=0;
407
408   // Add the First round key to the state before starting the rounds.
409   AddRoundKey(Nr);
410
411   // There will be Nr rounds.
412   // The first Nr-1 rounds are identical.
413   // These Nr-1 rounds are executed in the loop below.
414   for(round=Nr-1;round>0;round--)
415   {
416     InvShiftRows();
417     InvSubBytes();
```

```
418     AddRoundKey(round);
419     InvMixColumns();
420   }
421
422   // The last round is given below.
423   // The MixColumns function is not here in the last round.
424   InvShiftRows();
425   InvSubBytes();
426   AddRoundKey(0);
427 }
428
429 static void BlockCopy(uint8_t* output, uint8_t* input)
430 {
431   uint8_t i;
432   for (i=0;i<KEYLEN;++i)
433   {
434     output[i] = input[i];
435   }
436 }
437
438 /*****************************************************************************/
439 /* Public functions:                                                         */
440 /*****************************************************************************/
441 #if defined(ECB) && ECB
442
443 void AES128_ECB_encrypt(uint8_t* input, const uint8_t* key, uint8_t* output)
444 {
445   // Copy input to output, and work in-memory on output
446   BlockCopy(output, input);
447   state = (state_t*)output;
448
449   Key = key;
450   KeyExpansion();
451
452   // The next function call encrypts the PlainText with the Key using AES
      algorithm.
453   Cipher();
454 }
455
456 void AES128_ECB_decrypt(uint8_t* input, const uint8_t* key, uint8_t *output)
457 {
458   // Copy input to output, and work in-memory on output
459   BlockCopy(output, input);
460   state = (state_t*)output;
461
462   // The KeyExpansion routine must be called before encryption.
463   Key = key;
464   KeyExpansion();
465
466   InvCipher();
467 }
468 #endif // #if defined(ECB) && ECB
469
470 #if defined(CBC) && CBC
471 static void XorWithIv(uint8_t* buf)
472 {
473   uint8_t i;
474   for(i = 0; i < KEYLEN; ++i)
475   {
476     buf[i] ^= Iv[i];
477   }
```

```
478 }
479
480 void AES128_CBC_encrypt_buffer(uint8_t* output, uint8_t* input, uint32_t length,
       const uint8_t* key, const uint8_t* iv)
481 {
482   uintptr_t i;
483   uint8_t remainders = length % KEYLEN; /* Remaining bytes in the last non-full
      block */
484
485   BlockCopy(output, input);
486   state = (state_t*)output;
487
488   // Skip the key expansion if key is passed as 0
489   if(0 != key)
490   {
491     Key = key;
492     KeyExpansion();
493   }
494
495   if(iv != 0)
496   {
497     Iv = (uint8_t*)iv;
498   }
499
500   for(i = 0; i < length; i += KEYLEN)
501   {
502     XorWithIv(input);
503     BlockCopy(output, input);
504     state = (state_t*)output;
505     Cipher();
506     Iv = output;
507     input += KEYLEN;
508     output += KEYLEN;
509   }
510
511   if(remainders)
512   {
513     BlockCopy(output, input);
514     memset(output + remainders, 0, KEYLEN - remainders); /* add 0-padding */
515     state = (state_t*)output;
516     Cipher();
517   }
518 }
519
520 void AES128_CBC_decrypt_buffer(uint8_t* output, uint8_t* input, uint32_t length,
       const uint8_t* key, const uint8_t* iv)
521 {
522   uintptr_t i;
523   uint8_t remainders = length % KEYLEN; /* Remaining bytes in the last non-full
      block */
524
525   BlockCopy(output, input);
526   state = (state_t*)output;
527
528   // Skip the key expansion if key is passed as 0
529   if(0 != key)
530   {
531     Key = key;
532     KeyExpansion();
533   }
534
```

```
535   // If iv is passed as 0, we continue to encrypt without re-setting the Iv
536   if(iv != 0)
537   {
538     Iv = (uint8_t*)iv;
539   }
540
541   for(i = 0; i < length; i += KEYLEN)
542   {
543     BlockCopy(output, input);
544     state = (state_t*)output;
545     InvCipher();
546     XorWithIv(output);
547     Iv = input;
548     input += KEYLEN;
549     output += KEYLEN;
550   }
551
552   if(remainders)
553   {
554     BlockCopy(output, input);
555     memset(output+remainders, 0, KEYLEN - remainders); /* add 0-padding */
556     state = (state_t*)output;
557     InvCipher();
558   }
559 }
560 #endif // #if defined(CBC) && CBC
```

Listing B.25: aes.c

```
1  #include "crc32.h"
2
3  // Source: http://www.hackersdelight.org/hdcodetxt/crc.c.txt
4  // Adapted for use in my thesis by
5  // Dieter Steiner <e0326004@student.tuwien.ac.at>
6  //
7  // const correctness
8  // data types
9
10 // ----------------------------- crc32b -------------------------------
11
12 /* This is the basic CRC-32 calculation with some optimization but no
13 table lookup. The the byte reversal is avoided by shifting the crc reg
14 right instead of left and by using a reversed 32-bit word to represent
15 the polynomial.
16    When compiled to Cyclops with GCC, this function executes in 8 + 72n
17 instructions, where n is the number of bytes in the input message. It
18 should be doable in 4 + 61n instructions.
19    If the inner loop is strung out (approx. 5*8 = 40 instructions),
20 it would take about 6 + 46n instructions. */
21
22 uint32_t rc_crc32(uint32_t crc, const char *buf, const size_t size) {
23    crc = ~crc;
24    //const uint8_t *p = buf;
25    for(size_t i = 0; i < size; ++i) {
26       crc ^= buf[i];            // Get next byte.;
27
28       for(size_t j = 0; j < 8; ++j) {    // Do eight times.
29          const uint32_t mask = -(crc & 1);
30          crc = (crc >> 1) ^ (0xEDB88320 & mask);
31       }
32    }
```

```
33     return ~crc;
34 }
```

Listing B.26: crc32.c

```
1  #include <math.h>
2  #include "fft.h"
3
4  // Source: http://rosettacode.org/wiki/Fast_Fourier_transform#C
5  // Algorithm: https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm
6  // Adapted for use in my thesis by
7  // Dieter Steiner <e0326004@student.tuwien.ac.at>
8  //
9  // - Added the "fsign" parameter, so the iFFT can also be computed
10
11 double PI = (M_PI);
12
13 void _fft(cplx buf[], cplx out[], int n, int step, double fsign)
14 {
15     if (step < n) {
16         _fft(out, buf, n, step * 2, fsign);
17         _fft(out + step, buf + step, n, step * 2, fsign);
18
19         for (int i = 0; i < n; i += 2 * step) {
20             cplx t = cexp(fsign* -I * PI * i / n) * out[i + step];
21             buf[i / 2]     = out[i] + t;
22             buf[(i + n)/2] = out[i] - t;
23         }
24     }
25 }
26
27 void fft(cplx buf[], int n, double fsign)
28 {
29     cplx out[n];
30     for (int i = 0; i < n; i++) out[i] = buf[i];
31
32     _fft(buf, out, n, 1, fsign);
33 }
```

Listing B.27: fft.c

```
1  /*
2      FIR Filter
3      adapted from http://www.mrtc.mdh.se/projects/wcet/wcet_bench/fir/fir.c
4      2015 - Dieter Steiner
5      e0326004@student.tuwien.ac.at
6      *
7      * Use int instead of long to suppress compiler warnings (and we're not
8      * TurboC++ anymore)
9      *
10 */
11 #include "fir.h"
12
13 /*****************************************************************
14 fir_filter_int - Filters int data array based on passed int coefficients.
15
16 The length of the input and output arrays are equal
17 and are allocated by the calller.
18 The length of the coefficient array is passed.
19 An integer scale factor (passed) is used to divide the accumulation result.
```

```
20
21 void fir_filter_int(int *in,int *out,int in_len,
22                                   int *coef,int coef_len,int scale)
23
24     in          integer pointer to input array
25     out         integer pointer to output array
26     in_len      length of input and output arrays
27     coef        integer pointer to coefficient array
28     coef_len    length of coeffient array
29     scale       scale factor to divide after accumulation
30
31 No return value.
32
33 ******************************************************************************/
34
35
36 void fir_filter_int(int* in, int* out, int in_len,
37                     int* coef, int coef_len,
38                     int scale)
39 {
40   int i,j,coef_len2,acc_length;
41   int acc;
42   int *in_ptr,*data_ptr,*coef_start,*coef_ptr,*in_end;
43
44   /* set up for coefficients */
45   coef_start = coef;
46   coef_len2 = (coef_len + 1) >> 1;
47
48   /* set up input data pointers */
49   in_end = in + in_len - 1;
50   in_ptr = in + coef_len2 - 1;
51
52   /* initial value of accumulation length for startup */
53   acc_length = coef_len2;
54
55   for(i = 0 ; i < in_len ; i++) {
56
57     /* set up pointer for accumulation */
58     data_ptr = in_ptr;
59     coef_ptr = coef_start;
60
61     /* do accumulation and write result with scale factor */
62
63     acc = (int)(*coef_ptr++) * (*data_ptr--);
64     for(j = 1 ; j < acc_length ; j++)
65       acc += (int)(*coef_ptr++) * (*data_ptr--);
66
67     *out++ = (int)(acc/scale);
68
69     /* check for end case */
70
71     if(in_ptr == in_end) {
72       acc_length--;       /* one shorter each time */
73       coef_start++;       /* next coefficient each time */
74     }
75
76     /* if not at end, then check for startup, add to input pointer */
77
78     else {
79       if(acc_length < coef_len) acc_length++;
80       in_ptr++;
```

```
81      }
82    }
83  }
```

Listing B.28: fir.c

B.4.1 Disassembly sqrt

This listings are the disassembled versions of listing B.20 and listing B.21. They are shown to highlight the difference WCET optimization can have on the resulting machine code.

```
1   00000000 <sqrtfcn>:
2    0: eeb50a40  vcmp.f32  s0, #0.0
3    4: eef1fa10  vmrs  APSR_nzcv, fpscr
4    8: 0ddf7a1a  vldreq  s15, [pc, #104] ; 78 <sqrtfcn+0x78>
5    c: 0a000017  beq 70 <sqrtfcn+0x70>
6   10: eef27a04  vmov.f32  s15, #36  ; 0x24
7   14: e3a03013  mov r3, #19
8   18: eddf6a17  vldr  s13, [pc, #92]  ; 7c <sqrtfcn+0x7c>
9   1c: e3a02000  mov r2, #0
10  20: eec07a27  vdiv.f32  s15, s0, s15
11  24: e3520000  cmp r2, #0
12  28: 13a02001  movne r2, #1
13  2c: 1a00000d  bne 68 <sqrtfcn+0x68>
14  30: eeb07a40  vmov.f32  s14, s0
15  34: ee376aa7  vadd.f32  s12, s15, s15
16  38: eea77ae7  vfms.f32  s14, s15, s15
17  3c: ee877a06  vdiv.f32  s14, s14, s12
18  40: ee777a87  vadd.f32  s15, s15, s14
19  44: eeb07a40  vmov.f32  s14, s0
20  48: eea77ae7  vfms.f32  s14, s15, s15
21  4c: eeb57ac0  vcmpe.f32 s14, #0.0
22  50: eef1fa10  vmrs  APSR_nzcv, fpscr
23  54: 4eb17a47  vnegmi.f32  s14, s14
24  58: eeb47ae6  vcmpe.f32 s14, s13
25  5c: eef1fa10  vmrs  APSR_nzcv, fpscr
26  60: 93a02001  movls r2, #1
27  64: 83a02000  movhi r2, #0
28  68: e2533001  subs  r3, r3, #1
29  6c: 1afffec  bne 24 <sqrtfcn+0x24>
30  70: eeb00a67  vmov.f32  s0, s15
31  74: e12fff1e  bx  lr
32  78: 00000000  .word 0x00000000
33  7c: 38d1b717  .word 0x38d1b717
```

Listing B.29: sqrt_micro.s

```
1  00000000 <sqrtfcn>:
2     0: eef27a04  vmov.f32  s15, #36  ; 0x24
3     4: e3a03013  mov r3, #19
4     8: eeb67a00  vmov.f32  s14, #96  ; 0x60
5     c: eec07a27  vdiv.f32  s15, s0, s15
6    10: e2533001  subs  r3, r3, #1
7    14: eec06a27  vdiv.f32  s13, s0, s15
8    18: ee767aa7  vadd.f32  s15, s13, s15
9    1c: ee677a87  vmul.f32  s15, s15, s14
10   20: 1affffa  bne 10 <sqrtfcn+0x10>
11   24: eeb50a40  vcmp.f32  s0, #0.0
12   28: ed9f0a02  vldr  s0, [pc, #8]  ; 38 <sqrtfcn+0x38>
```

```
13  2c: eef1fa10  vmrs  APSR_nzcv, fpscr
14  30: 1eb00a67  vmovne.f32  s0, s15
15  34: e12fff1e  bx  lr
16  38: 00000000  .word 0x00000
```

Listing B.30: sqrt_micro_wcet.s

List of Figures

List of Tables

Listings

Glossary

API Application Programming Interface — the source code level interface of a library/program. 20, 57

COTS Component of the Shelf 25

FPU Floating Point Unit 26, 29, 36

ioctl input/output control, a syscall for configuring devices and special files in unixoid operating systems 17

Linux Linux operating system - `http://www.linux.org/` 17–20, 28, 29, 60

NEON ARM's Advances SIMD extension, also known as Media Processing Engine. Provides instructions for acceleration of media and signal processing applications. 26, 36

RAII resource acquisition is initialization, a programming idiom, holding a resource is tied to object lifetime. As long as objects are destroyed properly, no resource–/ memory leaks will occur. 20

SIMD Single Instruction Multiple Data. 36, 141

watchdog Also known as *watchdog timer*. A timer that is regularly restarted during normal operation to prevent it from elapsing. If it does not get restarted (e.g. because a task is stuck in an infinite loop), it will time out and trigger corrective actions. (typically bringing the system to a safe state) 39, 40

Acronyms

ACET Average Case Execution Time 2, 8

BCET Best Case Execution Time 8, 9, 46, 60

CPS cyber-physical system 6

CPU Central Processing Unit 11, 13, 15, 26, 28, 29, 40, 44, 57, 60

FI Fault Injection 7

ISA Instruction Set Architecture 29

MMU Memory Management Unit 26

PCL Performance Counters for Linux 19

PMU Performance Monitoring Unit 19, 20, 60

RAM Random Access Memory 28

RTCS Real-Time Computer System 5

RTS Real-Time System 5, 6

SE Syscall Emulation 29

SoC System-on-a-Chip 25, 26, 57, 60

SSH Secure Shell 28

syscall system call 7, 17, 19, 20, 36

WCET Worst Case Execution Time 2, 8, 9, 12, 13, 36, 46, 47, 56, 57, 59, 60, 133

Bibliography

[ALR04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. “Basic concepts and taxonomy of dependable and secure computing”. In: *Dependable and Secure Computing, IEEE Transactions on* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[ARM11] ARM Ltd. *Cortex™ -R4 and Cortex-R4F. Technical Reference Manual*. Revision: r1p4. Apr. 2011. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0363g/DDI0363G_cortex_r4_r1p4_trm.pdf (visited on 2016-02-19).

[ARM12] ARM Ltd. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. Issue C.b. July 2012. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html (visited on 2015-12-23).

[ARM13] ARM Ltd. *Cortex™ -A7 MPCore™. Technical Reference Manual*. Revision: r0p5. Apr. 2013. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464f/DDI0464F_cortex_a7_mpcore_r0p5_trm.pdf (visited on 2015-11-27).

[ASJ13] Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. “Computer Safety, Reliability, and Security: 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013. Proceedings”. In: ed. by Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Chap. A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution, pp. 265–276. ISBN: 978-3-642-40793-2. DOI: 10.1007/978-3-642-40793-2_24. URL: http://dx.doi.org/10.1007/978-3-642-40793-2_24.

[Bak15] J.D. Bakos. *Embedded Systems: ARM Programming and Optimization*. Elsevier Science, 2015. Chap. 1.9. ISBN: 9780128004128. URL: https://books.google.at/books?id=Y9qcBAAAQBAJ.

[BBB11] Nathan Binkert, Bradford Beckmann, Gabriel Black, et al. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718.

[BDH06] 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”. In: *IEEE Micro* 26.4 (July 2006), pp. 52–60. ISSN: 0272-1732. DOI: 10.1109/MM.2006.82. URL: http://dx.doi.org/10.1109/MM.2006.82.

[BGO12] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. “Accuracy evaluation of GEM5 simulator system”. In: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*. July 2012, pp. 1–7. DOI: 10.1109/ReCoSoC.2012.6322869.

[CBC92] R. Chillarege, I.S. Bhandari, J.K. Chaar, et al. “Orthogonal defect classification-a concept for in-process measurements”. In: *Software Engineering, IEEE Transactions on* 18.11 (Nov. 1992), pp. 943–956. ISSN: 0098-5589. DOI: 10.1109/32.177364.

[CCS99] J.V. Carreira, D. Costa, and J.G. Silva. “Fault injection spot-checks computer system dependability”. In: *Spectrum, IEEE* 36.8 (Aug. 1999), pp. 50–55. ISSN: 0018-9235. DOI: 10.1109/6.780999. URL: http://dx.doi.org/10.1109/6.780999.

[CT65] J.W. Cooley and J.W. Tukey. “An algorithm for the machine calculation of complex Fourier series”. In: *Mathematics of computation* 19.90 (1965), pp. 297–301. DOI: 10.1090/S0025-5718-1965-0178586-1.

[Dre07] Ulrich Drepper. “What every programmer should know about memory”. In: *Red Hat, Inc* 11 (2007). URL: http://www.akkadia.org/drepper/cpumemory.pdf.

[ECC14] Fernando Endo, Damien Couroussé, Henri-Pierre Charles, et al. “Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*. IEEE. July 2014, pp. 266–273. DOI: 10.1109/SAMOS.2014.6893220.

[ECC15] Fernando A. Endo, Damien Couroussé, and Henri-Pierre Charles. “Micro-architectural Simulation of Embedded Core Heterogeneity with Gem5 and McPAT”. In: *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. RAPIDO ’15. Amsterdam, Holland: ACM, 2015, 7:1–7:6. ISBN: 978-1-60558-699-1. DOI: 10.1145/2693433.2693440. URL: http://doi.acm.org/10.1145/2693433.2693440.

[Flo64] Robert W. Floyd. “Algorithm 245: Treesort”. In: *Commun. ACM* 7.12 (Dec. 1964), pp. 701–. ISSN: 0001-0782. DOI: 10.1145/355588.365103. URL: http://doi.acm.org/10.1145/355588.365103.

[Fou15] Raspberry Pi Foundation. *Raspberry Pi 2 Model B*. Accessed: 2015-10-17. 2015. URL: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[GBE10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. “The Mälardalen WCET Benchmarks: Past, Present And Future”. In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Ed. by Björn Lisper. Vol. 15. OpenAccess Series in Informatics (OASIcs). The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146. ISBN: 978-3-939897-21-7. DOI: 10.4230/OASIcs.WCET.2010.136. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2833.

[Gon15] A. González. *Embedded Linux Projects Using Yocto Project Cookbook*. EBL-Schweitzer. Packt Publishing, 2015, 233 ff. ISBN: 9781784396343. URL: https://books.google.at/books?id=yNi6BwAAQBAJ.

[Hoa61] C. A. R. Hoare. “Algorithm 64: Quicksort”. In: *Commun. ACM* 4.7 (July 1961), pp. 321–. ISSN: 0001-0782. DOI: 10.1145/366622.366644. URL: http://doi.acm.org/10.1145/366622.366644.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. “Fault Injection Techniques and Tools”. In: *Computer* 30.4 (Apr. 1997), pp. 75–82. ISSN: 0018-9162. DOI: 10.1109/2.585157. URL: http://dx.doi.org/10.1109/2.585157.

[IEE12] IEEE Standards Association. *IEEE Std 802.3-2012 - Ethernet*. Tech. rep. IEEE, Dec. 2012. URL: http://standards.ieee.org/about/get/802/802.3.html.

[IKH14] Mafijul Md. Islam, Nithilan Meenakshi Karunakaran, Johan Haraldsson, Fredrik Bernin, and Johan Karlsson. “Binary-Level Fault Injection for AUTOSAR Systems (Short Paper)”. In: *Proceedings of the 2014 Tenth European Dependable Computing Conference*. EDCC '14. Washington, DC, USA: IEEE Computer Society, May 2014, pp. 138–141. ISBN: 978-1-4799-3804-9. DOI: 10.1109/EDCC.2014.21. URL: http://dx.doi.org/10.1109/EDCC.2014.21.

[IS83] Janet Incerpi and Robert Sedgewick. “Improved upper bounds on shellsort”. In: *Foundations of Computer Science, 1983., 24th Annual Symposium on*. Nov. 1983, pp. 48–55. DOI: 10.1109/SFCS.1983.26.

[Jef11] Brian Jeff. *Enabling Mobile Innovation with the Cortex™ - A7 Processor*. Tech. rep. ARM Ltd., Oct. 2011. URL: https://community.arm.com/docs/DOC-7612.

[JH11] Yue Jia and M. Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *Software Engineering, IEEE Transactions on* 37.5 (Sept. 2011), pp. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62.

[JSM07] A. Johansson, N. Suri, and B. Murphy. “On the Selection of Error Model(s) for OS Robustness Evaluation”. In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. June 2007, pp. 502–511. DOI: 10.1109/DSN.2007.71.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd Edition. Springer Publishing Company, Incorporated, 2011. ISBN: 1441982361, 9781441982360. DOI: 10.1007/978-1-4419-8237-7.

[KP10] Raimund Kirner and Peter Puschner. "Time-Predictable Computing". English. In: *Software Technologies for Embedded and Ubiquitous Systems*. Ed. by SangLyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer. Vol. 6399. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 23–34. ISBN: 978-3-642-16255-8. DOI: 10.1007/978-3-642-16256-5_5. URL: http://dx.doi.org/10.1007/978-3-642-16256-5_5.

[LS15] Edward A Lee and Sanjit A Seshia. *Introduction to Embedded Systems. A Cyber-Physical Systems Approach*. Second Edition. leeseshia.org, 2015. ISBN: 978-1-312-42740-2. URL: http://leeseshia.org/.

[MSB05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, et al. "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset". In: *SIGARCH Comput. Archit. News* 33.4 (Nov. 2005), pp. 92–99. ISSN: 0163-5964. DOI: 10.1145/1105734.1105747. URL: http://doi.acm.org/10.1145/1105734.1105747.

[NIS01] NIST FIPS Pub. "197: Advanced encryption standard (AES)". In: *Federal Information Processing Standards Publication* 197 (2001), pp. 441–0311. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[PKH12] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. "Compiling for Time Predictability". English. In: *Computer Safety, Reliability, and Security*. Ed. by Frank Ortmeier and Peter Daniel. Vol. 7613. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 382–391. ISBN: 978-3-642-33674-4. DOI: 10.1007/978-3-642-33675-1_35. URL: http://dx.doi.org/10.1007/978-3-642-33675-1_35.

[Pus03] P. Puschner. "The single-path approach towards WCET-analysable software". In: *Industrial Technology, 2003 IEEE International Conference on*. Vol. 2. Dec. 2003, 699–704 Vol.2. DOI: 10.1109/ICIT.2003.1290740.

[Pus05] Peter P. Puschner. "Experiments with WCET-oriented programming and the single-path architecture". In: *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*. Feb. 2005, pp. 205–210. DOI: 10.1109/WORDS.2005.36.

[San97] Santa Cruz Operation, Inc. *System V Application Binary Interface*. Edition 4.1. The Santa Cruz Operation, Inc. 1997. URL: https://refspecs.linuxbase.org/elf/gabi41.pdf (visited on 2015-12-04).

[Sed78] Robert Sedgewick. "Implementing Quicksort Programs". In: *Commun. ACM* 21.10 (Oct. 1978), pp. 847–857. ISSN: 0001-0782. DOI: 10.1145/359619.359631. URL: http://doi.acm.org/10.1145/359619.359631.

[Sed96] Robert Sedgewick. “Analysis of Shellsort and related algorithms”. English. In: *Algorithms — ESA ’96*. Ed. by Josep Diaz and Maria Serna. Vol. 1136. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 1–11. ISBN: 978-3-540-61680-1. DOI: 10.1007/3-540-61680-2_42. URL: http://dx.doi.org/10.1007/3-540-61680-2_42.

[Sed97] Robert Sedgewick. *Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Pearson Education, 1997. ISBN: 9780768685015. URL: https://books.google.at/books?id=7OdVCgAAQBAJ.

[VCL11] Robert A. Vitillo, Paolo Calafiura, and Wim Lavrijsen. “Performance Tools Developments”. Future computing in particle physics, Edinburgh, June 2011. URL: http://indico.cern.ch/event/141309/session/4/contribution/20/attachments/126021/178987/RobertoVitillo_FutureTech_EDI.pdf (visited on 2015-12-06).

[War12] Henry S Warren. *Hacker’s Delight*. 2nd Edition. Pearson Education, 2012. ISBN: 9780133085013. URL: http://www.hackersdelight.org/.

[WEE08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, et al. “The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools”. In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. URL: http://doi.acm.org/10.1145/1347375.1347389.

[Wil64] J. W. J. Williams. “Algorithm 232: Heapsort”. In: *Commun. ACM* 7.6 (June 1964). Ed. by G. E. Forsythe, pp. 347–348. ISSN: 0001-0782. DOI: 10.1145/512274.512284. URL: http://doi.acm.org/10.1145/512274.512284.