NTNU
Norwegian University of
Science and Technology

Climbing Mont Blanc - Back-end Improvements

Fredrik Pe Ingebrigtsen

Master of Science in Computer Science

Submission date:	July 2017
Supervisor:	Lasse Natvig, IDI
Co-supervisor:	Waqar Hussain, IDI
	Sindre Magnussen, Itera

Norwegian University of Science and Technology
Department of Computer Science

Climbing Mont Blanc - Backend Improvements

Climbing Mont Blanc (CMB) is a system for evaluation of programs executed on modern heterogeneous multicores, such as the Exynos Octa chips used in Samsung Galaxy S5 and S6 mobile phones. CMB evaluates both performance and energy efficiency, and provides the possibility of performance ranking lists and online competitions.

The system is available and under trial use. This master thesis project is focused at improving the more low-level aspects of the system, i.e., execution of code on the XU3-boards (backend).

The project has the following goals:

1. Generate and provide low level statistics for each submission.
2. Port bash scripts to Python.
3. Allow easier uploading of submissions (i.e., removing the requirement of uploading a ZIP-file containing the source file(s)).
4. Show, with examples, how the new metrics (see item 1) can help users improve the performance of their solutions.
5. Cleanup and improvements in the system architecture.
6. Analyze performance of parallel OpenMP programs.
7. Propose further improvements.

If time permits:

1. Create a integration test.
2. Improve general stability of the system.

The master thesis project is part of the EECS Strategic Research project at IME (www.ntnu.edu/ime/eecs).

Main supervisor: Prof. Lasse Natvig

Co-supervisor: MSc Sindre Magnussen.

Abstract

Energy efficiency in computing is becoming more and more important. With the rise of smart phones, a whole new industry was born where having a more energy efficient system would mean longer battery life and an edge over the competition. It has also recently been an area of interest in High-Perfomance Computing (HPC). This has fuelled research and development of heterogeneous multi-core architectures, utilizing different CPU cores to do different tasks.

Utilizing heterogeneous architectures fully is a challenge both for the hardware and software engineers. In Online judging systems, users can compete and learn while solving problems, getting feedback on correctness and efficiency of their submissions. Climbing Mont Blanc is an online judging system focusing on energy efficiency on heterogeneous multi-cores, and is to our knowledge the only such system measuring energy efficiency, aiming to provide an environment for education and practice in energy efficient programming.

The CMB system currently reports time, energy and energy delay product (EDP) per submission. To assist users in performance tuning their solutions, and to give a better picture of what the program execution looked liked, some more detailed low-level statistics were wanted as user feedback. In addition, some general system architectural improvements were needed to improve stability and ease of development. This thesis focuses improving the system with regards to these goals.

Sammendrag

Energieffektivitet har i det siste blitt mer og mer viktig. Med fremgangen av smarttelefoner har det vokst frem en helt ny industri, hvor energieffektivitet er særdeles viktig, spesielt med tanke på batterilevetid. Det er også vokst frem en interesse for energieffektivitet innen High-Performance computing (HPC). Dette har sammen inspirert mye ny forskning og nye produkter, som heterogene multikjerner, der forskjellige kjerner har spesialiserte formål.

Hvordan man best benytter seg av disse nye arkitekturene er en utfordring for både hardware of sowftware ingeniører. Online dømme systemer finnes der brukere kan konkurrere og lære mens de løser ulike problemer, med tilbakemeldinger om korrekthet og ytelse. Climbing Mont Blanc (CMB) er et slikt online dømme system, med fokus på energi effektivitet på heterogene multikjerner, og er, så vidt vi vet, det eneste systemet som måler energibruk, med mål om og lage et miljø der brukerne kan utdanne og øve seg på å programmere energieffektive løsninger.

CMB foreløpig rapporterer tidsbruk, energibruk og energiutsettelsesprodukt (EDP) per brukeropplasting. For å hjelpe brukerne med å justere løsningene sine, og for å gi et klarere bilde av hva som skjer i løpet av kodekjøringen, har mer detaljerte lavnivå informasjon vært ønsket av CMB teamet. I tillegg har noen mer generelle systemforbedringer vært trengt, for å forbedre systemstabilitet og å forenkle utvikling. Denne oppgaven omhandler forbedringen av CMB systemet i henhold til disse målene.

Preface

This thesis was created to fulfill a MSc in computer science at the Norwegian University of Science and Technology (NTNU), Trondheim. This work has been conducted at the Department of Computer and Information Science NTNU in the spring of 2017, alongside being employed as part of the course staff as a teaching assistant in TDT4102 [TDT17].

Acknowledgements

I would like to thank my supervisor Lasse Natvig for letting me contribute to the CMB system, and for his constructive guidance and feedback throughout the semester. His positivity has been greatly appreciated.

I would also like to thank my co-supervisor Sindre Magnussen for his technical help along the way, and always being available when I had questions. Explaining the system in the beginning, and helping setting it up on my own PC saved me a lot of effort, not to mention the code reviews he provided, which was of great benefit.

I addition, I would like to thank Jan Grønsberg for his help with the technical management of the servers and databases.

Table of Contents

List of Figures

Abbreviations

MBP	=	Mont Blanc Project
CMB	=	Climbing Mont Blanc
HPC	=	High-Performance Computing
CPU	=	Central Processing Unit
OS	=	Operating System
API	=	Application Programming Interface
OJ	=	Online Judge
SSH	=	Secure Shell Login
SCP	=	Secure Copy
UFW	=	Uncomplicated Firewall
EDP	=	Energy-Delay Product

Chapter 1

Introduction

The Climbing Mont Blanc system is an online judge focusing on energy efficiency. The system has been developed by master students at NTNU. In figure 1.1, an overview of the system is shown. This thesis' goals are improving various parts of the system, particularly adding more feedback about performance.

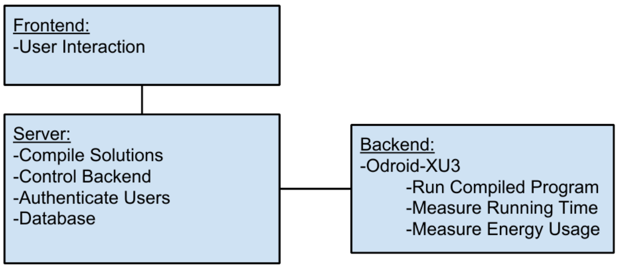


Figure 1.1: CMB system overview (taken from the master thesis of Follan and Støa [TF15]).

In section 1.1 the motivation for this thesis is outlined, section 1.2 presents the goals in detail, while section 1.3 explains the thesis structure.

1.1 Motivation

Energy efficiency in computing is becoming more and more important. With the rise of smart phones, a whole new industry was born where having a more energy efficient system would mean longer battery life and an edge over the competition. This has fuelled research and development of heterogeneous multi-core architectures, utilizing different CPU cores to do different tasks. One such example is the ARM big.LITTLE, which features four smaller, energy efficient cores, and four larger compute cores [ABL17].

The Mont Blanc Project (MBP) is an effort to investigate and demonstrate the usefulness of heterogeneous architecture in high-performance computing (HPC). Performance is the main objective in HPC, but as the new commodity platforms become better, constructing a supercomputer of such elements looks promising [RCG$^+$13]. The MBP aims to create a prototype said supercomputer, using less than 15 - 30x the energy of a regular supercomputer. If successful this would hugely decrease the cost of super-computing as the electricity bill is often surpasses construction costs in a matter of years [FFG08].

Utilizing heterogeneous architectures fully is a challenge both for the hardware and software engineers. Online judging systems (OJ) are platforms where users can compete and learn while solving problems, getting feedback on correctness and efficiency of their submissions. Climbing Mont Blanc (CMB) is an online judging (OJ) system focusing on energy efficiency on heterogeneous multi-cores (the ARM big.LITTLE). To our knowledge it is the only OJ measuring energy efficiency, and aims to provide an environment for education and practice in energy efficient programming.

The CMB system currently reports time, energy and energy delay product (EDP) per submission. To assist users in performance tuning their solutions, and to give a better picture of what the program execution looked liked, some more detailed low-level statistics were wanted as user feedback. Such performance tuning is perhaps easier and can be done in more detail on the users own machine before uploading, but because different architectures greatly impacts code execution, metrics specific to the Odroid XU3 backend (described further in section 2.2.3) is an advantage.

In addition, some general system architectural improvements were needed to improve stability and ease development. This thesis focuses improving the system with regards to these goals, outlined in further detail in the next section.

1.2 Thesis Goals

In this thesis the goal is to further improve the CMB system following the goals mentioned in the problem statement. They can be divided roughly in three categories: new features, general system improvements, and analysis and demonstration.

New Features

1. Generate and provide low level statistics for each submission: This is the main objective of this thesis, and involves generating a set of metrics that when provided to the user will assist in performance tuning of submissions. Exactly what these should be is an open question, and will be dependant on the particular problem, submission, among other things. It is therefore an aim to provide statistics that will be useful in a number of situations, like where CPU time is spent, memory usage, parallelization, and with regards to energy efficiency - detailed energy use.

General System Improvements

2. Port Bash scripts to Python: The bash scripts used in the system had consistently been a source of struggle to the CMB developers, and converting them to Python scripts was believed to help with debugging and stability, as well allowing for their functionality to be tested in the same testing framework as the rest of the system.

3. Improve general stability of the system: The server goes down from time to time, and while it has been a simple enough task to restart it, investigating the cause of theses failures and finding a solution would ease the administrative responsibilities.

4. Allow single source-file uploads: Magnussen in [Mag16], conducted a user study where it was found that the format and structure of file uploads was unclear, and that uploading of single source files was requested. Allowing single source-file uploads would further increase usability of the site.

Analysis

5. Show, how the new metrics help users improve the performance of their solutions: With the addition of more feedback, a demonstration of how they may help in performance tuning is provided to assess their usefulness and suitability.

6. Analyze performance of parallel OpenMP programs: This includes a general analysis of the performance and energy efficiency of OpenMP programs on the ARM big.LITTLE heterogeneous multi-core, as well as an investigation of how the CPU affinity features of OpenMP 4 is suited on the same architecture.

Other

7. Propose further improvements: A list of further possible improvements to the CMB system should be proposed, delineating potential implementation details.

8. Create a integration test (if time permits): The CMB system lacks an integration test, the inclusion of which would help development, detecting potential cross-unit bugs not exposed by the unit tests.

1.3 Thesis Structure

This thesis is structured as follows:

Chapter 2: This chapter presents background theory. The MBP is introduced, a thorough description of the current CMB system is presented, followed by an introduction to OpenMP, an overview of profilers and what they do, as well as an examination of the most known and popular online judges.

Chapter 3: Here the implementation details of the improvements added in this thesis is described. This includes the implementation of the new low-level statistics, from the backend to the frontend, the porting of Bash scripts to Python, allowing single source-file upload, stability improvements, and some other general system improvements.

Chapter 4: This chapter demonstrates the use of the new metrics, and presents an analysis of OpenMP performance on the Odroid XU3 board, including an examination the CPU affinity features of OpenMP 4 and how they can be used in the NAS Parallel Benchmarks to improve performance.

Chapter 5: The system stability is evaluated in this chapter, as well as an examination of measurement accuracy, reproducing the results found by Follan and Støa in [TF15].

Chapter 6: This chapter contains a discussion of the choice of low-level statistics, including the strengths and weaknesses of the possible choices, and the limitations of the final implementation.

Chapter 7: In this chapter a conclusion to the thesis and proposals of further work is presented.

Chapter 2

Background

In this chapter, the Mont Blanc Project is first presented in section 2.1, followed by an in depth description of the Climbing Mont Blanc (CMB) system in section 2.2. Section 2.3 is an introduction to OpenMP, and section 2.4 explains what profilers are, giving some well known examples. Finally in section 2.5, some existing work related to online judging systems is presented.

2.1 The Mont Blanc Project

The Mont-Blanc project has from its beginning in 2011 aimed at developing a new type of super-computer architecture, built from energy efficient solutions used in embedded and mobile devices, capable of creating new global HPC standards. The project is coordinated by the Barcelona Supercomputing Centre (BSC), with funding from the European Commission. Starting the first phase of the project with a budget of 14 million, the project has since then been extended for two more phases, the second from 2013-2016, and the current third phase, coordinated by Bull, from 2015-2018.

2.1.1 Prototypes

The Mont-Blanc Project have created several prototypes, the two most notable are described here.

Tibidabo was the first prototype created by the MB project. It is the worlds first ARM-based HPC cluster, built using commodity off-the-shelf components that are not designed for HPC [RRP+14]. The prototype contains compute-boards with NVIDIA Tegra 2 SoCs, with dual core ARM Cortex A9 @ 1GHz inside. It achieves 120 MFLOPS/W on HPL, competitive with AMD Operton 6128 and Intel Xeon X5660-based systems. The MB project identified a an inefficiency in that the power taken by the components required to integrate small low power dual-core processors offsets the high energy efficiency of the cores themselves.

Mont Blanc is the latest prototype from the MB project [Ram14]. The Mont-Blanc compute node is a Server-on-Module architecture. Each node is built around a Samsung Exynos 5250 mobile SoC containing ARM Cortex-A15 CPUs at 1.7 GHz dual core configuration sharing 1 MB of on-die L2 cache, and a mobile ARM Mali-T604 GPU [RRM+16]. The performance is about 35 TFLOPS and the power consumption is about 24 kW. In November 2014, the Mont Blanc Prototype had an energy eciency of 1.5 GFLOPS/W. The best Green500 ranking during that time was approximately 5.2 GFLOPS/W.

2.2 Climbing Mont Blanc

Climbing Mont Blanc (CMB) is an online judging system developed by previous master students at NTNU. Briefly put, it is a website containing various programming problems, where users may upload solutions to check their validity and performance. The site has been used for hosting exercises in various courses at NTNU. CMB is inspired by the Mont Blanc project, and is as far as we know, the only such system where energy efficiency on heterogeneous architecture is the main focus. The system was first prototyped by Torbjørn Follan and Simen Støa in their master thesis [TF15]. The following year two more master students continued developing the system, independently on two different parts. Christian Chavez examined in his master thesis the possibility of adding more backends to the system, for improving scalability, by adding a dispatcher for serving submissions to different backends [Cha16]. Sindre Magnussen improved the system usability, as well as other enhancements including changing the database management system to MySQL [Mag16]. This thesis continues development of the system version Magnussen created. Figure 2.1 shows an overview of the CMB system architecture and communication flow.

2.2.1 Frontend

The frontend holds all the code for the graphical user interface at the website, and manages communication with the server. It is a single-page web application, which only loads *one* HTML page, and dynamically updates it according to user interaction. It is built using AngularJS [AN17], which is a JavaScript-based open-source framework, maintained by Google. The frontend uses the Model-View-Controller pattern, where each view presented

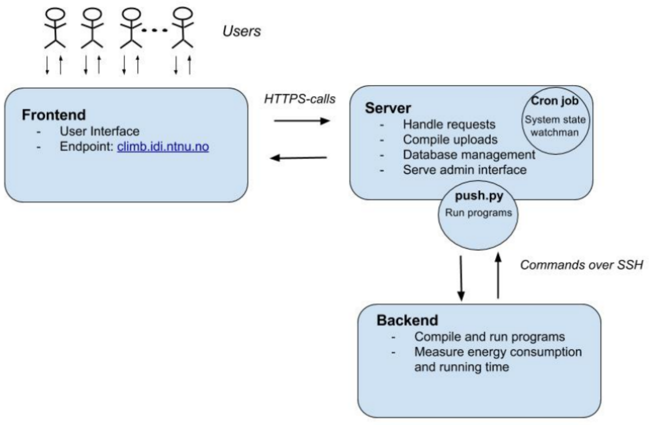


Figure 2.1: CMB system architecture (taken from the master thesis of Magnussen [Mag16]).

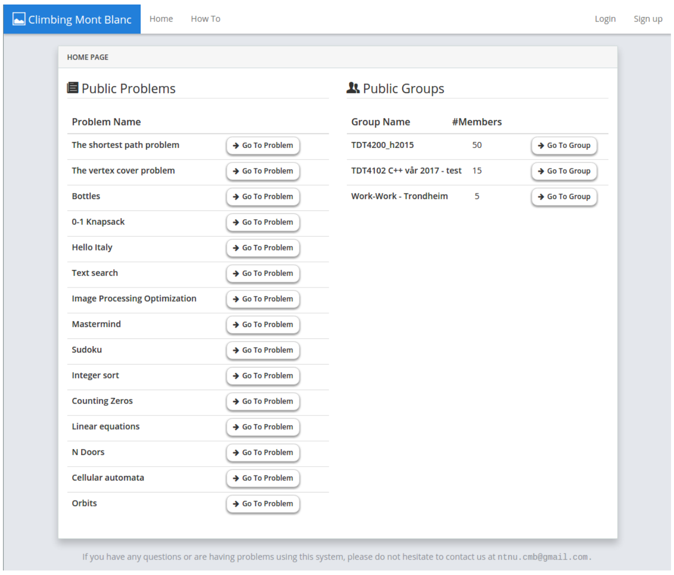
to the user has an associated controller, which retrieves data from the server to build a model. When the user changes the model through the view, the controller is responsible for updating its model as well as notifying the server with the new data. Figure 2.2 shows the main view of CMB. Clicking on a problem switches to the problem view, shown in figure 2.3, fetching related data from the server.

The frontend is hosted on the same physical machine as the server, but this is not a requirement. It uses the Node.js JavaScript run-time environment, which enables JavaScript to be used on the server, e.g., as scripts to produce dynamic web page content before the page is sent to the user's web browser.

Communication with the server is done mainly by Hyper Text Transfer Protocol (HTTP) requests that retrieves and delivers data to the server. Socket.io is used to allow real-time automatic updates in the frontend. Socket.io is an API that based on type of client and server automatically detects supported communication protocols, and sets up a ”socket” for persistent communication. When this channel is initiated, the server can notify the client when certain events happen (e.g. program compiled successfully), enabling real-time updates to the frontend.

2.2.2 Server

The server uses Python FLASK, which is an implementation of a REpresentational State Transfer [FT02] service in Python. A REST API satisfies certain constraints. It is stateless, meaning the necessary state to handle the request is contained within the request itself. Secondly, it has a uniform interface which ensures that requests are the same independently of intermediate components. Furthermore, it provides a clear client-server separation,



Public Problems

Problem Name	
The shortest path problem	Go To Problem
The vertex cover problem	Go To Problem
Bottles	Go To Problem
0-1 Knapsack	Go To Problem
Hello Italy	Go To Problem
Text search	Go To Problem
Image Processing Optimization	Go To Problem
Mastermind	Go To Problem
Sudoku	Go To Problem
Integer sort	Go To Problem
Counting Zeros	Go To Problem
Linear equations	Go To Problem
N Doors	Go To Problem
Cellular automata	Go To Problem
Orbits	Go To Problem

Public Groups

Group Name	#Members	
TDT4200_h2015	50	Go To Group
TDT4102 C++ vår 2017 - test	15	Go To Group
Work-Work - Trondheim	5	Go To Group

If you have any questions or are having problems using this system, please do not hesitate to contact us at ntnu.cmb@gmail.com.

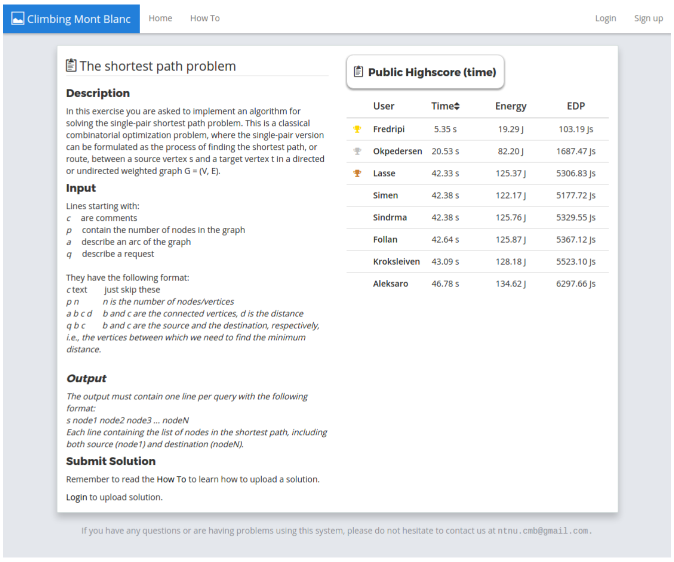
Figure 2.2: The main view of CMB.

cacheableness of requests, and possibility of layering so that a client cannot tell if it is directly connected to the server, or indirectly via some intermediary.

Other technologies used by the server includes: Gunicorn [GU17], a Python web server gateway interface (WSGI) HTTP server for UNIX systems, NGINX [NG17], a reverse proxy serving static files separately from dynamic content, which are forwarded to Gunicorn, and MySQL [MyS01], the database of the system. The underlying database schema is shown in Figure 2.4.

The admin frontend, found at the `climb.idi.ntnu.no/admin` endpoint, is hosted by the server. This is an interface where admin users can inspect the database, and make changes to it, such as deleting records or inserting new (for adding new problems). The admin interface also facilitates changing problem visibility, adding/removing users or admin users, amongst other things.

When the server receives submissions from the frontend, three steps are made. The server first tries to compile the files, and emits a Socket.io message with the results. If compilation exits without errors the files are stored in the file system. Lastly, an element is added to the FIFO queue keeping track of submissions that are ready to be executed on the backend. When the server receives the results from the board, containing measurement data as well as the program output, it performs a correctness check, updates the database, and finally informs the client if the submission succeeded.



The shortest path problem

Description

In this exercise you are asked to implement an algorithm for solving the single-pair shortest path problem. This is a classical combinatorial optimization problem, where the single-pair version can be formulated as the process of finding the shortest path, or route, between a source vertex s and a target vertex t in a directed or undirected weighted graph G = (V, E).

Input

Lines starting with:
c are comments
p contain the number of nodes in the graph
a describe an arc of the graph
q describe a request

They have the following format:
c text just skip these
p n *n is the number of nodes/vertices*
a b c d *b and c are the connected vertices, d is the distance*
q b c *b and c are the source and the destination, respectively, i.e., the vertices between which we need to find the minimum distance.*

Output

The output must contain one line per query with the following format:
s node1 node2 node3 ... nodeN
Each line containing the list of nodes in the shortest path, including both source (node1) and destination (nodeN).

Submit Solution

Remember to read the How To to learn how to upload a solution.

Login to upload solution.

Public Highscore (time)

	User	Time	Energy	EDP
🏆	Fredripi	5.35 s	19.29 J	103.19 Js
🏆	Okpedersen	20.53 s	82.20 J	1687.47 Js
🏆	Lasse	42.33 s	125.37 J	5306.83 Js
	Simen	42.38 s	122.17 J	5177.72 Js
	Sindrma	42.38 s	125.76 J	5329.55 Js
	Follan	42.64 s	125.87 J	5367.12 Js
	Kroksleiven	43.09 s	128.18 J	5523.10 Js
	Aleksaro	46.78 s	134.62 J	6297.66 Js

If you have any questions or are having problems using this system, please do not hesitate to contact us at `ntnu.cmb@gmail.com`.

Figure 2.3: A problem view on CMB.

The `push.py` script

Because the backend can only execute one submission at a time, the aforementioned queue is needed to handle the case where submissions are submitted faster than the backend is able to execute them. Operating this queue is a Python script called `push.py`, which is run as a background process in the server, continuously polling the submission queue, and, if non-empty, notifies the client that the submission is about to be executed, copies the corresponding submission files to the backend via SCP (Secure Copy), starts the backend execution and measuring bash script via SSH, and waits for results. To make sure programs that hangs (e.g., containing infinite loops) don't clog the system, the SSH command exits after a given timeout. Finally, it reports the results back to the server, before beginning the whole procedure anew.

2.2.3 Backend

The backend is an Odroid-XU3 board [OD17], and is responsible for executing and measuring the user-submitted code. The Odroid-XU3 board is one of the ARM-platforms used in the Mont Blanc project [MBP17], and features the Samsung Exynos 5 Octa (5422) [SE517], a System-on-Chip (SoC) with four ARM Cortex-A15 [A1517] and four ARM Cortex-A7 [A717] cores. Thus, it is a heterogeneous multi-core solution, utilizing the ARM big.LITTLE technology [ABL17], which enables the OS to perform Global-Task-Scheduling to dynamically assign threads to the most appropriate CPU based on run-time information. The GPU is a Mali-T628 [AM17] with support for OpenGL ES 3.0/2.0/1.1

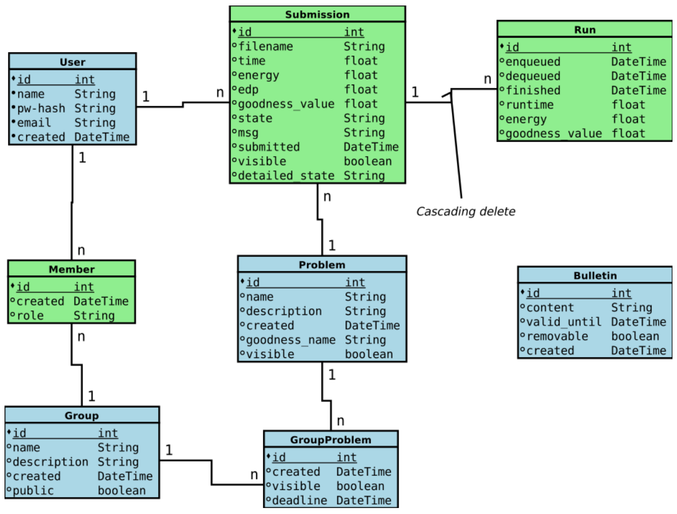


Figure 2.4: Database schema (taken from Master Thesis of Follan and Støa [TF15]).

and OpenCL 1.1. This is also the SoC seen in the Samsung Galaxy S5 smartphone [SG517].

In addition, the board has four integrated energy monitors, capable of measuring the power consumption of the four A-15 CPUs, the four A-7 CPUs, the Mali GPU, and the DRAM in real time. These monitors are used for the energy measuring described further in the next section.

As stated previously, the `push.py` script copies the submission files over the backend, and then starts a bash script over SSH. This run-script has several tasks. First it compiles the code [1], returning prematurely if erroneous, then it performs the small correctness test. The small correctness test is required by all problems, and provides a smaller version of the original problem, so that faulty submissions does not consume unnecessary time. Afterwards, if the test was successful, the initial conditions are set by first clearing the cache, starting the energy monitor, setting CPU temperature, and finally starting the timer (as is seen in figure 2.6). It then executes the program. These initial conditions are set to ensure a fair trial for all submissions, and an analysis of the stability and variance between different submissions were done in [TF15], and in section 5.1 an in depth inspection of measurement accuracy is presented.

[1] As the system does not have a cross-compiler, the compilation at the server is not sufficient and the board must also compile the code.

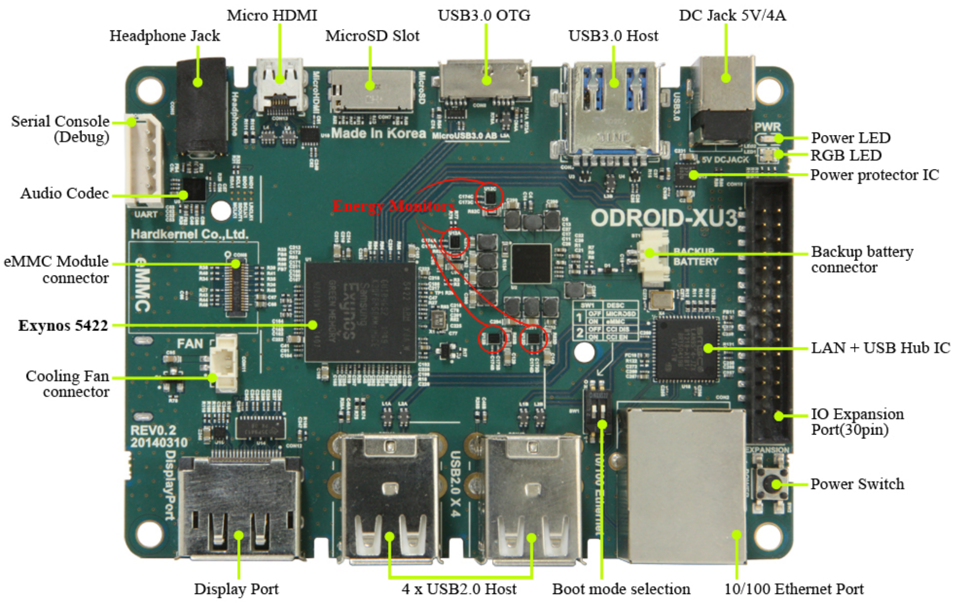


Figure 2.5: Board details of Odroid-XU3 (from the Hardkernel webpage [OD17]).

2.2.4 Energy Measurements

The Hardkernel website provides a program that reads the energy sensors and displays the data [OD17]. After the cache is cleared and the CPU temperature is set, the energy monitor program is started which continuously outputs energy readings, stored in a temporary file. The user program is then executed, and when finished, the energy monitor stops. With the time stamps used for measuring run-time, the relevant energy readings can be filtered and integrated giving the total energy consumption of the user program. The server receives the measured energy usage along with run time and correctness from the backend, and calculates the Energy-Delay Product (EDP).

$$EDP = E * D \tag{2.1}$$

Where E is energy used and D is the delay or execution time. In [HIG94], Gonzales and Horowitz argues that this is a good metric for energy efficiency at the chip level. This metric takes both energy and run time into account. Only considering energy is a poor energy efficiency metric since you could simply run a program slower to get a lower energy consumption. FLOPS/W is another highly used metric, but this metric is more useful when running the program several times to test the energy efficiency of different architectures. CMB However, compares different implementations of the same problems repeatedly, so the number of operations may differ from one implementation to another. For these reasons, EDP is preferred over FLOPS/W in the CMB system and is the primary

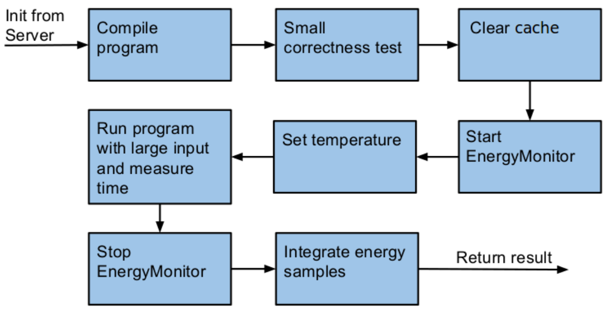


Figure 2.6: Backend execution pipeline (slightly modified from Master Thesis of Follan and Støa [TF15]).

energy efficiency metric used in the system.

2.3 OpenMP

OpenMP is a set of compiler directives and library routines for writing parallel shared-memory programming in C, C++, and FORTRAN [DM98]. A shared memory machine consists of different threads of execution that have access to a shared memory region. The threads run asynchronously, and conflicting memory reads/writes must be handled by the system or the programmer. Shared memory model is the prominent model for small-scale systems, notably SMP (symmetric multiprocessing) systems. OpenMP combines Single Program Multiple Data (SPMD) and fork-join styles of execution and offers work sharing constructs that allow distribution of loop iterations among threads. Aiming to simplify code parallelization, OpenMP allows beginners, as well as experts, to gradually move from serial to parallel programming. It extends serial code, such that with only a few directives, it can greatly improve performance, while maintaining the serial look and feel of the program. How the parallel threads are spawned and managed is left to the compiler, the developer only informs of what code should be parallelized. Should precise thread management and synchronization be needed, e.g., for avoiding data races (a common bug in multi-threaded programs), OpenMP also provides several constructs like atomic, critical, barrier, single, and master.

```
#include <iostream>
#include <omp.h>


int main ()
{
  int nts, tid;

#pragma omp parallel private(tid) shared(nts)
  {
    tid = omp_get_thread_num();
#pragma omp single
    {
      nts = omp_get_num_threads();
    }
#pragma omp critical
    {
      std::cout << "Hello World from thread " << tid << " of " \
        << nts << std::endl;
    }
  }
}
```

Listing 1: Hello World using OpenMP in C++. The first compiler directive (`#pragma omp parallel`) spawns threads that all do the enclosing work. The `single` directive marks work only one thread executes, and `critical` work is done serially.

2.4 Profiling

Profiling is the process of dynamically analyzing programs, learning details about their behaviour, to perform various performance engineering tasks. Different techniques are used to gather data, ranging from code instrumentation, instruction set simulation, hardware interrupts, and performance counters. The gathered data is then used to aid program optimization, and/or debugging, and typically consists of useful information about the program execution, e.g., time used, memory footprint, cache utilization, frequency and duration of function calls, etc. There are generally four ways of profiling a program: event based, using instrumentation, statistical sampling, simulation based, or a combination.

Event based profiling consists of taking measurements at different events, either software events (calls, object creation, thread enter/leave) which may be specific to a programming language, or more general hardware events (cycles, cache-references, branches, etc.), while executing.

Instrumentation based profiling is when code is instrumented with new instructions to collect the required information. This can be done manually by the programmer in the source code, or automatically by a tool at various stages (compile-time, run-time).

Simulation based profilers collects data interactively while the program is run in a simulation. With full control over program execution, these profilers can be very precise although they need information about the underlying architecture for the simulation to be correct. Naturally, they impose a significant overhead on execution time.

Statistical profiling, or *sampling*, probes the program at regular intervals using operating system interrupts (this may also be event-based). Although this approach is a statistical approximation, and the resulting data is not exact, it has very few side-effects, and does not greatly impact the program's time and space requirements. Consequently, sampling may actually give a more authentic picture of whats happening under normal program execution.

The amount of error depends on the sampling interval, if a value is n times the sampling period, the expected error is the square-root of n [FS88]. A way to minimize statistical error is to either make the program run longer, or combine several profiling runs into one output.

Hardware performance counters, used by many profilers, are a set of special-purpose registers, available on most modern microprocessors, that store the counts of hardware related activities, such as cycles, cache misses/references, or instructions. PAPI (The Performance API) is a standard API for using such hardware performance counters. It has two interfaces: a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs [MBDH99].

2.4.1 Valgrind

Valgrind is a simulation based profiler using dynamic binary recompilation techniques, offering a multitude of tools that can automatically detect many bugs, and profile programs in detail [NS07]. Valgrind runs on Linux on a lot of different architectures, and has some initial support for Mac OSX. It includes a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. A Valgrind tool loads the client program into the Valgrind process, and then recompiles the clients machine code, in a just-in-time, execution-driven fashion. The core disassembles the code block into an intermediate representation (IR), which is instrumented with analysis code, and then converted by the core back into machine code.

Being a virtual machine it simulates complete program runs, and as such has a very high precision, suitable for analysis of intricate parts of the program. This also means that Valgrind needs complete control of the program and cannot attach to already running processes. Because of Valgrinds high overhead, often at 20 - 30x execution time, it is unsuited for our purposes. Keeping profiling times relatively low is essential to giving a smooth user experience. In addition the system currently does not have a multitude of backends, so queue buildup is also a concern.

2.4.2 VTune

Intel's VTune Amplifier is a commercial performance profiler, but educational licenses are also available. VTune supports a wide variety of profiling techniques, including threading analysis of OpenMP and native threads, GPU application tuning, embedded systems (with energy profiling for battery use), and more [IN17]. It has both a graphical user interface and a command line tool, supporting source code analysis of many different languages.

Unfortunately, VTune is only supported on Intel CPU's, with some support on AMD CPU's. ARM architecture is not supported, hence it could not be used on our backends.

2.4.3 `perf`

The linux `perf` command is a powerful profiling tool included in the linux kernel [PE17]. It began as a tool for using performance counters, and has later expanded to include tracing capabilities. There are two main sub commands for profiling: `perf stat`, and `perf record`.

`perf stat` counts the number of times a given event happen. These events range from hardware events like cycles, cache-misses or instructions, to software events like context switching, or page faults. The command has several modes to count in, including per CPU, per thread, per process, system-wide, in the user level, kernel level, or both. Listing 2 shows an example run of `perf stat`, were default counts are collected system-wide for 1 second.

```
Performance counter stats for 'sleep 1':

   8052.157128 task-clock                # 8.020 CPUs utilized   [100.00%]
           896 context-switches          # 0.111 K/sec           [100.00%]
            27 cpu-migrations            # 0.003 K/sec           [100.00%]
           536 page-faults               # 0.067 K/sec
   116,502,087 cycles                    # 0.014 GHz             [100.00%]
  <not supported> stalled-cycles-frontend
  <not supported> stalled-cycles-backend
    35,257,060 instructions              # 0.30  insns per cycle [100.00%]
     3,982,809 branches                  # 0.495 M/sec           [100.00%]
       707,928 branch-misses             # 17.77% of all branches

   1.004007793 seconds time elapsed
```

Listing 2: Example output of `perf stat -a sleep 1`, which counts default counters system-wide for 1 second.

`perf record` is an event-based sampling profiler, which at a given frequency (or period) on a given event, collects information including time stamps and stack-traces (optional). As with `perf stat`, the mode can be specified. In listing 3 an example of how `perf record` can be used is shown. This example would record stack traces (and timings) for the specified background process for 10 seconds. `perf record` stores the results in a file which can be viewed later by the `perf report` or `perf annotate` commands.

```
perf record -F 99    # At 99 Hz
    -p PID           # Process to monitor
    -g               # Collect stack-traces
    -- sleep 10      # For 10 seconds
```

Listing 3: Example use of `perf record`, collecting stack traces of a background process for 10 seconds.

2.4.4 Poor man's profiler

The poor man's profiler is the manual way of looking at stack-traces [PMP17]. Many sampling profilers don't provide a way to differentiate threads in multi-threaded programs, often only providing data per single thread or all combined. They can show where CPU time is spent, but not what individual threads are blocking on. Many debuggers, e.g. GDB, can walk all threads and provide stacks. The idea then, is to manually use debuggers, or other tools, and collect stack-traces of background running processes at regular intervals. An example implementation is shown in listing 4. This is a small Bash script where, in a for-loop, a tool is used to collect 1000 stack-traces with two second intervals.

```
#!/bin/bash
nsamples=1000
sleeptime=2
pid=$(pidof mysqld)

for x in $(seq 1 $nsamples)
  do
    quickstack -f -p $pid
    sleep $sleeptime
  done
```

Listing 4: Example of a poor man's profiler, collecting stacks with quickstack every two seconds.

Quickstack

Quickstack is one such tool, which tries to minimize overheads [QS17]. It internally scans stack frames and guesses caller functions. Debuggers, which are way more complicated and provides lots of different features, may have a significant overhead when collecting stack-traces, momentarily stopping the process being monitored. Thus, they are not a good fit when you are collecting with a high frequency [2].

Quickstack does not support ARM processors, but a patch was applied to fix this [QF17]. As discussed in section 6.1.1, we did not end up using quickstack, but the implementation details are presented in Appendix B.1.

2.4.5 Flame Graph

Flame graphs are a method of visualizing call-stacks [Gre16]. Many profilers are capable of generating call-stacks, but a good way of displaying them is not always provided. The flame graph displays call-stacks as horizontal bars, where each callee is on top of their respective caller. This creates a choppy graph (resembling flames), and is perhaps best explained with an example. Figure 2.7 shows an example profile of a Java program. In this example the coloring corresponds to the different libraries the calls originates from: green = Java, yellow = C++, orange = kernel, red = user and system [3]. As the horizontal length correlates to amount of samples, i.e, estimated CPU time, flame graphs can immediately show the user where in the code path most time is used, and where potential bottlenecks are located.

The author of [Gre16], Brendan Greggs, has also made an open source implementation, hosted at Github [FG17]. This includes programs for interpreting the output of many different profilers, as well as the flame graph generator, which uses the SVG file format, an XML-based vector image format for two-dimensional graphics, to create flame graphs supporting zoom and search, easily inserted in HTML code.

2.5 Related Work

This section presents some of the best known online judging systems, as well as some crowd sourcing sites. These are both sites hosting programming problems, and data sets for their users which can then upload solutions to the problem. The difference of course, is that crowd sourcing gives out real world problems on behalf of companies which then in turn receives the best solutions. These sites often give out money prices to the top contenders. Online judging systems on the other hand, have an educational or recreational focus.

[2]Discussed further in section 6.1.1

[3]The yellow C++ functions with Java in their name are from the JVM (Java Virtual Machine), which is implemented in C++.

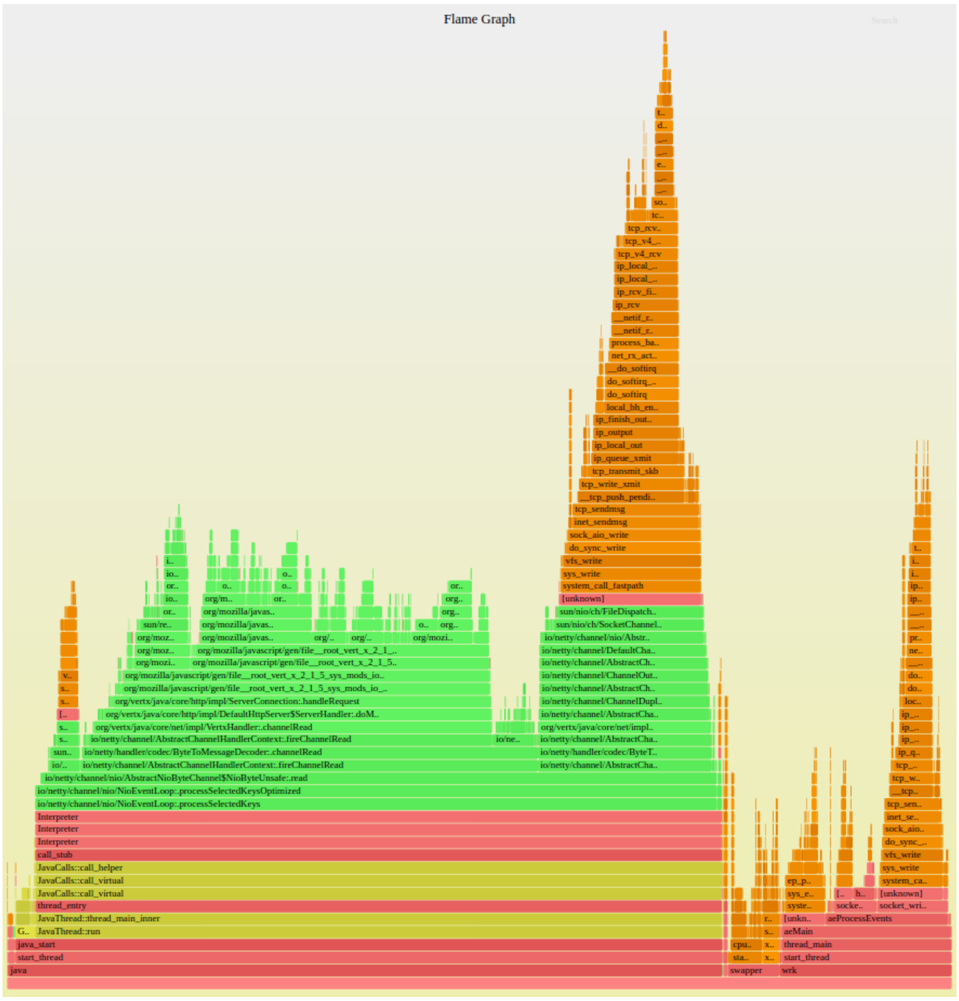


Figure 2.7: An example flame graph (taken from [FG17]). Green = Java, yellow = C++, orange = kernel, red = user and system functions.

2.5.1 Online Judging Systems

Kattis was developed in 2005 at KTH in Stockholm [EKN^{+}11]. First utilized as an automatic assignment checker for different courses at the university, it has later expanded a lot, and is now one of more well known and mature OJs available. Kattis is widely used for hosting programming competitions such as IDI Open [IDI17], removing the burden of manually checking the correctness of submissions.Kattis supports 15 languages and host over 1000 problems, ranging from very simple to more challenging.

UVa Online Judge is probably the oldest known online judging system. First developed by a student at the University of Valladolid in 1995 [RML08]. Later, it was expanded for use in the ICPC programming contest [ICP17]. Its problem archive has over 4300 problems and user registration is open to everyone. There are currently over 100000 registered users.

HackerRank was created by Vivek Ravisankar and Hari Karunanidhi as a way of alleviating some of the interview process when hiring new software developers [HR17]. They built automatic code challenges that developers could participate in promoting meritocracy.

2.5.2 Crowdsourcing Sites

Top Coder is perhaps the most popular crowdsourcing site available [TC17].The site has more than 1,000,000 members and offers over 7,000 challenges per year. It is used by companies like Amazon, Facebook, IBM, and Microsoft for crowdsourcing real-world problems. Money prizes are often awarded to the best solutions.

RecSys Challenge [RS17] is a crowdsourcing competition that aims to solve dierent problems in recommender systems. It began in 2010 and the top three winners of each challenge are invited to present their solution at the RecSys Conference, as well as receiving a money prize.

Chapter 3

CMB Improvements

This chapter describes the implementation of the changes and new features added to the CMB system. In section 3.1 the new low-level statistics feature is described, section 3.2 describes the Bash scripts converted to Python, section 3.3 how the uploading of single source files was implemented, section 3.4 includes the improvements made to the stability of the system , and finally, section 3.5 contains some general architectural improvements.

3.1 Generating Low Level Statistics

The aim of providing more low level statistics is to help the user analyze their programs, to find potential performance weaknesses/bottlenecks or bugs. It also helps in understanding why different implementations behave as they do, and what their strengths and weaknesses are. In this regard, the type of information wanted is often dependant on the specific problem, architecture, and implementation details. Thus, in accordance with the goal outlined in section 1.2 we try to provide some general statistics that are useful in a number of situations.

The profiling implementation required changes in all parts of the system, from the actual profiling happening on the backend, to the UI/frontend where the data is presented to the user. Following is a more in depth look at the implementation details.

3.1.1 Backend

An important consideration was whether the profiler should run in parallel to the normal program execution, or if it should be done separately. Because of the wish to make profiling optional, both per problem and for the user, and because we wanted to keep the submissions from the previous system version relevant, the profiling is not done together with normal execution. This way it is easy to control when profiling occurs, and no side-effects are imposed on normal execution.

In the script managing the profiling the program is executed in a background job, with two different profilers attaching to the process for monitoring. The first one is a ”poor man’s profiler” (section 2.?), which continuously collects stack-traces with GDB. This is done using a simple while-loop, which executes as long as the process is running. The stack-traces are then later converted to a text file containing the call-stacks, in the input format required by flame graph. Lastly, the flame graph is generated.

```
gdb -ex "thread apply all bt"   % Print backtraces
    -batch                      % Not interactive mode
    -p $pid                     % Which process
```

Listing 5: Collecting stack-traces with GDB.

The second profiler is a simple event counter using `perf`. This command uses performance counters, CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branch-misses. These statistics are added to a JSON object, and written to standard output.

```
perf stat -o output.txt -x:            % Output file
    -e cache-references,cache-misses,  % What to count
    branches,branch-misses
    -p $PID                            % Which process
```

Listing 6: Performance counters with perf.

When `perf stat` is attached to a running process it is required to hit Ctrl+C to stop the command even though the process has terminated. This is troublesome to do in a script so a patch was applied to fix this. Now the `perf stat` command terminates alongside the process it has been attached to. This patch is detailed further in appendix B.

3.1.2 Server

The server copies the flame graph from the backend, stores the flame graph locally, and inserts the other data in the database. The flame graph is not stored in the database because storing images in databases is generally not recommended. It is instead stored in the file system along with the other submission files. The flame graphs produced are relatively small in size (less than 200 KB), so storing 1000 flame graphs would take up less than 200 MB of space (our current server has 7.5 GB of space and $\sim$ 3000 submissions).

When extending the database to accommodate profiling information for a given submission, we had several ways of implementing it. The first decision was whether a new table was needed, or if adding columns to the submissions table sufficed. Because we decided to enable multiple profiling runs per submission (as is done with runs), we used the former option. A new ”profilings” table was created, resembling the run table but with the `profiling_data` attribute instead of time/energy. One could have added multiple columns (one for each performance metric), but we wanted a single JSON column which contained all of the data. This solution has the advantage that if some other metrics are added in the future, the database doesn't need to be migrated. Unfortunately, JSON columns are only supported in MySQL versions above 5.7 (ours is 5.6), so instead we used a String column, storing the data in the JSON format, but as a String. The new database scheme is presented in figure 3.1.

Note that the `profiling_data` attribute also lies in the submissions table. This is because while multiple profiling runs are possible only the last one is shown to the user, and this last result is stored along the submission for easier retrieval. This is the same method used for the different runs, where the latest time and energy measurements are also stored in the submissions table.

For providing the profiling data to the frontend, only one new endpoint were added to the Flask REST API. This endpoint serves the flame graph from the file system, if it exist. The other profiling data also lies in the submissions table, which already has an endpoint for retrieval, and is automatically retrieved when the user enters a problem view.

3.1.3 Frontend

The accepted programs table in the problem view was extended with two new columns, one with buttons for starting profiling runs, and one with buttons for displaying profiling data as seen in figure 3.2. A new HTML partial (a frontend view) was added to show the profiling information. This view is entered when the user presses the show profiling button, and is associated with a given submission. When entered the frontend retrieves the flame graph from the server, and displays it alongside the profiling data found in the submission data. Figure 3.3 displays the profiling view.

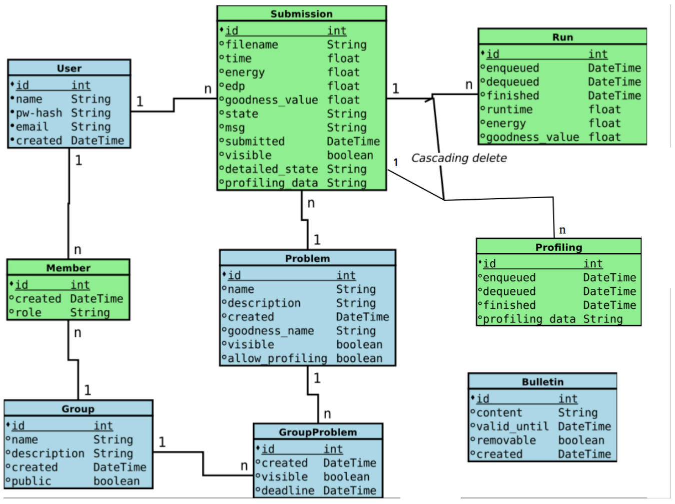
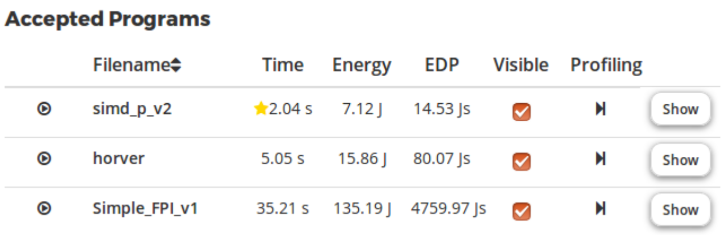


Figure 3.1: New database schema. Profiling table added, the `allow_profiling` attribute in the Problem table and the `profiling_data` in the Submission table


Accepted Programs

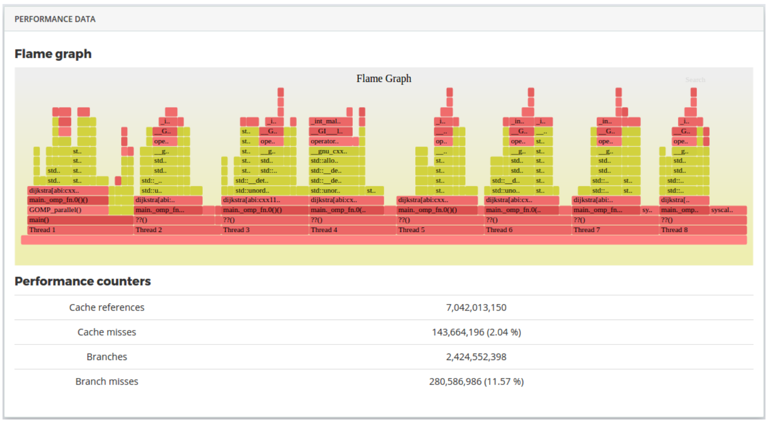
	Filename	Time	Energy	EDP	Visible	Profiling	
▶	simd_p_v2	★2.04 s	7.12 J	14.53 Js	✓	⏭	Show
▶	horver	5.05 s	15.86 J	80.07 Js	✓	⏭	Show
▶	Simple_FPI_v1	35.21 s	135.19 J	4759.97 Js	✓	⏭	Show

Figure 3.2: Button for profiling a submission, and a button for displaying the profiling data.

3.2 Porting Bash Scripts to Python

The bash scripts used in the system have consistently been the most difficult parts to debug. Bash scripts don't give helpful error messages (if at all), may hide bugs, and can be very challenging to understand without a lot of experience and helpful comments. This is especially true when some scripts starts other scripts over SSH. Therefore, it has been suggested that they be converted to Python wherever possible.

The server previously had two bash scripts: a compile script for checking that a submission compiles correctly, and a run script for copying files and starting the backend run script (which executes the submitted program on the board). In a server otherwise written exclusively in Python, these were cumbersome to deal with and quite unnecessary. When rewriting their functionality with Python, they can easily be added to the testing framework, which greatly helps with debugging. Another added benefit in Python is exception



Performance counters

Cache references	7,042,013,150
Cache misses	143,664,196 (2.04 %)
Branches	2,424,552,398
Branch misses	280,586,986 (11.57 %)

Figure 3.3: Page for displaying profiling data.

handling, which is not possible in bash. Previously, one could not distinguish the errors produced within these scripts, e.g., a compilation fail vs. various server faults that may occur in the same moment. This is now correctly distinguished, which in turn also gives better feedback to the user [1].

The bash script responsible for executing the user submitted code on the board was not rewritten in Python. This was mainly because we did not want to change the environment in how code is being executed. Right before the user program is run, the temperature of the CPU is set to a predefined constant to ensure a fair starting point when measuring energy consumption. It is probably not impossible to create the same conditions via Python, but extensive testing would be required to be sure [2].

3.3 Allowing Different File Uploads

From the user study conducted in [Mag16] it was found that the format and structure of file uploads was unclear, and that uploading of single c/cpp source files was requested. The requirement that uploaded files must be in a zipped folder was mainly because of simplicity, and minimizing file size. These however, are not really concerns the user should need worry about, and zipping submitted files before they are stored in the server is not difficult. The system now accepts single C/C++ files, creating a zipped folder containing the single file and sending the zipped folder to the server. This was a simple solution, only requiring small changes to the front-end code, which improves system usability.

[1]Preventing server faults of being mistakenly reported as compilation errors.
[2]Another option is of course to create new starting conditions, thus invalidating all previous runs.

3.4 Stability Improvements

3.4.1 Push in Thread

The system originally consisted of three separate processes: the Gunicorn server, the `push.py` script, and the frontend. The most commonly failing part was the `push.py` script, with which we had much struggles from time to time. When the server receives a request to run a submission, it stores this information in a job queue. The push script is a process that requests objects from the job queue, and sends jobs to be executed on the backend. As mentioned this process regularly failed, mainly when it could not contact the server [3]. Putting this script in its own process seemed unnecessarily complex, so we proposed a simplification where it is run in a thread by the server.

When the push script is a thread managed by the server, it simplifies several tasks. To begin with, having one less process is easier to handle at system start/restart. Also, if the push script crashes on an unexpected error (temporary connection error to the backend to give an example), there is no harm done as the script simply retries at next scheduling. The thread is spawned every two seconds after the termination of last thread.

Immediately after this change was implemented, a considerable improvement in overall system stability was seen. We have had almost no incidents of server crashes since, where it previously often crashed at server restarts, which happens at 2 am when important security updates are needed.

3.5 General Architectural Cleanup and Improvements

3.5.1 Removing the `compiledSolutions` Folder

When uploading a submission the server first stores the submitted files as a zipped folder, and then checks if the code compiles before sending it to the backend for execution. If it compiles correctly it stores the source files unzipped in a folder called `compiledSolutions`. This folder exists for legacy reasons from when cross-platform compiling was done. This entire folder is now redundant as the submission files are stored in the problems directory anyways. Furthermore, because the server always checks compilation, there is no need to keep track of which programs compiled without error. When the server sends files to the backend for execution, it temporarily unzips the submission folder containing the files. Removing this folder completely saves space on the server, and makes it easier to manage submission files (as they are only stored at one place).

[3]Which it in theory always should, but when restarting the system (which happens every now and again) the first request sent would sometimes fail. Also it had no restart functionality, so it was susceptible to momentary network failures.

3.5.2 Improvements to the backend run-script

Some improvements has been made to the backend run-script. This is the script responsible for executing and measuring the user-submitted programs on the Odroid-XU3 board. The way energy is measured is by continually noting the power levels, and afterwards integrating in the time interval the program was executing. Previously, a function in the bash script parsed the text file containing the power measurements, removing all measurements not in the given time interval. This was unnecessary as the Python script doing the integration has to parse the file anyway, and can easily handle removing the first and last lines (which are outside the interval). This was therefore moved to be handled by the Python script.

The server executes the run-script over SSH, which writes the results to stdout. Therefore, all unwanted outputs and errors from intermediate commands in the script are redirected to `/dev/null` [4]. This has led to the script being difficult to debug, because all errors and wrong output from most commands are not seen. To alleviate these difficulties, the script was strengthened with simple debug functionality. The idea is to change the behaviour if an extra dummy argument is passed to the script. Individual stream redirection is removed from the script, and instead both stdout and and stderr are immediately redirected to `/dev/null`. The few commands we actually *want* the output from are redirected to a new stdout stream, which was redirected to stdout before it itself changed. If an additional dummy argument is passed all commands show error messages, making debugging a lot easier. In listing 7, this functionality is displayed. Note that the file descriptors *1* and *2* are the normal stdout and stderr streams in Bash. Two new file descriptors (*3* and *4*) take their place.

3.5.3 Database Dump

The admins of the CMB team wanted a quick way to inspect the database, to gather different data to create statistics. An extra page in the admin panel was added that prints the complete contents of the database as comma separated values, excluding certain columns like password-hashes. This gives the admins the ability to further process the data according to their needs, for example to make a graph of submissions over time for a certain problem.

[4] Redirection to `/dev/null` is a common way to hide an I/O stream.

```bash
# Stdout and stderr are redirected to /dev/null if
# no argument is passed (= not debug mode)
exec 3>&1                       # 3 is the new stdout
exec 4>&2                       # 4 is the new stderr
if [ $# -gt 0 ]                 # If more than 0 args
then
        DEBUG="True"
else
        DEBUG="False"
        exec 1>/dev/null        # Normal stdout is hidden
        exec 2>/dev/null        # Normal stderr is hidden
fi

# Later, if output is wanted regardless of mode
command 1>&3
```

Listing 7: Debugging functionality in a Bash script. If no extra arguments stdout and stderr are silenced.

Chapter 4

Results and Analysis

This chapter has two sections. The first describes how the new feedback can be used to tune performance of submitted programs, and is divided into two case studies involving problems hosted on the CMB system. The next section (4.2), is an analysis of the performance and energy efficiency of OpenMP programs on the Odroid XU3 board, specifically how setting the CPU affinity manually with the new features of OpenMP 4 can be used to distribute work among the different cores of the ARM big.LITTLE heterogeneous multicore.

4.1 How the New Metrics can Assist Performance Tuning

In this section concrete examples of how the generated profiling information can be used in performance tuning are presented. We perform a case study of the ”Text Search Problem”, and ”The Shortest Path Problem” in which different metrics help highlight different issues.

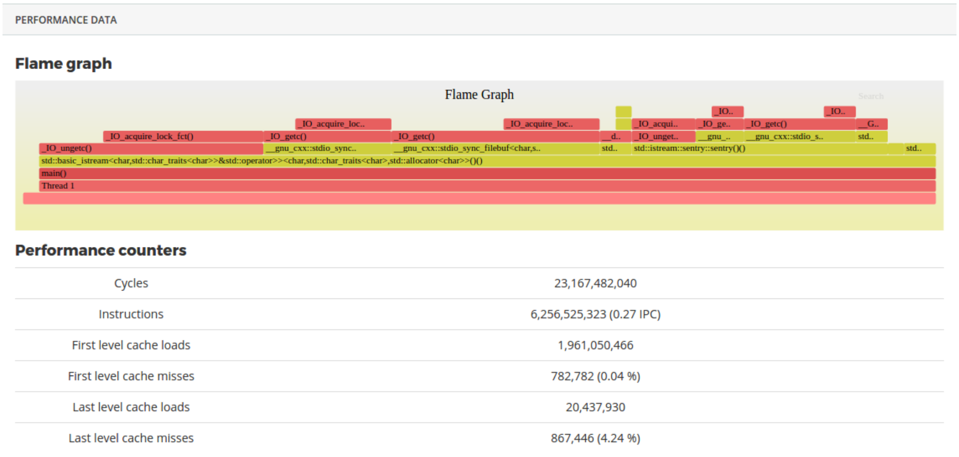
4.1.1 Case study: Text Search

In the text search problem the task is to count number of occurrences of different words in a large text file. The problem description goes as follows:

> Find words in a text. In this problem you are going to count the number of occurrences of different words in a text file (stdin). Letter case should not be differentiated, and punctuation ignored. In other words, upper case should be

> transformed to lower case, and all letters not in 'a' .. 'z' should be ignored. As an example, if one of the search words is "frank" and the text "My name is Frank, Fraaaank! #fRaNk #frankistheman" the count is 2.

An initial solution to the problem is presented in Listing 8. This solution had a running time of 11.74 seconds, and used 33.85 Joules. In figure 4.1 the corresponding profiling data is showed. We immediately see that most of the time is spent in various IO functions, stemming from the calls to `std::basic_istream<...>operator<<`, which corresponds in our program to the calls `cin << word`. Further up the call-stack we see that the program spends a lot of time in a function called `_IO_acquire_lock_fct()`, actually 30 out of the total 57 samples culminated in this function. It turns out that this is a C library function, which locks standard input/output from other accesses. The standard C++ streams (cin, cout, cerr, etc.) are synchronized to the standard C streams (stdin, stdout, and stderr) by default after each input/output operation. This makes it possible to freely mix C++ and C I/O, and also guarantees thread-safety for the C++ streams, i.e., no data races. It is however, possible to turn this synchronization off by a call to `std::ios::sync_with_stdio(false)`.


Flame Graph
_IO_acquire_lock_fct()
_IO_getc()
_IO_ungetc()
std::basic_istream<char,std::char_traits<char>>&std::operator>><char,std::char_traits<char>,std::allocator<char>>()()
main()
Thread 1

PERFORMANCE DATA

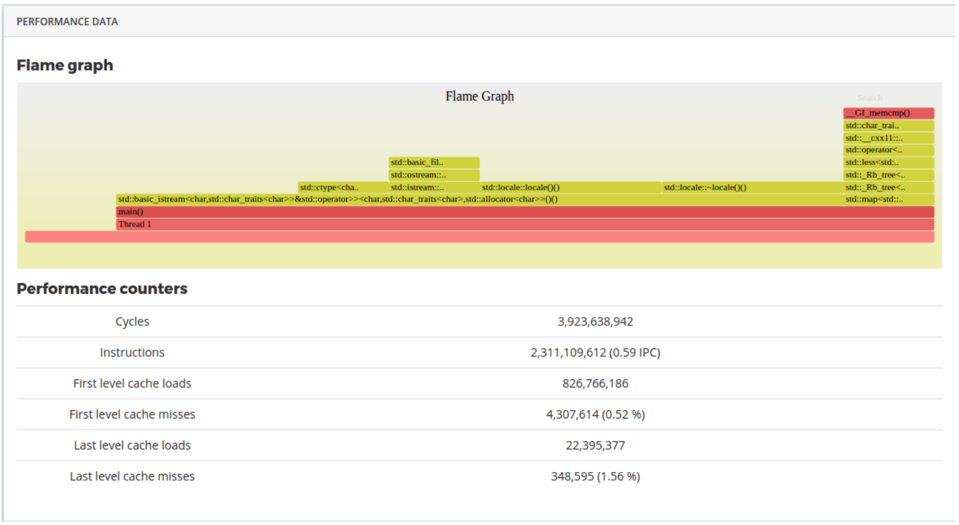
Flame graph

Performance counters

Cycles	23,167,482,040
Instructions	6,256,525,323 (0.27 IPC)
First level cache loads	1,961,050,466
First level cache misses	782,782 (0.04 %)
Last level cache loads	20,437,930
Last level cache misses	867,446 (4.24 %)

Figure 4.1: Profiling data for the initial solution.

After inserting a call to this function in the beginning of `main`, a huge speedup appears. It now runs in 1.85 seconds and uses 5.63 Joules. Indeed, the impact of unsynchronizing the C++ streams when a program spends a considerable percentage in I/O is so large that this simple trick is mentioned in the system's "How To"-page. In figure 4.2 the new profiling data is shown. Still, most of the time is spent in reading standard input, but it may be difficult to optimize this further. The last part of the call-stack is the call to `std::map<...>find()`. This function again call a function called `std::Rb_tree<...>find()` reminding us of the way a map is implemented in C++, namely with a red-black tree [GS78]. Red-black trees features insertion, deletion, and search in $O(\log n)$ time. As the map usage in our program is dominated by search calls, only inserting the words on the first line, and searching for existence for all remaining words, a data structure with a better search performance would be preferable. The C++ standard library provides *unordered map* as an alternative to map, which is an unsorted map (map keeps all elements sorted), implemented as a hash-table. Search, insertion, and removal of elements have an average time complexity of $O(n)$.



PERFORMANCE DATA

Flame graph

Performance counters

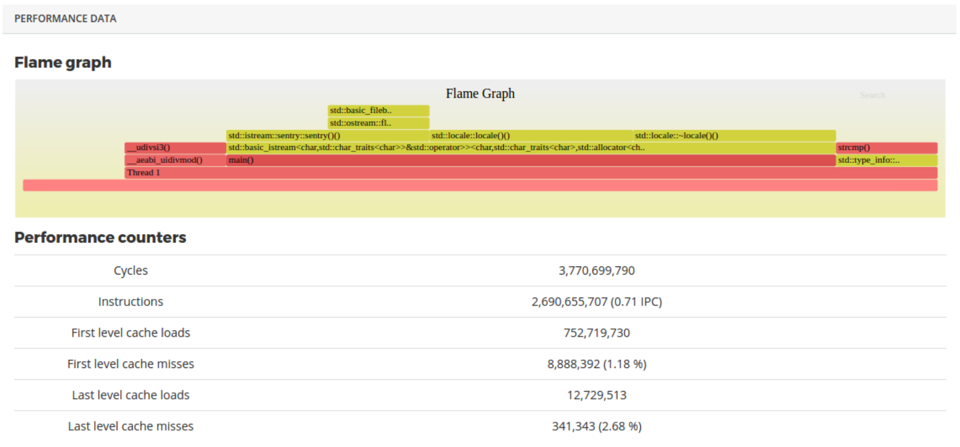
Cycles	3,923,638,942
Instructions	2,311,109,612 (0.59 IPC)
First level cache loads	826,766,186
First level cache misses	4,307,614 (0.52 %)
Last level cache loads	22,395,377
Last level cache misses	348,595 (1.56 %)

Figure 4.2: Profiling data after removing the synchronization with the C streams.

The program now runs in 2.04 seconds... a 10 % slow down! This probably means that the number of words to search for is very small[1]. The programs uses around 5 % more instructions, but also have a higher IPC. It has roughly the same amount of last level cache misses, but almost 4 times as many 1st level misses. This indicates that when the element count is low, a regular map has a more cache friendly memory layout, trumping the fact that search is asymptotically slower.

The observant reader will have noticed that the function filtering the strings is sub-optimal. This parses the string twice, once removing all non-alpha characters, and once transforming the remaining to lowercase. Of course, this can be done in one for loop, and results in a ~ 5 % speed up. Further optimization is probably possible, for instance using an efficient implementation of a trie as a data structure, and doing the filtering while searching the trie, but this was not tried (!) out.

[1]Indeed, it is only 15, but this information is not known to the users.


PERFORMANCE DATA

Flame graph

Performance counters

Cycles	3,770,699,790
Instructions	2,690,655,707 (0.71 IPC)
First level cache loads	752,719,730
First level cache misses	8,888,392 (1.18 %)
Last level cache loads	12,729,513
Last level cache misses	341,343 (2.68 %)

Figure 4.3: Profiling data using a hash-table instead of a red-black tree.

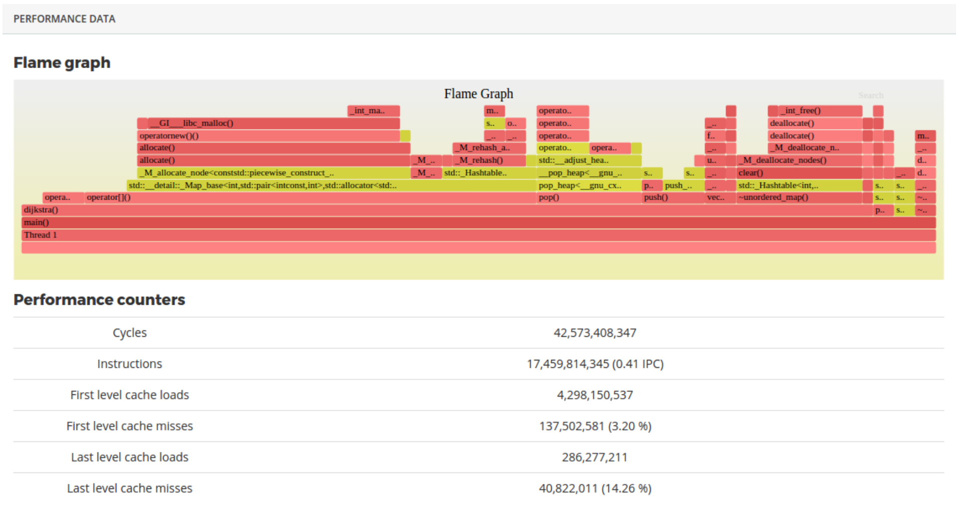
```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <map>

using namespace std;

// Filter a string. Erases all non-alpha characters
// then changes the remaining to lowercase
void filter(std::string& s) {
    s.erase(remove_if(s.begin(), s.end(),
                [](char x){
                    return !isalpha(x);
                }), s.end());
    transform(s.begin(), s.end(), s.begin(), ::tolower);
}

int main() {
    // Vector containing the words to search for
    vector<string> search_words;

    // Map with counts for each word
    map<string, int> word_counts;

    string word;

    // Reading the first line and initializing
    while (cin.peek() != '\n') {
        cin >> word;
        filter(word);
        word_counts[word] = 0;
        search_words.push_back(word);
    }
    cin.ignore();

    // Read rest of file, filter word, and check for existence in the map
    while (cin >> word) {
        filter(word);
        auto it = word_counts.find(word);
        if (it != word_counts.end()){
            it->second++;
        }
    }

    // Print counts in original order
    for (auto const& w : search_words) {
        cout << word_counts[w] << " ";
    }
}
```

Listing 8: The initial solution to the Text Search problem.

4.1.2 Case study: The Shortest Path Problem

In the next case study the shortest path problem is analyzed. The input is a list of adjacent nodes, and their distance, following the list of node pairs one is to find the shortest path between. A natural choice of algorithm to use is Dijkstra's algorithm [Dij59], which traverses through the graph along the current minimum distance until the target node is reached.

In listing 9, the initial implementation of Dijkstra's algorithm is shown. When submitted to CMB it has a running time of 20.42 seconds, and uses 56.11 Joules. In figure 4.4 the corresponding profiling information is shown. What is immediately evident is how much time is used in retrieving elements from a map (the large `operator[]()` call 49 %), and the map destructor (`~unordered_map()` 13 %). Combined over 60 % when the only map used is the one keeping track of what the previous nodes, needed when reconstructing the path when a solution is found. It seemed reasonable to use a hash map (C++'s unordered map) for this purpose as we want a node to node index of parents. However, when the nodes represented as integers, one can use a vector for the same purpose, where the index represents the child node. The downside is of course that all nodes in the graph need to be initialized, i.e., the vector must contain all nodes whereas the map can contain only those visited up until a solution is found.


PERFORMANCE DATA

Flame graph

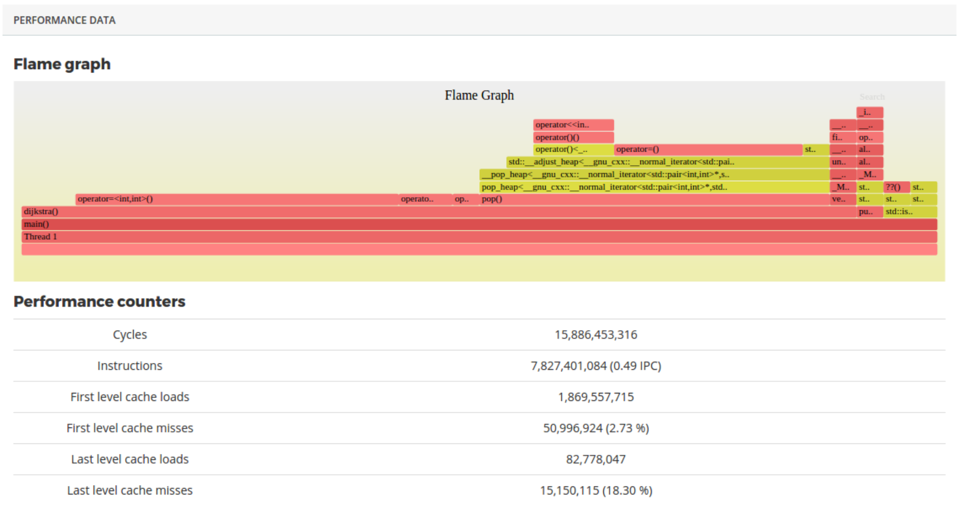
Performance counters

Cycles	42,573,408,347
Instructions	17,459,814,345 (0.41 IPC)
First level cache loads	4,298,150,537
First level cache misses	137,502,581 (3.20 %)
Last level cache loads	286,277,211
Last level cache misses	40,822,011 (14.26 %)

Figure 4.4: Profiling data for the initial shortest path solution.

After switching to a vector the running time and energy use drops to 8.29 seconds and 23.70 Joules. From the related information in figure 4.5, it seems now that most of the CPU time is used in various functions related to the priority queue. We see the `pop()` operation, and the comparing function (`operator=<int, int>()`) taking up most time. The priority queue (a minimum heap) is a crucial part of the algorithm, and as such is expected to consume a lot of resources. There might be some clever ways of improving this heap further, perhaps by joining this data structure with the visited list, and/or the main graph.

Dijkstra's algorithm is not easily parallelized. It can be divided in two, with one instance searching from the source node, and the other searching from the sink, but further parallelization has proven quite difficult [2]. Nevertheless, the shortest path problem asks for the shortest path between many different source/sink pairs, and these are trivial to parallelize. When we parallelize the algorithm using OpenMP, the running time and energy use drops by a factor of four.



PERFORMANCE DATA

Flame graph

Performance counters

Cycles	15,886,453,316
Instructions	7,827,401,084 (0.49 IPC)
First level cache loads	1,869,557,715
First level cache misses	50,996,924 (2.73 %)
Last level cache loads	82,778,047
Last level cache misses	15,150,115 (18.30 %)

Figure 4.5: Profiling data after switching to use a vector for storing the previous nodes.

[2]In [JAE+12], Jasika et al. investigate the performance gain of several parallel implementations of Dijkstra's algorithm, finding an average speedup of only 10 %.

```
bool dijkstra(int start, int end, const std::vector<distVec>& graph,
              std::vector<int>& path) {
    // Dijkstra's algorithm with priority queue
    const int N = graph.size();

    // For keeping track of visited nodes
    std::vector<bool> visited(N, false);

    // Holds current known best distances to nodes
    std::vector<int> distances(N, std::numeric_limits<int>::max());
    distances[start] = 0;

    std::unordered_map<int, int> prevNodes;

    // The priority queue. Holds a <pri, node> pair.
    typedef std::pair<int, int> P;
    std::priority_queue<P, std::vector<P>, std::greater<P>> pq;
    pq.push(std::make_pair(0, start));

    int current, current_dist;
    while (!pq.empty()) {
        std::tie(current_dist, current) = pq.top();
        pq.pop();

        if (current == end) {  // Found path
            while (current != start) {
                path.push_back(current);
                current = prevNodes[current];
            }
            // Need to reverse the path
            std::reverse(path.begin(), path.end());

            return true;
        }

        visited[current] = true;

        int neighbor, dist_to_neighbor;
        for (const auto& tmp : graph[current]) {
            std::tie(dist_to_neighbor, neighbor) = tmp;
            if (visited[neighbor])
                continue;
            int alt_dist = current_dist + dist_to_neighbor;
            if (distances[neighbor] > alt_dist) {
                distances[neighbor] = alt_dist;
                prevNodes[neighbor] = current;
                pq.push(std::make_pair(alt_dist, neighbor));
            }
        }
    }
    return false;  // No path exists
}
```

Listing 9: The initial implementation of Dijkstra's algorithm.

4.2 Analysis of OpenMP programs on the Odroid-XU3

4.2.1 Choosing CPU affinity manually

With the new thread affinity features in OpenMP 4 [OMP13], it is possible to choose more specifically where threads should be executed. By setting the environment variable `OMP_PLACES`, which is a platform-specific list of execution units (e.g., CPUs), one can later use the new `proc_bind` clause for affinity strategy. There are three arguments for this clause. The first one is `master`, making all threads execute on the same place as the master thread. `close`, which puts the new thread positions close to the master thread. And `spread`, where the threads are spread out as much as possible. Figure 4.6 shows how these different arguments to the `proc_bind` clause will distribute two threads among four places.

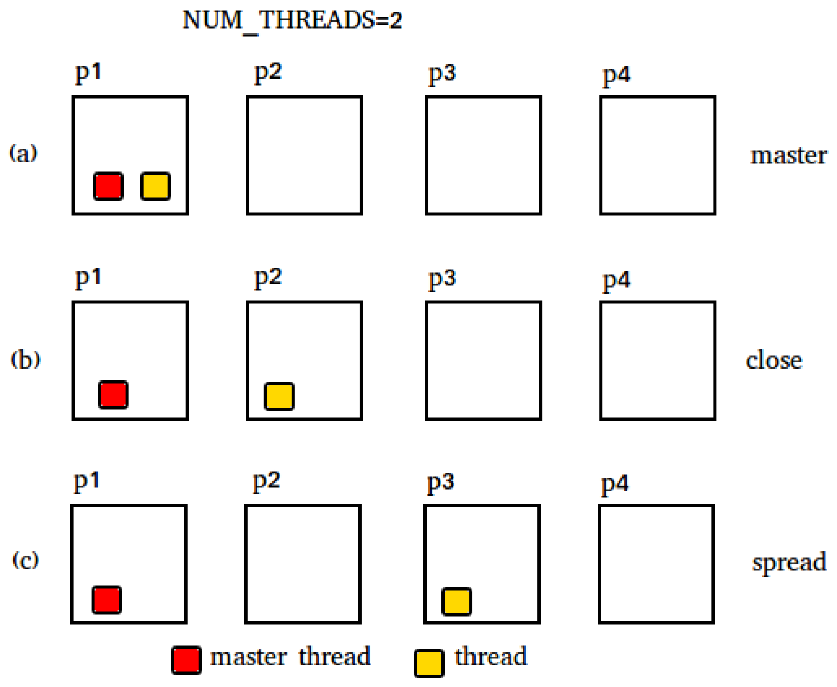


Figure 4.6: Thread affinity specifications in OpenMP 4. In (a) all threads are in the same place as master, in (b) close to master, and in (c) equally spread among places.

By using these it is now possible to choose which CPUs does what. As an example we created a small program containing two different workloads, one low intensity where the CPU is sleeping most of the time, and one high intensity with expensive math operations 10. The `OMP_PLACES` variable is set to 0,1,2,3,4,5,6,7[3] so that in the first section the

[3]On the Odroid-XU3 this corresponds to the 8 CPUs, where 0-3 are the smaller Cortex-A7, and 4-7 are the

threads are distributed in all cores, and in the second section in the four first (0, 1, 2, 3) [4].

So what happens if we don't set thread affinities, but instead let the OS choose what work goes where? By removing the `OMP_PLACES` variable, the `proc_bind` clause and changing to 4 threads in the first section, we let the OS decide how to distribute work. Interestingly, we now get the exact same results! The high intensity workloads are done on the large cores, while the low intensity workloads are done on the smaller ones. The Global Task Scheduling (GTS) distributing work on the big.LITTLE architecture is capable of figuring out what kind of work is being done, and dynamically schedules threads to appropriate CPUs. The GTS keeps track of load history as each thread runs, and uses the history to anticipate the performance needs of the thread next time it runs [ABL17].

larger Cortex-A15. Setting `OMP_PLACES=cores` does the same thing.

[4]In this example it's important to also set the environment variable `OMP_NESTED=TRUE` so that the sections are allowed to spawn internal threads.

```
#include <iostream>
#include <stdlib.h>
#include <omp.h>
#include <sched.h>
#include <unistd.h>

// A low intensity workload, where the CPU is
// sleeping most of the time
int low_intensity()
{
    int result = 0;
    for (int i=0; i<5; i++) {
        usleep(1000000);  // 1 second
        result += 3*i*i + 45*i + 235;
    }
    return result;
}

// A high intensity workload, doing some
// arbitrary math
int high_intensity()
{
    int result = 0;
    for (int i=0; i<500000; i++) {
        result += (3*i*i + 45*i + 235) % (1 + i * 50);
    }
    return result;
}

int main()
{
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            #pragma omp parallel num_threads(8) proc_bind(close)
            {
                if (sched_getcpu() > 3) {  // If we are at a big CPU
                    int r = high_intensity();
                    #pragma omp critical
                    std::cout << "High intensity at CPU " << sched_getcpu() << "\n";
                }
            }
        }
        #pragma omp section
        {
            #pragma omp parallel num_threads(4) proc_bind(close)
            {
                int r = low_intensity();
                #pragma omp critical
                std::cout << "Low intensity at CPU " << sched_getcpu() << "\n";
            }
        }
    }
}
```

Listing 10: OpenMP example setting thread affinity with the `proc_bind` clause. There are four high and four low intensity workloads, distributed to the Cortex-A15 and A7 cores.

4.2.2 Performance of The Nasa Parallel Benchmarks on the Odroid-XU3

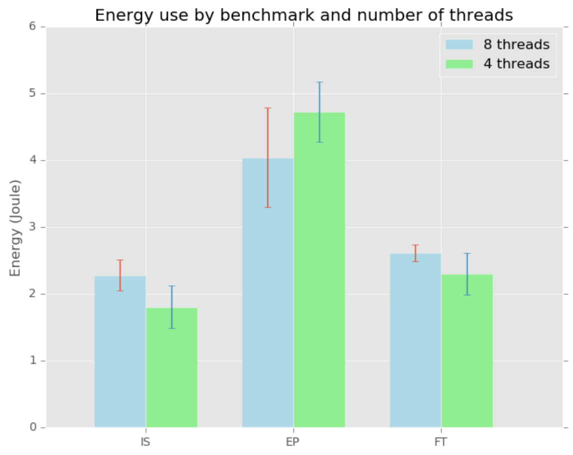
The NASA Advanced Supercomputing (NAS) division Parallel Benchmarks (NPB) is a benchmarks suite, developed in 1991, targeting performance evaluation of highly parallel supercomputers. They are made and maintained by NASA. The benchmarks originally contained five kernels and three pseudo-applications, and has later been expanded to include more benchmarks. It has several reference implementations, including MPI, OpenMP, and serialized versions.

In this section we explore the possibility of ”outperforming” the GTS on a selected subset of the NPB. More specifically, can we distribute work manually among the cores in a better performing way (with regards to time and energy use) than the GTS? We look at three benchmarks: IS, an integer sorting algorithm, EP, the ”embarrassingly parallel” benchmark, and FT, a 3D discrete fast Fourier transform [5].

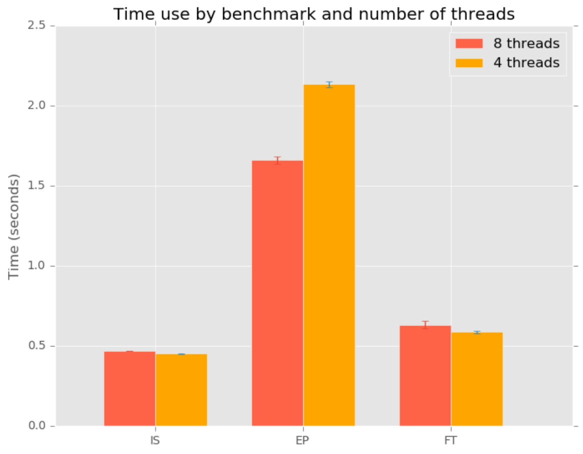
As these are all benchmarks with heavy computation, and little to no low intensity work, our first strategy is analyzing the performance if all work is restricted to the four large cores. Figure 4.7 shows the average energy and time usage for the three benchmarks using four or eight threads. A performance increase is seen in the IS and FT benchmarks, for both time and energy use, while the EP benchmark performs worse. To find out what causes these results, a further inspection of how the OpenMP threads do work in each benchmark was needed. By instrumenting the code to notify which CPU is currently working we get a better picture of how the GTS distributes work. It turns out that both the IS and FT benchmarks have a relatively few number of workloads, where each one is computationally heavy. This resulted in each of the eight threads always being assigned to the big cores, which in turn means that having eight threads, as opposed to four, adds some unnecessary overhead of spawning and joining. In the case of the EP benchmark, a much larger number of workloads were present, and the small cores were occasionally contributing. This particular benchmark has a very high parallelism, so restricting the number of cores only hindered performance.

It is evident that choosing the number of OpenMP threads, and how to distribute them on a heterogeneous system, is not a straight forward task, and that deep knowledge of the specific problem is needed. The default in OpenMP is to create as many threads as there are CPUs. Coupling this with the GTS, which is capable of dynamically assigning work to appropriate cores, this makes for a sound strategy in most cases. Indeed, the performance gain by our strategy in the IS and FT benchmarks were very slight, and fiddling with thread allocations is more likely to cripple performance if care is not taken. It would be interesting to examine if better performance gains can be achieved in benchmarks suites specifically made for heterogeneous systems, such as the Rodinia suite [CSB$^+$10]. If there are a greater contrast between workloads, some low intensity and befitting the smaller cores, manually assigning workloads in OpenMP may be an advantage.

[5] The implementation is included in the digital appendix.



(a) Energy use.



(b) Time use.

Figure 4.7: Energy and time use of three NPB benchmarks on Odroid-XU3

Chapter 5

Stability

This chapter takes a look at the measurement accuracy of the new system (section 5.1), and how the results compare to the results of the testing done by Follan and Støa in [TF15]. In section 5.2 the overall system stability in the time of the author working on the CMB system is measured, from January to June 2017.

5.1 Measurement Accuracy

When Follan and Støa first developed the CMB system, it was a priority to make the time and energy measurements as accurate as possible to ensure fairness among submissions. To that end, a thorough inspection was undergone, where issues with cache usage were identified and fixed. This section tries to reproduce their results, as a general system examination, but also for evaluating potential repercussions the new changes may have had on measurement accuracy.

5.1.1 Experiment setup and Methodology

Trying to replicate the results of Follan and Støa, we use as close to the same setup as we could. The tests are run on the Odroid-XU3 board, which besides having an updated linux kernel, uses the same software to run programs. Before each run the core temperature is set to 60 degrees Celsius, and cache is cleared with the following command:

```
sync && echo 3 > /proc/sys/vm/drop_caches
```

To calculate the mean and relative standard deviation, the following equations are used:

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{5.1}$$

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{5.2}$$

$$RSD = \frac{s}{\overline{x}} \tag{5.3}$$

N is the number of samples, x_i is sample i, and s the standard deviation. The RSD can be used to compare standard deviation when mean running time and energy use differs.

5.1.2 Results and discussion

The time and energy used was measured for a naive solution to the shortest path problem, the same solution used by Follan and Støa. The program was run every 15 minutes over a period of several days. Figure 5.1 shows the results of these measurements. One thing immediately stands out. There was 1.5 second increase in running time for 12 hours between the 2. and 3. of June. The cause of the run time (and energy) spike is not understood. A potential reason can be background running processes taking up CPU time, but they could not be identified. With an RSD of 0.7 % this is a slight increase in variation compared to the reported RSD of roughly 0.2 % reported by Follan and Støa. Without the spike we get an RSD of 0.15 %.

It is also not fully understood why the same program now takes 8 seconds longer. The only change to how the system executes user submitted programs is the updated linux kernel, the same compiler with the same flags are used in the same environment. Indeed, we see a change in run time for many of the previous submissions for several different problems, some taking longer and some with a faster running time.

5.2 System stability

A cron job is run run every 15 minutes on the server, notifying the CMB team when parts of the system is down. This testes the three parts of the system: the Gunicorn server, the board, and the Python push script. If any of these are offline, an email is sent so appropriate actions might be taken to restore the failing part, and the maintenance page is automatically displayed at the web page. Using these data a plot of the systems down

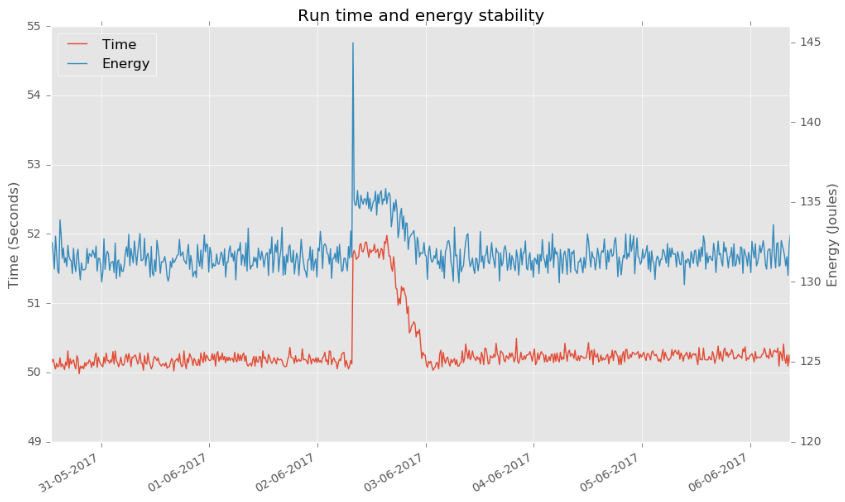


Figure 5.1: Measuring accuracy of a solution to the shortest path problem.

times can be made. Figure 5.2, the system stability from 1. January 2017 to 1. June 2017 is shown.

The end of the push graph in figure 5.2 represents the integration of the push script into the Gunicorn server as further detailed in section 3.4.1. As mentioned, this completely removed the troubles with the push script, and the only system down time since, not as a result of scheduled maintenance, is when the board was down on the 4th and 16th of April. The board going offline is usually because of momentary down time of the University network, and the problem often solves itself when connectivity resumes.

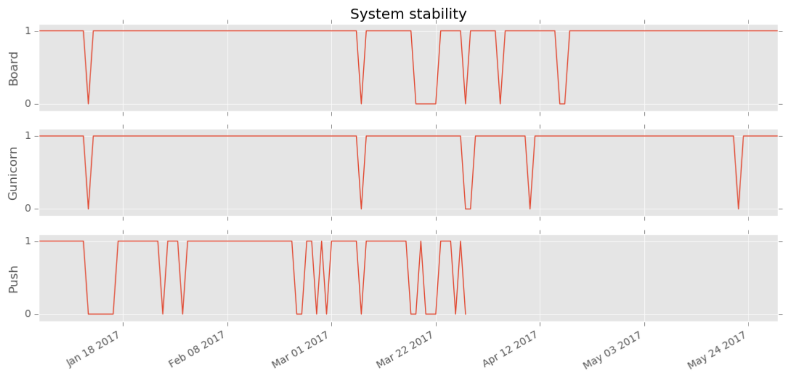


Figure 5.2: Stability of the Odroid XU-3 board, Gunicorn, and Push script. 1 means online, 0 offline.

Chapter 6

Discussion

In this chapter we discuss some of the choices presented when implementing the profiling and some of its limitations, difficulties, and what we would have liked to see.

6.1 Profiling

To provide a good analysis for further performance tuning, we wanted the profiler to provide certain data. Specifically, we wanted to provide an accurate call graph, showing where the program spends CPU time, and where the location of potential bottlenecks. It was also desired to show how the different cores are utilized, particularly for parallel programs, where the distribution of work in the heterogeneous environment is difficult to visualize. Additionally, more information about energy usage, for example showing the energy consumption over time. Following, we discuss the method used for profiling, and its advantages and limitations.

6.1.1 Choosing the Profiler

There were many factors to consider when choosing a profiler. First and foremost, it needs to be supported on the Odroid XU3 card (for obvious reasons). Secondly, user experience is important, so the profiler should give results back in a reasonable amount of time. Having a relatively short profiling time is also essential with regards to queue buildup. There is currently only one backends so queue buildup is a very real concern (especially if profiling and execution is happening on the same one!). Furthermore, to keep the integrity of the existing database, it was a requirement that if the profiling is done together with normal execution, the overhead must be unnoticeable.

We ended up using a ”poor man’s profiler” for collecting stack-traces as no other functioning alternative was found. As the programs we wanted to profile may be very short lived, to get produce statistically relevant stack-traces very frequent probing is important. The poor man’s method of collecting stack-traces is more suited for long running processes, where frequent probing is not necessary, and you can instead analyze over a longer time-period.

Our method does not sample as frequent as a real profiler is able to, and while it’s not fully understood why, we believe it due to how they are able to attach to processes. Although the while-loop (shown in listing 11) has no sleep calls, so the only pause is the one introduced by the `kill` command (checking wether the process is still alive), it is run as a different process and it’s up to the operating system to distribute CPU-time. We tried two different ways of collecting stack-traces: GDB, and Quickstack. Quickstack have very small overheads, but the call-stacks produced were often missing the last parts of the stack, and in some cases seemed entirely jumbled (as in clearly giving the wrong order of caller/callee). Consequently, we opted to use GDB which gave better call-stacks though with higher overheads. This resulted in a slow down of ~ 3 times normal execution.

The reason for GDB’s large overhead is because it uses the `ptrace` system call to monitor processes. This system call enables a controller to inspect and manipulate the internal state of its target, but necessitates two context switches for each call and momentarily halts execution. Thus, GDB is not particularly suited for sampling stacks with a high frequency.

```
while kill -0 $PID 2>/dev/null;
do
    gdb -ex "thread apply all bt" -batch -p $PID
done
```

Listing 11: Collecting stack-traces with GDB in a while loop.

The other profiling tool used is `perf stat` for counting performance counters. With these counts, the number of key hardware events are reported back to the user. These include total instructions used, 1st and last level cache loads and misses, and cycles. The number of cache misses can be especially helpful for comparing the cache ”friendliness” of different solutions to a problem.

6.1.2 Limitations

We did not figure out a way in which to measure exactly how CPUs are utilized. The current method is capable of distinguishing threads, but whether or not it *should* could be discussed. If one simply wants to know where most of the CPU time is spent, distinguishing between threads is not necessary, and may clutter the information. On the other hand, when profiling OpenMP programs, distinguishing threads can be helpful. If the OpenMP program does not utilize native threading, the different threads will correspond only to OpenMP threads, and provide a glance of how OpenMP distributes work among CPUs. However, this is not necessarily a very *accurate* glance. How the threads are distributed among CPUs is mostly up to the OS (in our case ARM big.LITTLE's GTS) as mentioned in section 4.2.1. Nonetheless, it still provides information of what the particular threads are doing.

Chapter 7

Conclusion and Future work

This chapter concludes the thesis, evaluating which goals were met, and finally proposes further improvements to the CMB system.

7.1 Conclusion

This thesis has described several improvements and new features to the CMB system, in addition to some performance analysis of OpenMP programs on the ARM big.LITTLE architecture. The main goal was providing more detailed low level information to the user. To that end, the system can now optionally provide profiling information for successful submissions. This consists of a flame graph, a visualization of program call stacks, and performance counters such as instructions used and cache statistics. These are intended to help users, and two case studies of different problems on CMB were performed to show how the profiling information may be of assistance. In addition, several general architectural improvements have been made such as converting key bash scripts into Python, moving the functionality of the push script to the Gunicorn server, and improvements to the run script. Their implementation as well as the implementation of the low level statistics were described in detail in chapter 3. Some other minor improvements include allowing the upload of single source code files, removing the compiledSolutions folder, and an easily accessible database dump in the admin interface.

An analysis of the system stability and measurement accuracy was performed and detailed in chapter 5. This showed encouraging results, confirming the perceived increase in system stability. The measurement accuracy was high, although a slight change in the run time of some submissions had changed.

7.1.1 Thesis Goals Achievement

This section evaluates which goals were met, in accordance to the goals set in section 1.2.

1. Generate and provide low level statistics for each submission: Covered by section 3.1 where the implementation is described. A Flame Graph where the call stacks are visualized, as well as certain performance counters were added as feedback. Some of the choices made were discussed in section 6.1.

2. Port Bash scripts to Python: This goal is considered covered by section 3.2 which details the changes needed to port the Bash scripts, and why not all were converted.

3. Improve general stability of the system: The changes done to the push script discussed in section 3.4.1, drastically improved the system stability and as such covers the goal.

4. Allow single source-file uploads: This is covered by section 3.3, which describes the implementation details.

Analysis

5. Show, how the new metrics help users improve the performance of their solutions: Considered covered by section 4.1 which gives examples of how these new features can help users with the performance of their submissions, in the form of two case studies of problems found on the existing CMB system.

6. Analyze performance of parallel OpenMP programs: Section 4.2 is considered to cover this goal. The performance and energy efficiency of OpenMP programs on the Odroid XU3 board is measured in the form of the NAS parallel benchmarks, as well as an investigation of how the CPU affinity features of OpenMP 4 can be utilized to manually distribute tasks among the different CPUs of ARM big.LITTLE heterogeneous multi-core.

Other

7. Propose further improvements: This is considered covered by the ensuing section where several future improvements are proposed in an order of decreasing priority.

8. Create a integration test (if time permits): The CMB system lacks an integration test, the inclusion of which would help development, detecting potential cross-unit bugs not exposed by the unit tests.

7.2 Future Work

This section describes some potential new features that can be added to the CMB system. The master thesis of Magnussen [Mag16], presents an extensive list, including some points tackled in this thesis, and still contains very relevant suggestions. The following list is an extension following the same ordering based on priority, marked either as A (high), B (medium), or C (low).

Development and testing

A. Create an integration test: An integration test would ease development making it faster to catch potential bugs introduced. The unit tests cannot catch all bugs, and a manual test of the submission to execution pipeline is often required when changes have been made. A good integration test should work on the production server, development server, and locally, and hopefully indicate where problems lie.

Features and Improvements

A. Support for more backends: The main bottleneck of the system is currently the execution of code on the backend. If the system is to be used by a larger audience, for instance, in combination with courses at NTNU, scaling the backend by adding more boards would be beneficial.

B. Frontend user statistics: Providing more statistics to the user could increase user enjoyment and be of assistance when conducting competitions on the site. Such statistics might include global high score, total number of submissions/runs, and number of top three finishes to name a few.

B. More detailed performance information: There are a lot more information about the program execution that would be interesting to see. Specifically, more detailed information of energy expenditure, such as energy use over time, which is definitely possible as energy readings are already recorded under the whole execution. Additionally, hardware use over time (which cores are used, GPU), would be quite challenging, but very interesting.

B. More Problems: The system could always benefit from more problems being added. Especially, it lacks certain advanced problems suited for parallelizing using OpenMP, Pthreads or OpenCL. It also lacks problems where energy efficiency is a top priority, or especially difficult. One could for instance imagine problems mimicking the behaviour of real world examples where energy efficiency is important, such as smart phone apps where low and high intensity workloads are intertwined, and perhaps the time duration is set.

C. Seasons: Dividing the year into seasons, where season high scores are reset every now and then (e.g., each semester), could help keep problems fresh as new participants joining the site would not have to compete against very old submissions.

C. Support for additional languages: Adding support for more accepted languages (e.g., Python, Java) is a large task, but would greatly expand the potential user base and courses that could utilize the system. This would require changes to all parts of the system.

C. User control over the CPU affinity with OpenMP: As discussed in section 4.2.1, it is possible for the user to control which cores are used if certain run time parameters are set before execution (meaning environment variables in the backend OS). If these could be set on a per problem basis, the users could employ the affinity features of OpenMP 4. However, this requires particular knowledge of OpenMP by the users, if it is not aptly explained.

Bibliography

[A1517] ARM Cortex-A15. http://www.arm.com/products/processors/cortex-a/cortex-a15.php, Apr 2017.

[A717] ARM Cortex-A7. http://www.arm.com/products/processors/cortex-a/cortex-a7.php, Apr 2017.

[ABL17] ARM big.LITTLE technology. http://www.arm.com/products/processors/technologies/biglittleprocessing.php, Apr 2017.

[AM17] ARM Mali-628. http://www.arm.com/products/multimedia/mali-performance-efficient-graphics/mali-t628.php, Apr 2017.

[AN17] What is AngularJS. https://docs.angularjs.org/guide/introduction, Apr 2017.

[Cha16] Christian Chavez. Climbing mont blanc and scalability. 2016.

[CSB+10] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[EKN+11] Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann. Five years with kattisusing an automated assessment system in teaching. In *Frontiers in Education Conference (FIE), 2011*, pages T3J–1. IEEE, 2011.

[FFG08] Wu-chun Feng, Xizhou Feng, and Rong Ge. Green supercomputing comes of age. *IT professional*, 10(1), 2008.

[FG17] Flame Graphs Implementation. `https://github.com/brendangregg/FlameGraph`, Apr 2017.

[FPL17] Linux Kernel with perf changes for Odroid XU3. `https://github.com/fredrikpe/linux/tree/odroidxu3-3.10.y`, Jun 2017.

[FS88] Jay Fenlason and Richard Stallman. Gnu gprof. *GNU binutils.[Online]. Available: http://www. gnu. org/software/binutils*, 1988.

[FT02] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[GCL17] The GNU C Library. `https://www.gnu.org/software/libc/`, Apr 2017.

[Gre16] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[GS78] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21. IEEE, 1978.

[GU17] Gunicorn - Python WSGI HTTP Server for UNIX. `https://gunicorn.org`, Apr 2017.

[HIG94] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11. IEEE, 1994.

[HR17] Hackerrank. `https://www.hackerrank.com/`, Jun 2017.

[ICP17] The ACM-ICPC International Collegiate Programming Contest. `https://icpc.baylor.edu/`, Jun 2017.

[IDI17] IDI Open. `https://idiopen.idi.ntnu.no/open17/`, Jun 2017.

[IN17] Intel® VTune™ Amplifier 2017. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`, Apr 2017.

[JAE+12] Nadira Jasika, Naida Alispahic, Arslanagic Elma, Kurtovic Ilvana, Lagumdzija Elma, and Novica Nosovic. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In *MIPRO, 2012 proceedings of the 35th international convention*, pages 1811–1815. IEEE, 2012.

[Mag16] Sindre Magnussen. Improving system usability of climbing mont blanc - an online judge for energy efficient programming. 2016.

[MBDH99] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[MBP17] Mont Blanc Prototypes. http://www.montblanc-project.eu/arm-based-platforms, Apr 2017.

[Mid17] Uli Middelberg. How to compile a custom linux kernel for your arm device. https://github.com/umiddelb/armhf/wiki/How-To-compile-a-custom-Linux-kernel-for-your-ARM-device, Mar 2017.

[MyS01] AB MySQL. Mysql, 2001.

[NG17] NGINX. https://www.nginx.com/resources/wiki/, Apr 2017.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[OD17] Odoid-XU3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127, Apr 2017.

[OMP13] OpenMP 4.0 Specification. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, Apr 2013.

[Pat17] Hardik Patel. Added correct dts tree interrupts/nodes for hw perf events/counters odroid-xu3 dev board). https://github.com/patelhardik/kendroid_kernel_hardkernel_odroidxu3/commit/e90ac2cb272437fe40e947cbcab148b65591b06d, Mar 2017.

[PE17] Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page, Apr 2017.

[PMP17] poor man's profiler. https://poormansprofiler.org/, Apr 2017.

[QF17] Quickstack patch for ARM 32. https://github.com/fredrikpe/quickstack, Apr 2017.

[QS17] quickstack. https://github.com/yoshinorim/quickstack, Apr 2017.

[Ram14] Alex Ramirez. The mont-blanc prototype, 2014.

[RCG+13] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with commodity cpus: Are mobile socs ready for hpc? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 40. ACM, 2013.

[RML08] Miguel A Revilla, Shahriar Manzoor, and Rujia Liu. Competitive learning in informatics: The uva online judge experience. *Olympiads in Informatics*, 2:131–148, 2008.

[RRM$^+$16] Nikola Rajovic, Alejandro Rico, Filippo Mantovani, Daniel Ruiz, Josep Oriol Vilarrubi, Constantino Gomez, Luna Backes, Diego Nieto, Harald Servat, Xavier Martorell, et al. The mont-blanc prototype: an alternative approach for hpc systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 444–455. IEEE, 2016.

[RRP$^+$14] Nikola Rajovic, Alejandro Rico, Nikola Puzovic, Chris Adeniyi-Jones, and Alex Ramirez. Tibidabo: Making the case for an arm-based hpc system. *Future Generation Computer Systems*, 36:322–334, 2014.

[RS17] RecSys - ACM Recommender Systems. `https://recsys.acm.org/`, Jun 2017.

[SE517] Samsung Exynos 5422. `http://www.samsung.com/global/business/semiconductor/product/application/detail?productId=7978&iaId=2341`, Apr 2017.

[SG517] Samsung Galaxy S5. `http://www.samsung.com/uk/smartphones/galaxy-s5-g900f/SM-G900FZKABTU/`, Apr 2017.

[TC17] TopCoder CrowdSourcing. `https://www.topcoder.com/`, Jun 2017.

[TDT17] TDT4102 - Procedural and Object-Oriented Programming. `https://www.ntnu.no/studier/emner/TDT4102`, Jun 2017.

[TF15] Simen Støa Torbjørn Follan. Climbing mont blanc - a prototype system for online energy efficiency based programming competitionson arm platforms. 2015.

Appendix A

Installation and Setup

Magnussen provides a comprehensive installation guide of the system in [Mag16]. For new developers it's highly recommended to follow those instructions when setting up the system. This chapter expands and updates the sections in need to match the new version of the system.

A.1 Backend Setup

Some additional technologies are needed on the backend to support the new profiling feature. These are the linux perf tools, and the Flame Graph tool.

A.1.1 Installing perf on OdroidXU3

The perf profiling tool might not be supported out of the box on the OdroidXU3 board. To find out you can perform a simple check using the command `perf stat ls`. If the most of the metrics are listed as `<not supported>`, following the guide at [Pat17] and [Mid17] (which is simplified beneath) should fix the issue.

What is needed is a recompilation of the linux kernel with the modification of two files. A branch with these modifications (and some other small changes) is available at [FPL17], and using this branch simplifies the process. Following is the method of recompiling the linux kernel.

The other changes included in the linux branch at [FPL17], include a bug fix of the perf tools, and a change so that perf stat ends with the process if attached to it (explained in section 3.1.1). New developers should be aware of these changes if they are using a different linux kernel. Indeed, if hardkernel releases a newer linux kernel for the Odroid XU3 (our version was 3.10), the instructions in this section are no longer valid.

Followig are the steps for recompiling the kernal.

```
$ git clone --depth 1 --single-branch -b odroidxu3-3.10.y
   https://github.com/fredrikpe/linux
$ cd linux
$ make odroidxu3_defconfig
$ make -j 8 zImage dtbs modules
$ sudo cp arch/arm/boot/zImage arch/arm/boot/dts/*.dtb /media/boot
$ sudo make modules_install
$ sudo make firmware_install
$ sudo make headers_install INSTALL_HDR_PATH=/usr
$ kver=`make kernelrelease`
$ sudo cp .config /boot/config-${kver}
$ cd /boot
$ sudo update-initramfs -c -k ${kver}
$ sudo mkimage -A arm -O linux -T ramdisk -a 0x0 -e 0x0
   -n initrd.img-${kver} -d initrd.img-${kver} uInitrd-${kver}
$ sudo cp uInitrd-${kver} /media/boot/uInitrd
```

A restart of the OS is now required. Afterwards run the following to install perf.

```
$ cd linux/tools/perf
$ make -j `getconf _NPROCESSORS_ONLN` perf
$ cp perf /usr/bin/
```

If done correctly the command `perf stat ls` should now display the appropriate metrics (i.e., not all of them but most).

A.1.2 Flame graph

Installing Flame Graph is a simple git clone.

```
$ git clone https://github.com/brendangregg/FlameGraph
```

Appendix B

Quickstack and perf patches

B.1 Quickstack Patch for ARM processors

Quickstack uses the stack pointer and the instruction pointer for figuring out stack straces. It includes the header file sys/user.h from the GNU C Library [GCL17], which is only used by GDB, and includes a struct called `user_regs_struct` referencing the registers. This is implemented differently based on the architecture of the CPU. Quickstack supported x86 and i386 variants, but not ARM 32 or 64. To add support for 32 bit ARM, for our Cortex-A15 and Cortex-A7 CPUs, some changes to the accessing of these registers were necessary. These are highlighted in listing reflst:qspatch

```
@@ -13,6 +13,9 @@
 #elif defined(__x86_64__)
 #define STACK_IP rip
 #define STACK_SP rsp
+#elif defined(__arm__)
+#define STACK_IP 15
+#define STACK_IP 13
@@ -199,9 +202,18 @@ static bool match_debug_file(const string& name, const char* file) {
  }

 static int get_user_regs(int pid, user_regs_struct& regs) {
+#if defined(__arm__)
+  struct iovec iov;
+  iov.iov_base = (void*) &regs;
+  iov.iov_len = sizeof(regs);
+#endif
    int count = 100;
    while (1) {
+#if defined(__arm__)
+    int e = ptrace(PTRACE_GETREGSET, pid, NT_PRSTATUS, &iov);
+#else
      int e = ptrace(PTRACE_GETREGS, pid, 0, &regs);
+#endif
      if (e != 0) {
        if (errno == ESRCH && count-- > 0) {
          sched_yield();
@@ -405,7 +417,7 @@ static bool check_shlib(const std::string& fn) {
      return false;
    }
    ulong vaddr = 0;
-#if defined(__i386__)
+#if defined(__i386__) || defined(__arm__)
    Elf32_Ehdr* const ehdr = elf32_getehdr(elf);
    Elf32_Phdr* const phdr = elf32_getphdr(elf);
@@ -414,7 +426,7 @@ static bool check_shlib(const std::string& fn) {
    const int num_phdr = ehdr->e_phnum;
    for (int i = 0; i < num_phdr; ++i) {
-#if defined(__i386__)
+#if defined(__i386__) || defined(__arm__)
      Elf32_Phdr* const p = phdr + i;
  #else
      Elf64_Phdr* const p = phdr + i;
@@ -663,8 +675,13 @@ static int get_stack_trace(int pid,
    uint n_scanned_from_last_frame = 0;
    bool sp_jumped = false;

+#if defined(__arm__)
+  ulong sp = regs.uregs[STACK_SP];
+  ulong top_addr = regs.uregs[STACK_IP];
+#else
    ulong sp = regs.STACK_SP;
    ulong top_addr = regs.STACK_IP;
+#endif
@@ -1393,7 +1410,11 @@ int main(int argc, char** argv) {
    get_tids(target_pid, threads);
    _attach_started = (int*)mmap(
+#if defined(__arm__)
+       0, getpagesize(), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
+#else
        0, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
+#endif

    pid_t quickstack_core_pid = fork();
    if (quickstack_core_pid < 0) {
```

Listing 12: Changes to quickstack.cc to support 32-bit ARM

B.2 Linux Perf Tools Patch

A patch was applied to change the behaviour of the `perf stat` command when it is attached to running processes. Normally it is required to hit Ctrl+C to stop the command even though the process has terminated. After this patch the `perf stat` command terminates alongside the process it has been attached to.

```
 if (WIFSIGNALED(status))
         psignal(WTERMSIG(status), argv[0]);
 } else {
-       while (!done) {
-           nanosleep(&ts, NULL);
-               if (interval)
-                       print_interval();
-               }
+       char piddir[40];
+       int check_proc = target.pid && access("/proc", X_OK) == 0 \
+           && !strchr(target.pid, ',');
+       if (check_proc)
+           snprintf(piddir, sizeof piddir, "/proc/%d", atoi(target.pid));
+       while(!done) {
+           nanosleep(&ts, NULL);
+           if (interval)
+               print_interval();
+           if (check_proc && access(piddir, X_OK) < 0 && errno == ENOENT)
+               break;
+       }
 }

 t1 = rdclock();
```

Listing 13: Changes to builtin-stat.c to change perf stat attachment behaviour

Appendix C

Backlog

- **Bugs and known issues:**
  - Flame graph is saved in browser and does not reload when a new is generated.
  - Some characters give problems in DB dump. Should be encoded to Unicode first.
- **Security issues:**
  - Security when profiling. User program should be run under the worker user.