

Ole Kristian Eidem Pedersen

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Ole Kristian Eidem Pedersen

Security and Measurement Stability in the Climbing Mont Blanc Online Judge

August 2019



Norwegian University of
Science and Technology

Security and Measurement Stability in the Climbing Mont Blanc Online Judge

Ole Kristian Eidem Pedersen

Computer Science

Submission date: August 2019

Supervisor: Lasse Natvig

Co-supervisor: Rajiv Nishtala

Norwegian University of Science and Technology
Department of Computer Science

Programmer Feedback, Security and Measurement Stability in the Climbing Mont Blanc Online Judge

Climbing Mont Blanc (CMB)¹ is a system for evaluation of programs executed on modern heterogeneous multicores such as those used in mobile phones. CMB evaluates both performance and energy efficiency, and provides the possibility of performance ranking lists and online competitions.

The student should:

1. Improve feedback given to CMB users about typical compilation and runtime errors.
2. Implement mitigations for security challenges related to evaluating user code and giving feedback, and (optionally) any other part of the system.
3. Improve the stability of measurements, especially focusing on energy measurements, but also time measurements.
4. Continuously refactor and improve the quality of the code base during the thesis work, including implementing more tests and improving logging, in order to increase system maintainability and reliability.

If time permits, the student should:

- A. Conduct a user experiment to evaluate how improved feedback affects usability.
- B. Propose improvements to the testing of submitted programs in order to give better feedback to the user.
- C. Propose solutions for implementing support for multiple languages and compilers.
- D. Propose solutions for handling multiple XU3-boards and different execution platforms (back ends).
- E. Suggest general improvements and bug-fixes to improve any other aspect of the CMB project.
- F. Develop a command line client for automated uploads to the CMB system from the user's command line (instead of using the website).
- G. Implement some of the proposed solutions after approval by CMB project's coordinator.

¹<https://www.ntnu.edu/idi/lab/cal/cmb>

Abstract

Energy consumption is increasing alongside the need for performance for supercomputers and small, mobile computing devices alike. In order to increase energy-efficiency and performance, energy-constrained platforms are moving from homogeneous to heterogeneous multicores.

The need for developers with knowledge of how to build energy-efficient, high-performance applications is increasing. Many online training platforms for programmers exists, but before the Climbing Mont Blanc project, no publicly available training platform with the focus on heterogeneous multicores or energy-efficient programming existed.

This thesis improves and adds more features to the Climbing Mont Blanc system, mainly related to measurement stability and program evaluation security, but progress has also been made on the maintainability of the system and feedback to users. For single-threaded, computation-bound programs the coefficient of variation for measurements has been improved from 0.13% to 0.012%.

Sammendrag

Energiforbruk og ytelsesbehov øker for både superdatamaskiner og mindre, mobile enheter. For å øke energieffektivitet og ytelse beveger energibegrensede plattformer seg fra homogene til heterogene multikjerner.

Behovet for utviklere som har kunnskapen til å bygge energieffektive høyttelsesapplikasjoner øker. Mange nettbaserte treningsplattformer for programmerere eksisterer, men før Climbing Mont Blanc-prosjektet eksisterte ingen allment tilgjengelige treningsplattformer med fokus på heterogene multikjerner eller energieffektiv programmering.

Denne oppgaven forbedrer og utvider funksjonaliteten til Climbing Mont Blanc-systemet, primært målingsstabilitet og programevalueringssikkerhet, men vedlikeholdbarhet og tilbakemeldinger til brukere har også blitt forbedret. For beregningskrevende enkelttrådprogrammer har variasjonskoeffisienten blitt forbedret med 0.13% til 0.012%.

Preface

This master's thesis is submitted to Norwegian University of Science and Technology (NTNU) in fulfilment of the final requirement for the degree of Master of Science (MSc) and is a continuation of my² specialization project (TDT4501) conducted during the autumn of 2018. The work has been conducted at the Department of Computer Science (IDI), NTNU, Trondheim, Norway throughout the spring of 2019. Alongside working on the thesis, I was employed in a 50% position as a research assistant in the course TDT4102 — Procedural and Object-Oriented Programming, involving primarily administrative duties, such as answering emails from the 800 students in the course, managing a staff of 55 teaching assistants, and giving exercise-focused and repetition lectures.

²In what follows, “I”, “we”, “my” or “our” will be used to refer to my exclusive contributions.

Acknowledgements

First and foremost, I would like to thank my supervisors Prof. Lasse Natvig and Rajiv Nishtala, Ph.D., for their valuable feedback throughout the thesis work. They have provided guidance, answers and encouragement when needed, and for that I am grateful.

I am also thankful to my family for their continuous support during my five years of master's studies culminating in this thesis.

Finally, I would like to thank Helene Westerby for all her love and support, and for her patientence and understanding when finishing the thesis took more time than expected.

Table of Contents

Problem statement	i
Abstract	iii
Sammendrag	v
Preface	vii
Acknowledgements	ix
Table of Contents	xi
List of Tables	xv
List of Figures	xvii
List of Listings	xix
Acronyms	xxi
1 Introduction	1
1.1 Motivation	1
1.2 Climbing Mont Blanc project history	2
1.3 Thesis Scope and Goals	4
1.4 Contributions	5
1.5 Outline	6
2 Background	9
2.1 The Climbing Mont Blanc System	9
2.1.1 Overview	9
2.1.2 Front End	12
2.1.3 Server	12

2.1.4	Back End	15
2.1.5	Energy and Energy Efficiency Measurements	18
2.1.6	Program Evaluation Security	20
2.2	Related Work	23
2.2.1	Selected Educational Online Judges	24
2.2.2	Selected Programming Contest Platforms	25
2.2.3	Selected Open Source Online Judges	28
2.2.4	Selected Recruitment Platforms	29
2.3	Summary	30
3	CMB Challenge 2019	31
3.1	Competition Format	32
3.2	Problem Statements	34
3.2.1	To Quote Hamlet...	34
3.2.2	Pirates and Probabilities	34
3.2.3	The Huckybucky Forest	34
3.2.4	In Ventus	35
3.2.5	Flower Power	35
3.2.6	There and Back Again	36
3.3	Solution Improvements	36
3.3.1	Loop Unrolling	36
3.3.2	Task and Data Parallelism	37
3.3.3	“Skip Seed”	37
3.3.4	Miscellaneous Methods	40
3.4	Questionnaire	41
3.5	Summary	41
4	Implementation	43
4.1	Operating System Upgrade	43
4.2	Implementing Tests on the Back End	45
4.3	Automating Installation on the Back End	47
4.4	Program Evaluation Improvements	48
4.4.1	Measurement Stability	48
4.4.2	Security	52
4.4.3	Run-Time Error Feedback	55
4.4.4	Refactorings and Bug Fixes on the Back End	56
4.5	Measurement Stability Experiments	56
4.5.1	Measurement Stability Statistic	57
4.5.2	Measurement Stability Experimental Setup	57
4.5.3	Measurement Stability Test Programs	58
4.6	Refactorings and Bug Fixes on the Server	59
5	Results and Discussion	61
5.1	Prior Experiments Using the CMB System	61
5.2	Experimental Results	63
5.2.1	Baseline Comparison	63

5.2.2	Discussion and Evaluation of Hypotheses	65
5.3	Errors and Threats to Validity	71
6	Evaluation and Conclusion	75
6.1	Evaluation	75
6.2	Conclusion	77
7	Future Work	79
7.1	Project Management and Development Process	79
7.1.1	Change Repository Hosting	79
7.1.2	Continuous Integration	79
7.2	Front end	80
7.2.1	Upgrade or Rewrite Web Application	80
7.2.2	Improve HowTo	80
7.2.3	Contest Features	80
7.2.4	Command Line Client	80
7.3	Server	81
7.3.1	Improving Compilation Error Messages	81
7.3.2	Upgrade to Python 3	81
7.3.3	Logging	82
7.3.4	Securing Compilation	82
7.4	Back End	82
7.4.1	Improving Measurement Stability	82
7.4.2	Improving Security	84
7.4.3	Other	85
	Bibliography	87
	Appendices	95
A	Installation Instructions	95
A.1	Back End	95
A.2	Server and Front End	96
B	CMB Challenge 2019	97
B.1	General information, rules and technical information	97
B.2	Problem descriptions	100
B.3	Questionnaire	117
C	Test Programs	121
D	Additional Data	131
E	Digital Appendix	133

List of Tables

1.1	Thesis contributions	6
4.1	Configuration options for <code>runscript_v2.sh</code>	56
5.1	C_v (lower is better) from experiments performed by Støa and Follan [SF15].	61
5.2	C_v (lower is better) from experiments performed by Ingebrigtsen [Ing17].	62
5.3	Sample mean, sample standard deviation and C_v (lower is better) from measurements performed by the Climbing Mont Blanc (CMB) project coordinator in April, 2018.	63
5.4	A summary of the conclusion to the hypotheses.	71
D.1	Sample mean time, sample standard deviation and C_v (lower is better) for Hello World, Sort and Sort w/ <code>RandInt</code>	131
D.2	Sample mean time, sample standard deviation and C_v (lower is better) for the Mandelbrot programs.	131
D.3	Sample mean energy consumption, sample standard deviation and C_v (lower is better) for Hello World, Sort and Sort w/ <code>RandInt</code>	132
D.4	Sample mean energy consumption, sample standard deviation and C_v (lower is better) for the Mandelbrot programs.	132

List of Figures

1.1	A flame graph generated by the CMB system.	3
2.1	Overview of subsystems.	10
2.2	The group interface.	11
2.3	Google Analytics dashboard.	11
2.4	Screenshot from Google Analytics showing the number of users and some aggregate metrics for the CMB website over a period of 180 days.	13
2.5	The Unicorn pre-fork model with asynchronous workers.	14
2.6	The current server setup.	15
2.7	Segmentation fault error message.	16
2.8	Odroid-XU3 block diagram.	17
2.9	Overview of temperature sensors and power monitors.	19
2.10	A visualization of how time and energy affects the EDP.	19
2.11	Codecademy tutorial screenshot.	25
2.12	Codewars solution sharing.	26
2.13	Codewars integrated code editor.	26
2.14	A screenshot from the uHunt tool [Uhu] for UVa.	27
2.15	The HackerRank autocompletion feature.	30
3.1	The LCG state machine for random number generation.	40
5.1	<i>Baseline</i> experiment results, showing the C_v (lower is better) for the test programs.	64
5.3	The Mandelbrot program in the <i>Perf + C++ + chroot + taskset + nice</i> experiment.	65
5.2	The C_v results for all versions of the Mandelbrot program.	66
5.4	Time-energy plots for the Mandelbrot program showing that the grouping of measurements disappears when implementing measurements in C++.	68

5.5	Comparison of the I/O-bound program <i>Sort</i> to the computation-bound program <i>Sort w/RandInt</i> in the <i>Performance governor + C++ + chroot</i> experiment.	71
5.6	Plot of the time and energy measurements for the Mandelbrot program in the <i>Performance governor</i> and <i>Performance governor + C++</i> experiments that shows that measurements might be affected by tasks performed prior to the beginning measurements.	72

List of Listings

2.1	<code>sched_setaffinity(2)</code> example.	18
2.2	Exploit for unsecured profiling.	22
3.1	The <code>RandInt</code> class handed out to students for number generation.	33
3.2	An excerpt from a solution to the “To Quote Hamlet” problem.	35
3.3	A loop unroll optimization written by a student.	37
3.4	The “Skip Seed” method.	39
3.5	A student’s solution guessing the answer.	40
3.6	An alternative <code>RandInt</code> implementation.	42
4.1	C++ inspect files exploit.	53
C.1	The <i>Hello World</i> test program.	121
C.2	The <i>Sort</i> test program.	122
C.3	The <i>Sort w/RandInt</i> test program.	123
C.4	The <i>Mandelbrot</i> test program.	125
C.5	The <i>Mandelbrot (OpenMP)</i> test program.	127
C.6	The <i>Mandelbrot (OpenCL)</i> test program.	129
C.7	The kernel used with the <i>Mandelbrot (OpenCL)</i> test program.	130

Acronyms

- AGPL** GNU Affero General Public License. 29
- API** application programming interface. 10, 12–15, 43, 46, 56, 60, 80, 81
- CI** continuous integration. 79, 80
- CLI** command line interface. 80, 81
- CMB** Climbing Mont Blanc. xv, 1–7, 9, 10, 12, 13, 18, 20, 21, 23–25, 28–32, 41, 43, 44, 47, 49–52, 55, 61–64, 67–71, 76, 77, 79–81, 86
- DoS** denial-of-service. 20, 82
- DRY** Don't Repeat Yourself. 55
- EDP** energy delay product. 18, 19, 32, 36
- EOL** end-of-life. 43, 44, 56, 81
- GPL** GNU General Public License. 28, 29
- GPU** graphical processing unit. 9, 15, 20, 44, 59, 67
- GTS** Global Task Scheduling. 16
- HMP** heterogeneous multi-processing. 16
- HTTP** Hypertext Transfer Protocol. 12
- HTTPS** HTTP Secure. 13, 14
- ICPC** ACM International Collegiate Programming Contest. 27, 28
- IDI** Department of Computer Science. vii, 4, 10, 14

IoT Internet of Things. 1

JSON JavaScript Object Notation. 12, 45, 55, 58, 81

LCG linear congruential generator. 38

NTNU Norwegian University of Science and Technology. vii, 4, 31, 49

OJ online judge. 2, 6, 9, 20, 21, 23–25, 27–30

ORM object-relational mapper. 14

OS operating system. 16, 28, 43, 44, 47, 55, 58, 59, 63, 77, 81

REST Representational State Transfer. 12, 13

RSD relative standard deviation. 19

SDK software development kit. 44

SoC system-on-chip. 2, 9, 15, 16

SPA single-page application. 12

SSH Secure Shell. 14

TSP Traveling Salesperson Problem. 36

UFW Uncomplicated Firewall. 49

UI user interface. 12, 41

Chapter 1

Introduction

This chapter presents the motivation and the history of the Climbing Mont Blanc (CMB) project to provide some context for the thesis. Further, the problem statement interpretation is presented, followed by the contributions and the outline of this thesis.

1.1 Motivation

The HiPEAC Vision 2017 states that “energy efficiency of computing systems remains a major challenge for the coming years” [DDG+17, p. 7]. Energy consumption is increasing alongside the need for performance for supercomputers and small computing devices (e.g., mobile phones, Internet of Things (IoT) devices) alike. If energy efficiency of computing is not improved, the energy consumption cost for post-exascale computers may be higher than most countries are willing to spend [DDC+19], and the small (battery-powered) devices introduced by the IoT revolution will very likely be unsuccessful due to their limited autonomy [DDG+17]. This shows the need for increased attention on energy efficient computing in the future.

More efficient hardware with specialized devices (i.e., accelerators) are one way to increase energy efficiency, but at the cost of more complex programming. Energy-constrained computing platforms are now leaving homogeneous multicores for heterogeneous multicores in order to increase performance. Moreover, considering energy consumption during software development is important to ensure energy efficiency [DDC+19].

The need for developers with knowledge of how to build energy-efficient, high-performance applications is increasing. Many online training platforms for programmers exist, allowing users to practice and improve their programming language knowledge, solve algorithmic problems and compare themselves to other users. The online judges (OJs) presented in Section 2.2 are examples of such platforms. However, no other publicly available training platform with the possibility of programming heterogeneous multicores or measuring energy consumption and energy-efficiency is known to the CMB project.

Noticing the lack of training platforms for beginner programmers with focus on energy efficiency and heterogeneous programming, Prof. Lasse Natvig created the CMB project. The aim of the CMB project is to stimulate and help programmers overcome the challenges related to programming heterogeneous computers and exploiting their potential energy-efficiency effectively [NFS+15].

The CMB system is available to programmers and researchers alike at <https://climb.idi.ntnu.no>. Although the CMB system has been publicly available since the 2015, challenges with the CMB system still remain. The system is not able to handle many users simultaneously, has issues with measurement stability, is vulnerable to a wide range of security issues and is hard to maintain. This thesis tries to remedy or propose solutions to some of these issues.

1.2 Climbing Mont Blanc project history

The initial inspiration for the CMB project came from the Mont Blanc project [Mon]. The Mont Blanc project used a large number of system-on-chips (SoCs) to build an energy-efficient high-performance computer with commercially available low-power embedded technology. The supercomputer prototype ran on heterogeneous ARM-based hardware such as Samsung’s Exynos mobile processors.

Observing the rise in popularity of OJs such as the UVa Online Judge [Uva] and Kattis [Kat] further motivated creating the CMB project as a training platform for programmers. In addition to using the same SoC as the Mont Blanc project, the creators of the CMB project envisioned competing programmers “climbing” the high-score lists (the Mont Blanc) when submitting programs, resulting in the name “Climbing Mont Blanc” [NFS+15].

During the autumn of 2014 and spring of 2015 the first version of the CMB system was built by Støa and Follan as part of their specialization project and master’s thesis [SF15]. The system used the Odroid-XU3 board, containing the Samsung Exynos 5422 SoC and built-in energy sensors [Odrb]. This SoC consists of four “small” ARM A7-cores, and four “big” ARM A15-cores, following the ARM big.LITTLE heterogeneous architecture [Big]. Additionally, the SoC has a six-core

ARM Mali-T628 GPU. The Odroid-XU3 board is still the only board used by the CMB system.

The system has been tested with real users on multiple occasions. During the spring of 2015 Støa and Follan [SF15] tested the system on users during the training session of IDI Open 2015 [Idi; SF15]. It was also used as part of mandatory exercises in the course TDT4200 Parallel Computing [Tdtc] in the autumn of 2015 [Mag16].

As part of his specialization project and master’s thesis work during the autumn of 2015 and spring of 2016, Magnussen [Mag16] worked on improving the system usability of the CMB system using the feedback received from previous experiences. A user experiment was also conducted, showing that users were more satisfied with the usability of the improved system [Mag16].

Alongside the work of Magnussen [Mag16], two other projects related to the CMB system were carried out. The first project was the master’s thesis by Lier and Mathisen, “Experiments towards digital exam with auto-grading in C++ programming courses”, in which they tested the use of the CMB system as an exam auto-grading system [LM16]. User experiments were also conducted as part of this project, providing more useful feedback about the CMB system. Some of the more important issues discovered was resolved by Magnussen [Mag16].

The second project was the master’s thesis by Chavez [Cha16]. This project focused on scaling the CMB project to use multiple Odroid-XU3 boards. This work was done separately from the work of Magnussen, creating diverging paths for the project [Cha16]. The CMB project’s coordinator eventually decided to discontinue the work done by Chavez.

Later, in the spring of 2017 Ingebrigtsen [Ing17] improved the stability of the system during his master’s thesis, in addition to implementing profiling capabilities. This allowed users to view performance counters and a flame graph (shown in Fig. 1.1), potentially making it easier for users to discover bottlenecks in their code [Ing17].

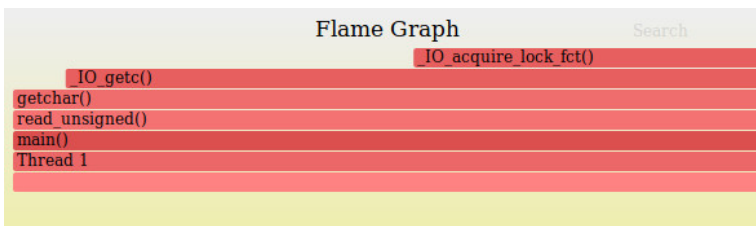


Figure 1.1: A flame graph generated by the CMB system.

This thesis will hereby refer to the version developed by Støa and Follan [SF15] as *system version one*, the version developed by Magnussen [Mag16] as *system version two*, the version developed by Ingebrigtsen [Ing17] as *system version three* and the

version developed as part of this thesis as *system version four*. Note that the work done by Chavez [Cha16] is not included in the current version of the system, and thus isn't assigned a version.

The CMB system has been used for the “CMB Challenge”: competitions between students in the C++ programming course (TDT4102 [Tdda]) at IDI, NTNU and was first arranged in 2017. Most of the attending students are new to programming, having had only one introductory programming course before the C++ course. However, these competitions have given the CMB project useful insights into problem types, the span of optimizations possible and general use of the CMB system. The competition in 2018 with tasks and findings were published by Natvig et al. [NSLH19]. The findings from the 2019 competition are presented in Chapter 3.

1.3 Thesis Scope and Goals

The problem statement objectives are formulated as high-level goals. However, in order to make the work more focused, several more concrete subgoals have been defined. These subgoals serve as a checklist to define what has to be done to complete the goal.

1. Improve the feedback given to CMB users about typical compilation and runtime errors.
 - (a) Signals and output from the program/compiler must be parsed and presented to the user in a user-friendly way.
 - (b) The messages presented to the users must not reveal sensitive information that may be exploited by malicious users.
 - (c) Implement tests to ensure that no regressions occur during future upgrades of system components (e.g., when upgrading compilers).
2. Implement mitigations for security challenges related to evaluating user code and giving feedback, and (optionally) any other part of the system.
 - (a) Upgrade both server and Odroid-XU3 boards to use Ubuntu 18.04 LTS.
 - (b) Identify and implement mitigations for potential threats.
 - (c) Implement tests to ensure no future changes to the system leave the system vulnerable to previously fixed errors.
3. Improve the stability of measurements, especially focusing on energy measurements, but also time measurements.
 - (a) Implement a small framework to quantify and compare the accuracy of time and energy measurements, using suitable metrics.

- (b) If possible, compare measurement accuracy results from the upgraded system to results from the system before the upgrade.
 - (c) Identify and implement suitable approaches to improve measurement stability, measuring the impact of each approach.
4. Continuously refactor and improve the quality of the code base during the thesis work, including implementing more tests and improved logging, in order to increase system maintainability and reliability.
- (a) Select tools for implementing tests and implement a test suite for the Odroid-XU3.
 - (b) Improve logging messages to more easily track down errors.
 - (c) Refactor and reorganize the code base when necessary.

When writing the problem statement, the complexity of implementing the objectives was not known. However, during the thesis work the complexity has been discovered and it was decided that implementing all the objectives in a good, maintainable way within the time frame of this thesis would not be feasible. As a result, improving feedback (objective 1) has been deprioritized and changes that could not be implemented within the time frame are suggested in Chapter 7.

1.4 Contributions

This thesis primarily contributes towards improving the evaluation of programs in the CMB system in terms of measurement stability and security. The stability of measurements is important for competitions and research purposes, and security is important to protect the system from malicious users. However, security features might interfere with measurements, requiring careful implementation and testing.

During the specialization project it was discovered that the code base is characterized by multiple developers working with short-term goals, and that this affected the quality of the code base. Therefore, an explicit objective of improving the quality of the code base was made part of the problem statement to promote long-term thinking during development.

Some contributions have been made to the CMB project that were not covered by the thesis description. The CMB Challenge 2019 described in Chapter 3 was arranged for students in the TDT4102 course. Arranging this competition and analyzing submissions from the students was the most time-consuming of these contributions.

Table 1.1 has a summary of where to find descriptions of the work performed to fulfill the thesis objectives.

Table 1.1: Thesis contributions

Objective	Sections/Chapters
1	Sections 4.2 and 4.4.3 and Chapter 7
2	Sections 4.1, 4.2 and 4.4.2
3	Sections 4.4.1, 4.5 and 5.1
4	Sections 4.2, 4.3, 4.4.4 and 4.6

1.5 Outline

This thesis is structured as follows:

Chapter 2: The chapter presents the Climbing Mont Blanc system at the start of this thesis, focusing primarily at the server and the back end. The method of performing energy and energy-efficiency measurements are described. The security measures when evaluating programs are also described. Finally, other OJs and other related work is presented.

Chapter 3: The chapter describes the CMB Challenge competition that was arranged as part of this thesis. The competition format, and a brief description of problem statements and solutions, are presented. Further, techniques to improve solutions that are found by the students participating in the competition, as well as the insights gained from the questionnaire distributed after the competition are presented.

Chapter 4: The chapter presents the development work performed to fulfill the objectives of this thesis. Implementation details and different approaches considered are also presented, along with hypotheses for approaches that might affect the measurement stability.

Chapter 5: The chapter presents and discusses the experimental results. Prior measurements are presented to act as a baseline, and the experimental results from this thesis are used to discuss and draw conclusions regarding the hypotheses. Finally, errors and threats to validity are discussed.

Chapter 6: The chapter presents an evaluation of the thesis work performed and the achievement of the problem statement objectives, before concluding this thesis.

Chapter 7: The final chapter presents suggestions of future work based on knowledge of the CMB project accumulated during the thesis work.

Chapter 2

Background

2.1 The Climbing Mont Blanc System

This section describes the state of the CMB system before the start of the thesis work (i.e., system version three).

2.1.1 Overview

The CMB system is designed as an OJ, allowing users to submit solutions to a set of problems. The users receive feedback in the form of run time, energy consumption and energy efficiency, and their submissions are ranked by these properties on a scoreboard. The major difference between the CMB system and other OJs is that the CMB system reports energy consumption and energy efficiency.

The code uploaded to the system is evaluated on a 14-core heterogeneous SoC—four energy-efficient cores, four high-performance cores, and six graphical processing unit (GPU) cores—which gives users multiple optimization possibilities. The hardware is described in more detail in Section 2.1.4.

Most OJs only support single-threaded programming, but a few also allow multi-threaded programming using features integrated into the languages ¹, which is

¹The CMB project is only aware of one other OJ that supports third-party libraries for parallel programming (e.g., OpenMP), named JudgeGirl (located at <https://judgegirl.csie.org/>), but this OJ is only available to students from Nanyang Technology University in Singapore.

usually achieved by compiling with pthreads enabled (for C/C++). In addition to compiling with pthreads, the CMB system supports the OpenMP and OpenCL libraries. UNIX system calls are also supported, for scheduling programs on specific cores, for example.

The CMB system contains three separate subsystems in addition to a database. A high-level overview of these subsystems is shown in Fig. 2.1. A user can primarily communicate with the system through the front end; a web application available at <https://climb.idi.ntnu.no>. The server serves data through an application programming interface (API) endpoint and provides the administrator interface. The back end compiles and runs programs upon request from the server, returning the time and energy measurements, or errors in case the program fails.

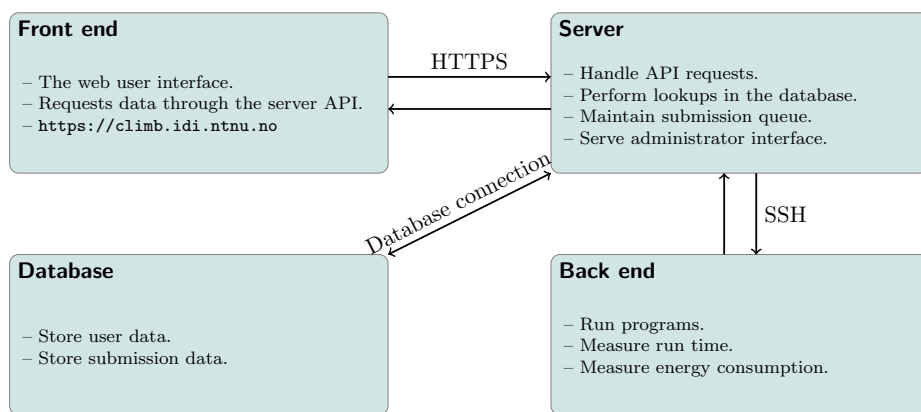


Figure 2.1: The CMB system contains three subsystems with different responsibilities and a database communicating using different protocols.

In order to facilitate competitions the CMB system has a group functionality. A group consists of a specified set of problems, decided by a group leader. Members of the group get access to a private scoreboard for each of the problems, which only shows other members of the group. Group leaders can download data for all submissions to the given problems made by the members of a group. This data can be used to score submissions and find winners of the competition. A view of the group interface is shown in Fig. 2.2.

The administrator interface gives the CMB system administrators a view of registered users, submissions and groups. Furthermore, it has functionality for uploading new problems and publishing news bulletins to the users. A screenshot of the administrator interface is shown in Fig. 2.3.

The front end and server is currently run on the same virtual Ubuntu instance on IDI's servers. The MySQL database is also provided and managed by IDI's

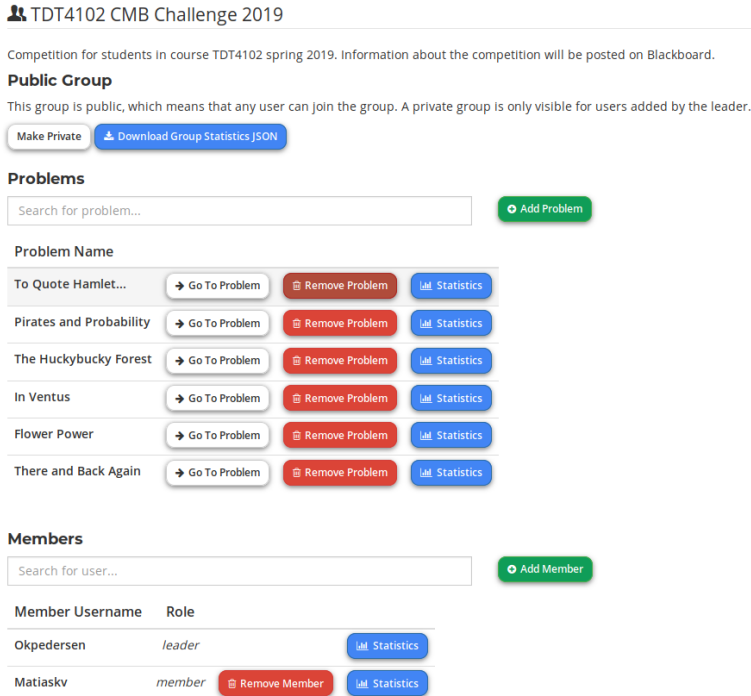


Figure 2.2: A screenshot of a group leader’s view of the group interface.

Admin

Home User Problem Submission Group Group Problem Member Run Profiling Bulletin Uploads Admin User DB Dump

okpedersen

List (4784) With selected

	Name	Time	Energy	Edp	Goodness Value	State	Detailed State	Msg	Submitted	Visible	Problem	User
<input type="checkbox"/>	std_threads_big_std_sort_v1.zip	13.0	41.9578	545.451		finished			2018-12-06 09:27:49	⊙	TEB-sort	Kalle
<input type="checkbox"/>	std_threads_big_std_sort_v1.zip	12.9	40.1166	517.505		finished			2018-12-06 09:27:47	⊙	TEB-sort	Kalle
<input type="checkbox"/>	std_threads_big_std_sort_v1.zip	13.41	43.8859	588.242		finished			2018-12-06 09:27:46	⊙	TEB-sort	Kalle
<input type="checkbox"/>	std_threads_big_std_sort_v1.zip	12.4	46.9418	582.079		finished			2018-12-06 09:27:43	⊙	TEB-sort	Kalle
<input type="checkbox"/>	std_threads_big_std_sort_v1.zip	12.19	44.9481	547.918		finished			2018-12-05 21:32:40	⊙	TEB-sort	Kalle
<input type="checkbox"/>	std_threads_big_std_sort_v1.zip	14.9	36.7081	546.951		finished			2018-12-05 20:05:53	⊙	TEB-sort	Kalle
<input type="checkbox"/>	std_threads_big_std_sort_v1.zip	13.96	41.1862	574.959		finished			2018-12-05 20:05:51	⊙	TEB-sort	Kalle
<input type="checkbox"/>	std_threads_big_std_sort_v1.zip	13.18	43.5955	574.589		finished			2018-12-05 20:05:49	⊙	TEB-sort	Kalle

Figure 2.3: A screenshot showing the submissions view of the administrator interface.

technical personnel. The Odroid-XU3 back ends are administered by the CMB team.

2.1.2 Front End

The front end is built as a web application, and provides the user interface (UI) with which all CMB users interact when solving problems. It is written as a single-page application (SPA) using AngularJS 1.3 [Ang], JavaScript ECMAScript 5th ed. [Ter17], HTML5 [FEL+17] and CSS [AER18].

The front end uses the Node.js runtime [Nod] for building all the assets needed (e.g., scripts and style sheets), using server-side JavaScript. Since all data is served through the API, the assets are identical for every user and do not require server-side processing when the user makes a request.

Socket.io [Soc] provides real-time, two-way communication between the server and connected clients (users). The CMB system uses this library to update the scoreboard after a submission has been evaluated and to give users notifications about their position in the submission queue.

The user activity on the website is monitored using Google Analytics [Goo]. This provides CMB administrators with data about how many users visits the site on any given day and some insights into user behavior such as average session duration and bounce rate, as shown in Fig. 2.4.

2.1.3 Server

The server is responsible for performing database operations and scheduling programs on the back end. A Representational State Transfer (REST) API is provided to make these operations available to a client (e.g., a user accessing the site through the front end).

A REST API is a simple, uniform Hypertext Transfer Protocol (HTTP) interface, independent of the technologies used by the client and the server. Such interfaces are often called RESTful web services [FT02]. RESTful web services are stateless, in other words, the server does not track the state of the front end. This behavior necessitates clients to send all required information to determine the state when making a request. For example, the client must send a valid authentication token when submitting programs to prove that the user is authorized to use the system.

The server is implemented in Python 2.7 [Pytb]. The Python Flask framework [Flaa] provides functionality for the REST API. JavaScript Object Notation (JSON)



Figure 2.4: Screenshot from Google Analytics showing the number of users and some aggregate metrics for the CMB website over a period of 180 days.

[Bra17] is used as the intermediary format for communication between the server and clients.

Gunicorn [Guna] is used for production and development servers to handle simultaneous requests from multiple users. Gunicorn uses a pre-fork worker model [Gunb]: when starting up, Gunicorn creates multiple forks, called *workers*, each running its own separate instance of the Flask application. There is no data sharing between these workers. When the application receives a request, Gunicorn routes it to one of the available workers or waits (*stalls*) until a worker becomes idle. The workers are configured as asynchronous workers using the *gevent* library [Gev], meaning each worker runs multiple (Python-style) threads. Asynchronous workers can serve multiple non-blocking requests simultaneously. This is shown in Fig. 2.5.

NGINX [Ngi] is used as a *reverse proxy*. It acts as an intermediary between the server and client, primarily redirecting traffic to the correct underlying web service (i.e., the Flask REST API) and serving static content. Additionally, it handles HTTP Secure (HTTPS) connections, abstracting away this extra complexity from underlying web services. It can also perform speed-ups (e.g., caching), and traffic-flow control (e.g., load-balancing between servers) [Rev].

The reverse proxy is currently placed on the same virtual server as the front end application and the Flask web service in the current CMB system setup, shown in Fig. 2.6. It serves *static content*, but forwards requests for *dynamic content*

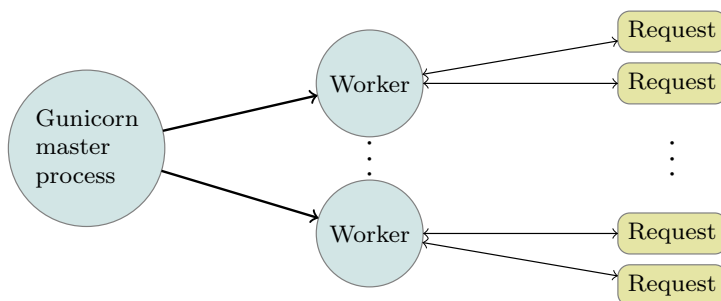


Figure 2.5: The Gunicorn pre-fork model with asynchronous workers for a Flask application. Each worker runs a separate instance of the Flask application, and can handle multiple non-blocking requests simultaneously.

to Gunicorn. Static content denotes files that are the same for all users, such as scripts, icons, images and CSS; dynamic content is content that may change between requests, or be different for different users, such as content stored in the database.

The production and development servers use MySQL databases [Mys] hosted by IDI’s technical department, while local developer environments (i.e., on the developer’s computer) primarily uses SQLite [Sqlb]. SQLAlchemy [Sqla] Python library makes this possible, functioning as an object-relational mapper (ORM). An ORM allows the developer to use a higher abstraction layer, using data structures and functions that are database independent. These data structures and functions are translated to the database’s SQL dialect by the ORM.

Submitted problems are added to the database and a queue managed by the application. This queue keeps track of the next submission to be run on the back end and is managed by sending HTTPS requests to specific (private) API endpoints to enqueue and dequeue submissions.

When a program is dequeued, the script will do a test compilation of the program on the server. If the compilation succeeds, the program is sent to the back end along with test data using the Secure Shell (SSH) utility [YL06]. The back end tests the programs and returns the answer to the measurement test set along with measurements made, or an appropriate error message if the program failed. The server checks that the answer to the measurement test set is correct and updates the database with the verdict (i.e., error message or measurement results). The solution to the big data set is never transported to the back end, in case a malicious user finds a way to read it. This provides an extra layer of security, but adds the extra complexity of checking correctness on the server.

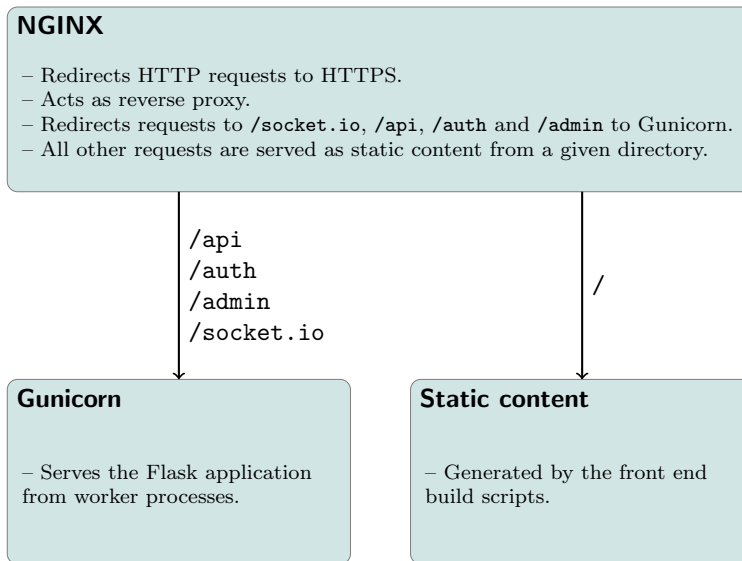


Figure 2.6: The current setup of the front end and server running on the same virtual server instance.

The result returned by the server through the API comprises the verdict, and an error message (if the program fails) or the measurement results. The error message explains the nature of the error, such as a compilation error or a runtime error. If it is a runtime error, the back end tries to determine the error based on the return code of the program, and shows the name of the signal if it is recognized. An example of a runtime error is shown in Fig. 2.7.

In order to support Socket.io at the front end, the Flask-SocketIO library [Flab] is used to handle these connections. By continuously emitting events over web socket, users are updated about the progress of the submission.

2.1.4 Back End

The back end is responsible for evaluating the code submitted by the users. Currently the back end runs on the Odroid-XU3 board, created by HardKernel [Odrb]. The board is powered by Samsung 5 Octa (Exynos-5422) SoC. This SoC employs the ARM big.LITTLE technology [CKC12], meaning that it contains four (“big”) high-performance, out-of-order Cortex-A15 2.0GHz cores, and four (“small”) power-efficient, in-order Cortex-A7 1.4 GHz-cores [Corb; Cora; CKC12; Odrb], allowing for improved processing capabilities while keeping power consumption low. The Exynos-5422 also employs a six-core ARM Mali-T628 GPU, supporting OpenGL

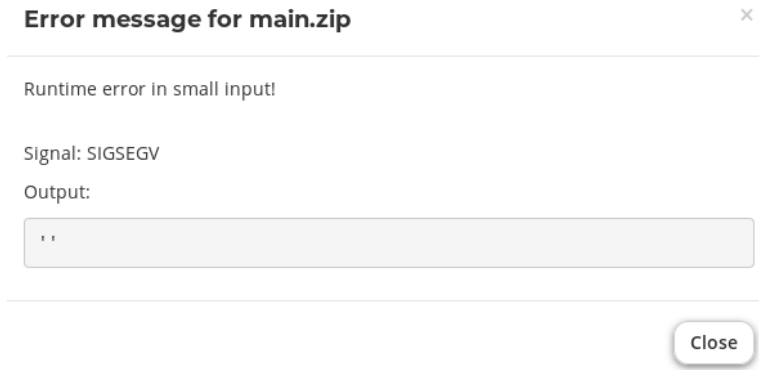


Figure 2.7: A screenshot of the error message given when provoking a segmentation fault during the small correctness test.

ES 3.0 and OpenCL 1.1. In total, this makes the Exynos SoC a *three-way* heterogeneous multicore with 14 cores. The full block diagram of the board is shown in Fig. 2.8.

The back end currently runs Ubuntu 14.04 (i.e., Ubuntu with the LXDE desktop environment). Running a full operating system (OS) provides some benefits, such as easier development and higher reusability of third-party packages. However, by using a full OS a lot of control is also given to the OS, which might create problems with measurement stability. This is discussed further in Section 2.1.5.

The OS is aware of both the high-performance and the power-efficient cores, and can dynamically schedule and migrate tasks independently to all cores simultaneously based on performance requirements. This model is known as heterogeneous multi-processing (HMP), or the Global Task Scheduling (GTS) model, and is used for SoCs using the big.LITTLE architecture [Big].

A program may manually override the scheduler by using Linux system calls. A minimal example using `sched_setaffinity(2)`² [Sch] is shown in Listing 2.1. It allows users to experiment with manual core assignment in order to achieve higher performance and energy efficiency.

The Odroid-XU3 board have sensors for measuring power and temperatures. The method for making measurements and calculating energy and energy efficiency is

²The number following the name refers to the section in the manual pages (known as *man pages*) describing the utility, and is used to distinguish utilities with the same name, for example, the command `chroot(1)` and the system call `chroot(2)` that have different man pages describing their functionality. The man pages can be viewed using the `man(1)` command on a Linux system, or online at <https://www.kernel.org/doc/man-pages/>.

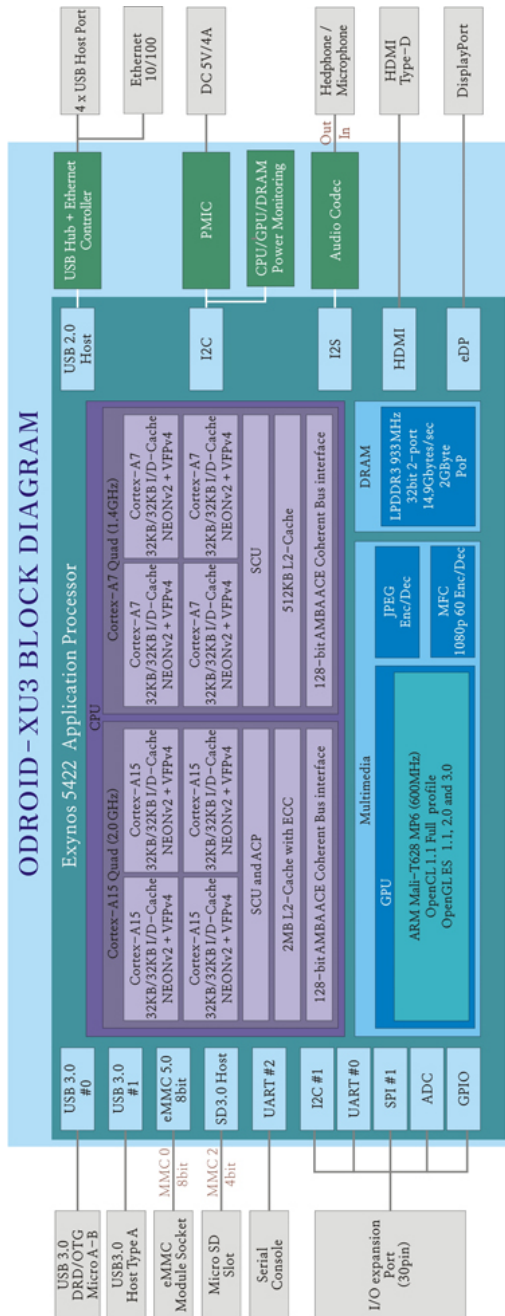


Figure 2.8: A block diagram for the Odroid-XU3 board, from the Odroid wiki [Odra].

```
#include <sched.h>
#include <stdio>

int main() {
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(3, &mask); // use core 3

    int result = sched_setaffinity(0, sizeof(mask), &mask);
    if (result == -1) // error, return early
        return -1;

    printf("Hello Italy!\nGreetings from the \
          CMB-team at NTNU in Trondheim, Norway.");
}
```

Listing 2.1: An example of how `sched_setaffinity(2)` [Sch] can be used to schedule the process to a given core in the “Hello Italy” problem.

outlined in Section 2.1.5. Additionally, manual fan control and turning off single cores is possible.

2.1.5 Energy and Energy Efficiency Measurements

A key piece of the CMB system is the energy consumption estimates. The Odroid-XU3 has Texas Instruments INA231 power monitors for measuring voltage, current and power used [Tii]. These properties are measured independently for the DRAM, GPU, the big cores and the small cores, as illustrated in Fig. 2.9.

An executable based on HardKernels EnergyMonitor program [Ene] is used to sample the power usage during program execution by running it as a background process. The sampling frequency is 100 MHz. Numerical integration of the samples provides us with an estimate of the energy consumption. The numerical integration is done using Simpson’s Rule from the Python SciPy package [Scib].

The chosen metric for energy efficiency is energy delay product (EDP), calculated eu the formula shown in Eq. (2.1). This metric provides a balance between the time

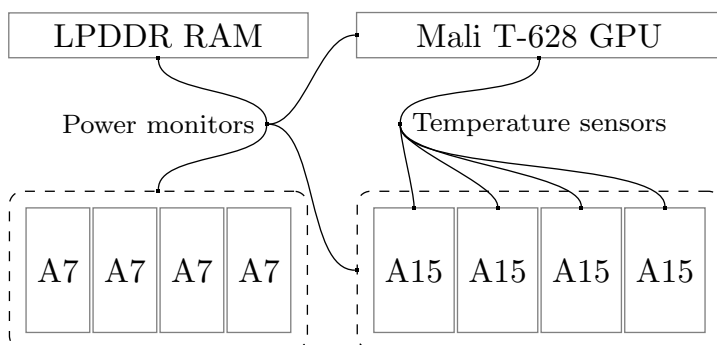


Figure 2.9: The power monitors on the Odroid-XU3 measures power usage for the memory, GPU, small cores and big cores separately. Temperature sensors measure temperature for each of the big cores and the GPU, but not the small cores.

spent and energy consumed by the program, shown in Fig. 2.10. A lower EDP is generally better.

$$EDP = E * T \quad (2.1)$$

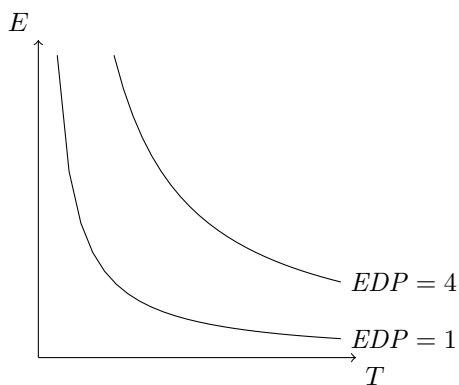


Figure 2.10: A visualization of how time and energy affects the EDP.

The measured power usage is affected by other processes running simultaneously on the XU3-board at the same time. Other programs might have unpredictable system resource usage patterns, possibly resulting in a large impact on the measurement stability. This was shown by Støa and Follan by reducing the relative standard deviation (RSD) from 2.0% to 0.15% by clearing the cache and disabling the *lightDM* display manager [SF15].

The Odroid XU3-board provides five temperature measurement sensors, one for each of the big cores, and one for the GPU, shown in Fig. 2.9. The small cores do not have a temperature sensor attached, making temperature adjustment for these cores impossible.

2.1.6 Program Evaluation Security

The largest security threat to the CMB system is running code from untrusted users who may have malicious intents. Security is therefore treated specially in this subsection. Properly securing the evaluation of programs is of high importance to the CMB project. [For06] lists examples of typical attacks on OJs. The most relevant categories of attack for the CMB system is listed below.

DoS attack A *denial-of-service (DoS) attack* denies legitimate users access or use of information systems or resources due to the actions of a malicious user. This is normally accomplished by excessive resource usage (e.g., CPU, memory, disk or network) until the target crashes or becomes unable to respond.

The CMB system can be exposed to DoS attacks by users submitting programs that compile or run for a long time, or use excessive amounts of memory during the compilation of the program. Excessive resource use may cause the server or the back end to crash, possibly requiring manual reboot.

Privileges escalation attack A *privileges escalation attack* occurs when a malicious user exploits a flaw in the system to gain access to restricted areas, such as files, folders or network resources.

In the context of the CMB back end this kind of attack might include users getting access to checker files, the correct answers or solutions of other users.

Destructive attacks A *destructive attack* occurs when a malicious user modifies or harms the environment in some way. This may involve deleting files, modifying files or replacing files.

The simplest attacks affecting the CMB system involves trying to delete all files, which, if it succeeds, causes denial of service. More subtle attacks include replacing the checker scripts or other system utilities to trick the system (e.g., replacing the checker executable with another executable which always returns “success”).

Covert channel exploits Users exploiting *covert channels* can use feedback from the system to gain unauthorized access to information.

The CMB system users can get information about test data by failing in specific ways. [For06] has example code where verdicts and memory consumption can be used to gain seven bits of data from a system that reports memory consumption. Such exploits, though not damaging to the system, gives users

an unfair advantage, and must be considered when giving feedback. The CMB system is especially vulnerable since it only has one hidden test case.

A typical OJ only evaluates programs on a single core, and restricts system calls such as `fork()` due to security concerns [For06]. The CMB system has—by design—a less restrictive system, because we want users to use multiple cores in order to achieve higher energy efficiency. This increases the number of possible exploits, and makes it very important that developers consider the security impact of changes. Additionally, when implementing security measures the effect on measurement stability must be taken into consideration—some security software may use an unpredictable amount of resources.

The CMB system has some security measures in place. Programs are run as a *worker*-user, with limited privileges, and the permission bits disable read and/or writes for this user. The solution for the large measurement test is never stored on the back end, and is therefore not available to the programs. These mitigations limits the potential of privileges escalation and destructive attacks but is not a perfect solution: the developer must remember to set correct file permissions to new files, and the user still has read and execute access to a large part of the file system, for example, `ls` and other utilities may be used to give the user information as part of a covert channel exploit.

Another security measure is that the program is killed if it uses more than 90 seconds to complete. This time limit is primarily to stop programs from running in infinite loops, as it is not enough to prevent denial-of-service attacks: a user could create a significant queue by submitting programs that succeed the correctness test, but times out on the large measurement test (100 submissions would take more than three hours to evaluate).

Compilation steps are not secured, and the compiler has full (non-root) access to the file system. This creates multiple opportunities to execute denial-of-service attacks, both on the server and on the back end.

When users profile their code, the code runs completely unsecured, meaning that the files and programs used to evaluate user submissions can be modified. An exploit is shown in Listing 2.2. If the user modifies the system, it has to be fixed manually by the CMB team.

```
#include <iostream>
#include <chrono>
using namespace std;

// Modify these
// Currently changes behavior after 15:00
const int hours = 15;
const int minutes = 0;

using Clock = chrono::system_clock;

int main() {
    auto now = Clock::to_time_t(Clock::now());
    struct tm *parts = std::localtime(&now);
    if (parts->tm_hour >= hours && parts->tm_min >= minutes) {
        // remove the script used to evaluate user programs
        system("rm ../../runscript_v2.sh");
    }

    cout << "Hello Italy!\nGreetings from the "
          "CMB-team at NTNU in Trondheim, Norway.\n";
}
```

Listing 2.2: An exploit for the unsecured profiling of code that removes the script used to evaluate submissions. By uploading and running the program before a specified time of day (here: 15:00), and profiling it after, the time of day can be used to determine whether the code is being profiled or not, and alter the behavior accordingly.

2.2 Related Work

This section presents other OJs which are popular and in use today. These OJs have many users and submissions, but none of them measure energy efficiency. Other OJs may, however, serve as an inspiration for the CMB project, and might impact the direction of the future development of the project.

Wasik et al. [WAB+18] provides a survey of OJs and divides them into four categories—online compilers; data mining, education and competitive programming platforms; recruitment platforms; and development platforms—described below. The CMB system belongs in the category of educational and competitive programming, because it has features for managing competitions, a problem archive and strives to be a learning platform for energy-efficient programming.

Online compilers

Online compilers are systems only allowing compilation (and in some cases running) of an arbitrary program in a supported language. These systems do not have problem statements, nor evaluate the user-uploaded code.

Data mining, education and competitive programming platforms

These platforms extend upon online compilers by evaluating solutions uploaded by users, usually in terms of correctness and running time.

Data mining platforms are often focused on data classification. Some platforms only require users to run their code locally and upload the resulting data set for evaluation.

Educational platforms focus on learning and educational processes, and often have features allowing users to share solutions and learn from solutions submitted by other users. Managing (university) courses are also a common feature for such platforms. Some systems have an achievement or progress tracking system to motivate users to do tasks.

Competitive programming platforms primarily focus on competitions. These platforms typically rank users based on the number of problems solved (on user rankings) and the run time of their solutions (on the ranking for a given problem).

Recruitment platforms

These platforms are usually similar to competitive programming platforms, but facilitate recruitment of software developers. This is typically done by letting companies looking for developers publish their own problems to potential applicants, and implementing special functionality to filter applicants.

Some recruitment platforms incorporate interviewing software, often including an integrated code editor shared with both the interviewer and the interviewee, as well as video call support for remote interviews.

Development platforms

Systems in this category are available to download. These systems are usually open source or provided as binary archives. These are available for anyone who wants to download and deploy their own OJ locally.

The following subsections will look at some OJs with features that are interesting for the CMB project.

2.2.1 Selected Educational Online Judges

Educational OJs are interesting because they serve as an inspiration for the educational aspect of the CMB system. The features described here could be considered for the CMB project.

2.2.1.1 Jutge.org

Jutge.org [Jut] has an educational focus, and was created by Universitat Politècnica de Catalunya (UPC) in 2006 [PGR12]. The OJ allows for anyone to register and practice programming. It has over 3000 problems, and a total of 2.7 million submissions.

It is designed for use as part of university courses, with many features helping course instructors manage and assess their class. Some features available for instructors include assignment management, instant statistics about students, sharing of lists and documents relevant for the course, and creating course specific problem sets, contest or even exams. Other features like *supervision* allow instructors to easily view and inspect their students' submissions to provide feedback and help.

2.2.1.2 Codecademy

Codecademy [Coda] tries to create a more engaging educational experience through their platform. Their web platform features step-wise tutorials (compared to standalone exercises), an online integrated code editor which runs the code, a terminal, and other tools required for practicing programming skills. The system allows for multiple interactive activities like multiple choice quizzes, free-form projects, and video clips with explanation of both exercise and concepts, some of which are shown in Fig. 2.11.

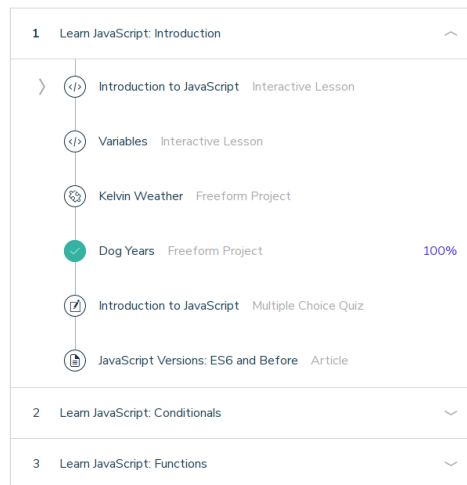


Figure 2.11: A screenshot from Codecademy, showing the beginning of the introductory JavaScript tutorial [Coda].

2.2.1.3 Codewars

Codewars is a community-driven OJ, where all users may create and upload their own problems (called “kata”). The users complete kata and earn higher rankings when they have completed a sufficient amount of kata. After completing a kata, the user is shown solutions by other users. All solutions are ranked by cleverness and how well they follow best practices, depending on how many times they are voted on by other users. This allows users to view submissions by other users and learn from their code. A screenshot displaying this feature is shown in Fig. 2.12.

Codewars supports a wide range of languages, and also supports multithreaded applications if the language used has built-in support for it (i.e., no external libraries for multithreading are used). The Codewars website also has an advanced integrated code editor. This editor can be customized by the user, and even lets users write their own test cases. This editor is shown in Fig. 2.13.

2.2.2 Selected Programming Contest Platforms

The CMB system is currently closer to the programming contest OJs, than any other category. Other OJs in this category have a more complete set of features compared to CMB, and should be studied for inspiration.

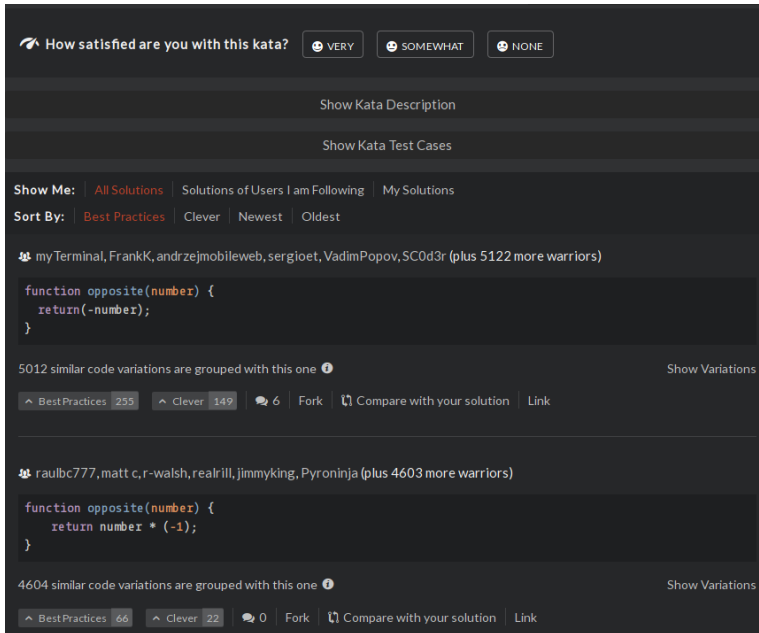


Figure 2.12: Solutions are shown to the users after solving a kata (from [Codb])

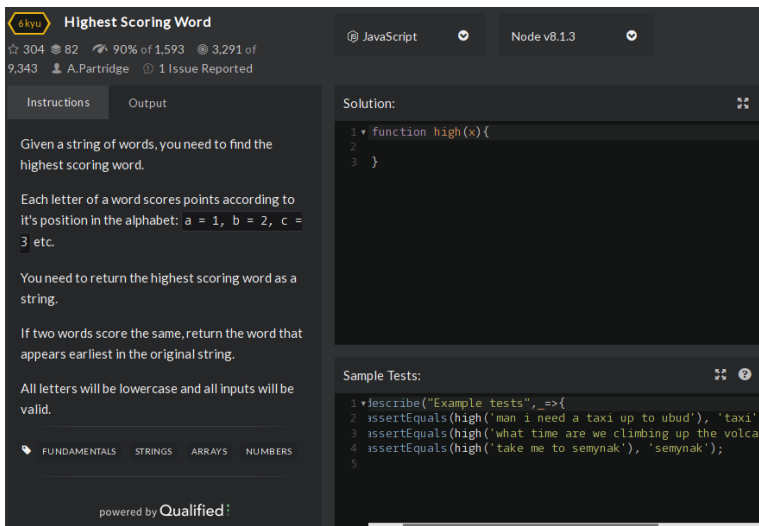


Figure 2.13: Codewars [Codb] has an editor integrated in the website, where users also may write their own test cases.

2.2.2.1 UVa Online Judge

UVa Online Judge [Uva] is one of the largest and oldest online judges, created in 1995 by Ciriaco García de Celis from the University of Valladolid in Spain [RML08]. The system has been public since 1997, and over 20 years later, it has close to two million submissions each year [Uva]. It features archives of problem sets from programming competitions such as ACM International Collegiate Programming Contest, in addition to many other problems, totalling over 5000 problems [WAB+18].

uHunt [Uhu] is an interesting complementary tool for the UVa OJ, providing live statistics, problem categorization and easy access to previous submissions and user rankings. A snippet is shown in Fig. 2.14.



Figure 2.14: A screenshot from the uHunt tool [Uhu] for UVa.

2.2.2.2 Kattis

Kattis [Kat] is an OJ developed by KTH — Royal Institute of Technology in Sweden, and have been used to asses programming exercises in their courses since 2005 [EKN+11]. Enström, Kreitz, Niemelä, Söderman, and Kann argues for *test-driven education*, encouraging a test-driven development process for students. They envision Kattis as an adversary in this context, failing programs that do not give the correct answers.

Even though Kattis can be used for educational purposes, it is most well known as a contest system for many programming competitions. The ACM International Collegiate Programming Contest (ICPC) [Icp] is probably the most well-known

competition, where Kattis is used both for some of the regional qualifiers, as well as the world finals [EKN+11].

Kattis also has some additional features, including a command line client for power users, an online code editor and per-country or per-university ranking lists. Kattis has some recruitment and company-sponsored problems and could be classified as a recruitment platform in this regard.

2.2.3 Selected Open Source Online Judges

Open source OJs are interesting for the CMB project, because they may serve as inspiration improving the security measures of the system and when implementing language support for new languages. These OJs might be released with software licenses that restrict copying or otherwise reusing the code.

2.2.3.1 DOMjudge

DOMjudge [Doma] is an automated judge system which is primarily used in programming contests like ICPC, with on-site teams, fixed problem sets and a given time frame. It has automatic judging and separate web interfaces for the teams, the jury and the general public. DOMjudge can be setup locally on a server, is modular and open for extension and supports any language (compilers and interpreters must be supplied as part of the setup).

This judge uses low-level OS utilities such as chroot, setrlimit, cgroups and signals to secure the program [Domb], which allows it to have more fine-grained control over program execution.

The DOMjudge source code is licensed under the GNU General Public License (GPL), version 2 or later, which implies that the CMB system may copy and modify source code from this project, as long as we don't distribute the CMB system (e.g., in the form of binary archives). But if the CMB system is distributed using code licensed under the GPL, the source code of the CMB system (or possibly, the subsystem using the code) must be made available and licensed under the GPL.

2.2.3.2 INGIInious

INGIInious [Inga] is an automated exercise assessment platform which is intended for educational use, created by Université catholique de Louvain, in Belgium. It

has features to run and grade student code in secured environments, and teacher and instructor roles to easily manage and monitor the progression of classes.

The system uses Docker [Doc] and SELinux [Sel] to securely run students' code [Ingb]. It is possible to customize this setup to include support for any language. An example of this is the new grading system in TDT4120 [Tdtb] at IDI, NTNU, which supports Julia.

The Inginious system is licensed under the GNU Affero General Public License (AGPL), a stricter form of the GPL. The AGPL counts use over networks as distribution and will in some cases require all the source code in the CMB system to be made available and licensed under the AGPL if code is reused from a project with this license.

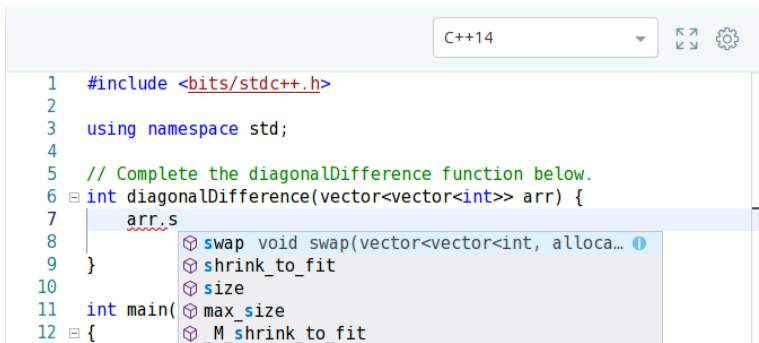
2.2.4 Selected Recruitment Platforms

Recruitment facilitation and profiling of companies are currently not planned the by CMB project. However, the OJs designed for helping companies recruit applicants often have more feature-rich environments, which is of interest.

2.2.4.1 HackerRank

HackerRank [Hac] defines itself as a “technological hiring platform”. They started in 2009 as a startup, and has since grown to over four million users. The platform hosts many programming problems and programming competitions, as well as tutorials in programming languages and general computer science topics. HackerRank provides interview and assessment tools which companies can use to track and evaluate candidates, and to perform technical interviews on the platform.

The code editor used at HackerRank provides multiple features not commonly seen on other platforms. The editor is initialized with templates, which allows the user to ignore minor details of the problem, such as reading input and writing output. Additionally, the editor provides automatic completion of code, helping the user to write code faster. This is shown in Fig. 2.15.

A screenshot of the HackerRank editor interface. The top right corner shows 'C++14' and some utility icons. The code editor contains the following C++ code:

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 // Complete the diagonalDifference function below.
6 int diagonalDifference(vector<vector<int>> arr) {
7     arr.s
8 }
9
10
11 int main() {
12     int n;
13     vector<vector<int>> arr;
```

An autocomplete dropdown menu is open over the 'arr.s' text on line 7. The menu lists several methods: 'swap void swap(vector<vector<int, alloca...', 'shrink_to_fit', 'size', 'max_size', and '_M_shrink_to_fit'. Each item has a small icon to its left.

Figure 2.15: A screenshot of the HackerRank editor, showing the autocompletion feature.

2.3 Summary

This chapter gave an overview of the CMB system as well as various OJs that are available and CMB can learn from. The main aspects of these OJs that are relevant for further progressing the CMB project include class management, solution sharing, statistics for users, command line client, online code editors and security measures.

Chapter 3

CMB Challenge 2019

The CMB Challenge 2019 was organized as part of the thesis work. This competition is for students in the course TDT4102 Procedural and Object-Oriented Programming, an introductory C++ programming course at NTNU attended by around 850 students [Tdda].

The TDT4102 course is primarily taken by engineering students who do not belong to a computer science programme. The majority of the students belong to the programmes Applied Physics and Mathematics, Cybernetics and Robotics, Electronics System Design and Innovation or Energy and Environmental Engineering. Most students take the TDT4102 course as their second programming course, and have not taken courses on algorithm and data structure theory or parallel programming.

The competition is fully optional for the students. This year's competition had 16 contestants who solved at least one problem, submitting a total of 665 programs. The competition lasted for three weeks, near the end of the course.

The competition was organized using system version three with a couple of small bug fixes (to prevent server failures), in other words, no major changes made to the system as part of this thesis were used when running the contest.

This chapter describes the competition format and the problems used, as well as general directions for how the problems could be solved. Further, some generalizable solution improvement strategies used by the contestants are described. Finally, the feedback received from contestants are discussed and some general pointers on how to improve future competitions using the CMB system are suggested.

3.1 Competition Format

The competition had six problems, ranging from simple counting problems to NP-Hard problems, briefly described in Section 3.2. The problems and the code used for random number generation were written by a course TA during the summer of 2018. We reviewed the problems and made them available on the CMB system before the competition.

The students got one point for each problem they solved. The three fastest submissions and the three most energy-energy efficient submissions (using EDP) got up to three extra points, making seven points the maximum score for each problem. The “There and Back Again” problem used a different scoring system when ranking submissions, described further in Section 3.2.6.

The CMB Challenge 2018 experienced troubles with I/O-bound problems, shifting the focus from solving the problem faster to reading input faster [NSLH19]. This led to trying a new approach in this year’s competition, using pseudo-random number generation, to generate input for five of the six problems. Listing 3.1 shows the code given to the students to generate random numbers.

The class used for random number generation is designed to be a simple, but portable implementation, in other words, it must give the same results on the students’ computers and on the CMB back end when given the same seed. C++ standard library facilities for random number generation include `rand()` and the `<random>` library. These libraries are (partially) non-portable, and were therefore not used. Specifically, for the `<random>` library, the random number generator algorithms are specified and therefore portable [Cppb, ch. 26.5.3], but the random number distribution algorithms are implementation-defined [Cppb, ch. 26.5.8.1, 3] and gives different results depending on the compiler used.

```
#include <algorithm>
using namespace std;
class RandInt {
private:
    static const unsigned int INCREMENT = 0xC39EC3;
    static const unsigned int MULTIPLIER = 0x43FD43FD;
    unsigned int m_nRnd;
public:
    RandInt(unsigned int nSeed) : m_nRnd(nSeed) {}
    int getInt(int nFrom, int nTo) {
        if (nTo < nFrom)
            swap(nTo, nFrom);
        else if (nTo == nFrom)
            return nTo;
        m_nRnd = (m_nRnd * MULTIPLIER + INCREMENT) & 0xFFFFFFFF;
        float fTmp = (float)m_nRnd / 16777216.0;
        return (int)((fTmp * (nTo - nFrom + 1)) + nFrom);
    }
};
```

Listing 3.1: The `RandInt` class was handed out to the students for pseudo-random number generation. Instances are instantiated using a seed, and random integers are generated in the closed range $[a, b]$ calling the public method `getInt` with arguments `a` and `b`.

3.2 Problem Statements

This section briefly describes each problem with proposed solution strategies. The full problem descriptions are provided in Appendix B.2. General strategies for improving solutions used by the students are described in Section 3.3.

3.2.1 To Quote Hamlet...

This problem is meant to be a simple problem that all students should be able to complete with relatively little effort. To solve the problem, the program must generate N (given by input) numbers in the closed range $[0, 19]$, find the number that occurs most frequently, and print this number along with a corresponding name (given in the problem statement). An excerpt from a possible solution is given in Listing 3.2.

3.2.2 Pirates and Probabilities

In this problem, the solutions must order a set of islands based on a given scoring criteria. The five best and worst candidate islands must be printed.

To solve the problem, a submitted program must generate three random numbers for each of the N islands. These random numbers represent some scoring for individual properties of a given island, and are combined to generate a “total score” for the island. The total score is used together with the associated ID of an island (to break ties) to order the islands. Finally, the best and worst five candidate islands must be printed together with the total score and the score of the individual properties.

The main challenge in this problem is the choice of data structures. Storing all islands might be easier, but slower than only storing the top and bottom five islands encountered so far. Since the ordering of islands depends on both total scores and the islands’ IDs, it is necessary to either customize STL data structures (e.g., by specifying custom compare functions) or write a custom ordering algorithm to correctly order the islands.

3.2.3 The Huckybucky Forest

To solve this problem the submission must calculate the CO_2 emission increase associated with moving from the city into the Huckybucky Forest for M years, and

```

vector<string> names{ /* ... all names ... */ };
vector<int> counts(20, 0);
RandInt gen{S};
for (int i = 0; i < N; i++) {
    ++counts[gen.getInt(0,19)];
}
auto mx = max_element(begin(counts), end(counts));
cout << names[distance(begin(counts), mx)] << " " << *mx << "\n";

```

Listing 3.2: An excerpt from a solution to the “To Quote Hamlet” problem. S and N are read from standard input. The names are given in the problem description, but are omitted for brevity.

also which months the CO_2 emissions were lower in the forest compared to living in the city.

This problem requires generating a lot of random numbers, performing calculations, and has some tricky output format requirements, but is otherwise straight-forward to solve. The main improvements here are achieved by performing calculations on the fly, when possible, instead of storing every generated random number.

3.2.4 In Ventus

This problem is a thinly veiled maximum flow problem. The program must generate an electrical grid from the random numbers, and find the maximum power that can be transported from a newly connected wind turbine to the main grid.

This problem requires the students to implement any maximum flow algorithm to solve the problem. The contestants that solved this problem implemented either the Edmonds-Karp algorithm [EK72], or Dinic’s algorithm [Din70]. There were significant differences in timing measurements (approximately a factor of two) between different implementations of Edmonds-Karp, showing that choices of data structures and effective traversal is important.

3.2.5 Flower Power

This problem uses random numbers to generate coordinates of rare flower occurrences in a large ($W \times H$) area. The user have to find a ($w \times h$) subarea that contains at least k occurrences of rare flowers. This area is a potential nature reserve.

Halim et al. defines this problem as a variant of the “Max 2D Range Sum” problem [HHSR13]. Naive solutions use four nested loops, running in $O(WHwh)$. The fastest algorithms pre-compute a cumulative two-dimensional array using the inclusion-exclusion principle to achieve a speed-up to $O(WH)$.

3.2.6 There and Back Again

This problem is simply an Euclidean variant of Traveling Salesperson Problem (TSP), where the students must implement an approximation algorithm to find a near-optimal tour in order to solve the problem. It does not depend on random number generation, but gives city coordinates as input. Because it is defined as an approximation problem, all permutations of cities are valid solutions. The tour length of a permutation becomes the “goodness” score displayed on the scoreboard. A combined score used to rank the submissions is calculated by multiplying the goodness with the EDP. Six, four and two points are given to the top three students.

The top students used a greedy approach, opting for a low EDP over more optimal tours. Some students tried known shortest path algorithms combined with heuristics, which resulted in a slightly more optimal tours, but significantly worse EDPs.

3.3 Solution Improvements

The solutions described for the problems are simple, single-threaded solutions. To improve the performance, choice of algorithms and data structures are probably the most important factors. Some students submitted single-threaded programs that were heavily optimized. However, the students were not limited to using a single core, and some students used multithreading successfully. A couple of students also found ways to use weaknesses in the system, detailed in Section 3.3.4. The following subsections go into more detail on different solution strategies.

3.3.1 Loop Unrolling

Loop unrolling had a surprisingly large effect on the run time of programs, improving the run time by a factor of four compared to the straight-forward solution. A snippet of code showing loop unrolling is shown in Listing 3.3.

Loop unrolling is a well-known compiler optimization [ALSU07], but GCC (version 4.9) does not take advantage of this optimization unless explicitly enabled. An

```

for (int i = 0; i < N/10; i++) {
    ++counts[gen.getInt(0,19)]; ++counts[gen.getInt(0,19)];
    ++counts[gen.getInt(0,19)]; ++counts[gen.getInt(0,19)];
    ++counts[gen.getInt(0,19)]; ++counts[gen.getInt(0,19)];
    ++counts[gen.getInt(0,19)]; ++counts[gen.getInt(0,19)];
    ++counts[gen.getInt(0,19)]; ++counts[gen.getInt(0,19)];
}

for (int i = 0; i < N%10; i++) {
    ++counts[gen.getInt(0,19)];
}

```

Listing 3.3: The loop unroll optimization written by a student. This code is a replacement for the loop in Listing 3.2.

alternative to manual loop unrolling is using *Function specific option pragmas* or *Function attributes* for setting optimization flags (e.g., the `unroll-loops` flag) for a single function [Gcc].

3.3.2 Task and Data Parallelism

Several students tried multithreaded approaches. Common approaches such as data parallelism or task parallelism were used by all those that tried multithreaded approaches, with varying degrees of success.

One example of the more successful approaches of data parallelism (not counting the “Skip Seed” method in Section 3.3.3) was using OpenMP for the Flower Power problem. Dividing the search between multiple threads resulted in programs that were orders of magnitude faster, only adding a single line of code.

Task parallelism was used successfully in the Huckybucky Forest problem, by keeping track of the best and worst islands in separate threads, resulting in a moderate speed-up.

3.3.3 “Skip Seed”

The most surprising method of speeding up programs that some students used modified the random number generation method, exploiting the pseudo-randomness of the random number generators. This method was named “Skip Seed”.

The “Skip Seed” method divides the n random numbers, Y_i , into k groups of size n/k , $\{\{Y_1, \dots, Y_{n/k}\}, \{Y_{n/k+1}, \dots, Y_{2n/k}\}, \dots, \{Y_{(k-1)n/k+1}, \dots, Y_n\}\}$. The starting seeds, X_i , for each of the k groups are calculated and used in separate random number engines (e.g., k instances of the `RandInt` class are instantiated).

A non-parallelized version of “Skip Seed” can be used if the random numbers have to be combined in some way to get a useful value. For example, in the Pirates and Probability problem, the properties, p_{1i}, p_{2i}, p_{3i} , for each island, i , is generated in the order $p_{11}, \dots, p_{1N}, p_{21}, \dots, p_{2N}, p_{31}, \dots, p_{3N}$, meaning that the numbers have to be stored before doing computations. However, by using “Skip Seed” to create three separate random engines these computations can be done on the fly, lessening storage requirements and time used to store and fetch data.

When used together with data parallelism, the “Skip Seed” method improved a program’s run time considerably, up to a factor of four depending on the number of threads. The parallelized “Skip Seed” method starts k separate threads that perform computations (one thread for each of the k groups), and the results are combined when every thread have finished. Listing 3.4 shows an example of using the parallelized “Skip Seed” method.

A linear congruential generator (LCG) is used to generate integers, X_i , by using Eq. (3.1a) with $m = 1000000_{16}$, $a = \text{FD43FD}_{16}$ and $c = \text{C39EC3}_{16}$ ¹. Since $m = 2^{24}$ Eq. (3.1a) can be simplified to Eq. (3.1b), which replaces the modulus operation with a faster bitwise AND operation.

$$X_{i+1} = (aX_i + c) \bmod m \tag{3.1a}$$

$$X_{i+1} = (aX_i + c) \wedge (m - 1) \quad \text{if } m = 2^n \quad n \in \mathbb{N} \tag{3.1b}$$

The randomly generated integer $X_i \in [0, m)$ must be transformed into an integer $Y_i \in [a, b]$ —by using Eq. (3.2)—before using it in calculations.

$$Y_i = \lfloor \frac{X_i}{m}(b - a + 1) + a \rfloor \tag{3.2}$$

The key insight for this method is that X_{i+1} does not depend on Y_i (visualized in Fig. 3.1). The calculation $X_i \Rightarrow X_{i+1}$ (Eq. (3.1b)), which only depends on relatively simple operations on integers, is much faster than the floating point calculation $X_i \Rightarrow Y_i$ (Eq. (3.2)), making this method feasible even when the X_i s are computed twice.

¹These numbers differ somewhat from Listing 3.1. Note that $m = 1000000_{16} = 16777216$, and that $43\text{FD}43\text{FD}_{16} \equiv \text{FD}43\text{FD}_{16} \pmod{1000000_{16}}$.


```

constexpr int num_threads = 4;
constexpr int persons = 20;
unsigned int counts[num_threads][persons];

unsigned skipSeed(unsigned seed, int n) {
    for (int i = 0; i < n; i++)
        //  $X_{i+1} = (aX_i + c) \wedge m$ 
        seed = (seed*0x43FD43FD + 0xC39EC3) & 0xFFFFFFFF;
    return seed;
}

void calc(int t_id, int seed, int n) {
    RandInt gen(seed);
    for (int i = 0; i < n; i++)
        ++counts[t_id][gen.getInt(0, persons-1)];
}

int main() {
    // reading variables (N and S) from stdin skipped for brevity

    vector<thread> threads;
    const int skip = N/num_threads;
    for (int i = 0; i < num_threads-1; i++) {
        threads.emplace_back(calc, i, S, skip); // start a new thread
        S = skipSeed(S, skip); // calculate next seed
    }

    // use main thread instead of starting a new thread
    calc(num_threads-1, S, N-skip*(num_threads-1));

    for (auto& th : threads)
        th.join();

    // sum counts calculated by each thread
    for (int i = 0; i < persons; i++)
        for (int j = 1; j < num_threads; j++)
            counts[0][i] += counts[j][i];

    auto mx = max_element(counts[0], counts[0]+persons);
    cout << names[distance(counts[0], mx)] << " " << *mx << "\n";
}

```

Listing 3.4: This listing extends upon the code from Listing 3.2 by using the “Skip Seed” method for parallelizing random number generation.

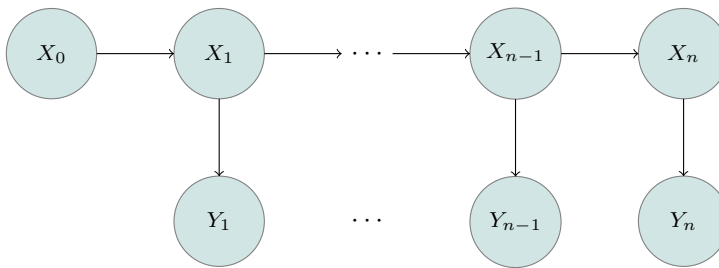


Figure 3.1: The state machine for generating random numbers Y_i with starting seed X_0 .

```
if (N == 23) {
    cout << "Lady Trent 3\n";
    return 0;
}
int count = 0;
for (int i = 0; i < N; i++) {
    if (gen.getInt(0,19) == 6)
        ++count;
}
cout << "Vera Stanhope " << count << '\n';
```

Listing 3.5: The student knows the input and solution of the small test case and prints the solution directly. For the large test case a partial answer has been guessed, and only the relevant values are counted, resulting in increased performance.

3.3.4 Miscellaneous Methods

For some problems it is feasible to guess partial solutions for the measurement test case, making it easy to get a faster solution. One such problem is shown in Listing 3.5. This flaw is exploitable because the system only has a small test case, and a big measurement test case. Mitigations are discussed in Section 7.4.2.1.

Previous competitions have shown that I/O operations can greatly affect run time. The simplest improvement is by calling `ios::sync_with_stdio(false)` at the start of the program to disable syncing between buffers in the `<iostream>` and `<cstdio>` libraries, and then using `cout/cin` or `printf/scanf`. A more complicated way of improving is using the (thread-unsafe) `_unlocked` variants of `getchar` and `putchar`. Results from CMB Challenge 2018 show that this is the most performant solution [NSLH19]. Only the simple approach was used by contestants in this year's competition.

3.4 Questionnaire

A questionnaire was published for the students participating in the competition. Only eight students responded. Data and insights from the responses that may be relevant for future competitions will be discussed in this section. The questionnaire is included in Appendix B.3.

When asked about how well generating random numbers worked, 75% of the respondents answered “Good” or “Very good” (Q6). All respondents preferred generating random numbers over reading numbers from standard input (Q7).

All respondents responded that feedback about run time and energy usage gave at least some insights into their own code (Q11).

62.5% of respondents rated the usability of the website as “Good” or “Very good” (Q8). Three respondents noted that the system is slow (Q9). This is a well known problem, but has been fixed as part of this thesis (Section 4.6). One respondent also noted that it wasn’t clear that they would have to sign up in a group to participate in the competition (Q9). Although this was clearly stated in the information published to the students (see Appendix B.1), the system’s UI could be improved to avoid misunderstanding.

When questioned about the use of CMB in the TDT4102 course and its effects on their motivation for the course and future courses (Q12), the respondents were generally positive about the competition, finding it to be an interesting addition to course assignments, even if it was not directly relevant for the exam.

3.5 Summary

The CMB Challenge 2019 was overall successful, improving on some of the weaknesses discussed in [NSLH19]. Since performance did not depend on fast I/O, we saw the students trying entirely new approaches compared to previous competitions (e.g., parallelizing their code). We also received valuable feedback from the students about the CMB system.

The random number generation approach worked well, but for the easier problems the number generation might dominate the run time. Future competitions should test multiple approaches for generating random numbers to find a method that is performant and yields number with an acceptable quality—the numbers used in this competition do not have to be high-quality random numbers. The random engines provided by the `<random>` library can be used for this (see Listing 3.6). Especially the engines based on `subtract_with_carry_engine` are fast [Cppa].

```
#include <random>
using namespace std;
using engine = linear_congruential_engine<unsigned, 0xFD43FD,
                                           0xC39EC3, 0x1000000>;

class RandInt {
    engine m_eng;
    constexpr static auto range = engine::max() - engine::min() + 1;
public:
    RandInt(unsigned int nSeed)
        : m_eng(nSeed)
    {}

    int getInt(int nFrom, int nTo) {
        int num = m_eng();
        float fTmp = (float)num / range;
        return (int)((fTmp * (nTo - nFrom + 1)) + nFrom);
    }
};
```

Listing 3.6: This is a functionally equivalent `RandInt` implementation to the implementation in Listing 3.1, provided as an example of using portable STL functionality on platforms supporting C++11.

Chapter 4

Implementation

This chapter describes the work done to fulfill the goals of this thesis. The following sections describe all the improvements made to the system during the thesis work. The sections divides the improvements topically rather than chronologically.

During development, much effort has been put into making improvements that are sustainable in the long term. This implies that changes that break APIs, or might make the CMB system more difficult to maintain in the future have been avoided. In particular, “stitching” new features on top of already hard-to-maintain code has been avoided, therefore, features have only been added if there was sufficient time to properly refactor and clean the related code.

4.1 Operating System Upgrade

Both the server and the back end ran on some variant of Ubuntu 14.04 LTS before the start of this project. The versions of Ubuntu that have long-term support (LTS) are supported for five years, and therefore the OS version used would reach end-of-life (EOL) in April, 2019¹, meaning no additional security patches would be released. During the specialization project it was decided that upgrading the OS on both the back end and server to the latest LTS version, 18.04, was of high priority.

¹Ubuntu version numbers indicate which month and year the software was released; Ubuntu 14.04 was released in April, 2014.

The Ubuntu distributions for the Odroid-XU3 back end is built and distributed by Hardkernel. The Odroid-XU3 previously ran on the Ubuntu 14.04 LTS distribution, in other words, Ubuntu with a LXDE desktop environment. However, Ubuntu 18.04 Minimal—without a graphical desktop environment—was chosen as the new OS. Choosing Minimal Ubuntu [Min] results in a system with fewer extra background processes that may interfere with measurements (e.g., *LightDM* removed by Støa and Follan as part of their thesis work [SF15]). The Ubuntu 18.04 Minimal releases distributed are targeted for Odroid-XU4—a newer version in the XU series, without power monitoring—but are fully compatible with the Odroid-XU3.

Ubuntu 14.04 LTS for the Odroid-XU3 is built upon version 3.10 of the Linux kernel, while Ubuntu 18.04 LTS for the Odroid-XU3 is built upon version 4.9 of the Linux kernel. The 4.9 mainline kernel is a long-term maintenance release, with kernel developers fixing bugs until the projected EOL in 2023. Using a long-term release is beneficial because it provides bug fixes and security patches, avoiding the need to upgrade the kernel frequently.

There are some differences between the OSs that affect the CMB system. The way to read temperatures have changed. The previous OS version stored the temperatures of the four “big” cores and the GPU in a single file, while the new version divides into *thermal zones* that store these values one-per-file in separate directories. A thermal zone manages one thermal sensor and the behavior of associated cooling devices (e.g., fans). Thermal zones can be managed through the *sysfs interface* [Sysa]. For the new OS version thermal zones 0–3 are associated with the temperature sensors for each of the “big” cores, and thermal zone 4 is associated with the temperature sensor for the GPU.

The new OS also used a newer compiler. Previously, GCC-4.9 was used, but the new OS uses GCC-7.4 by default. GCC-7.4 compiles to C++14 by default (previously C++11), in addition to supporting OpenMP 4.5 (previously OpenMP 4.0) and some C++17 features (if the `-std=c++17` flag is given) [Opeb].

A software development kit (SDK) provided by ARM was previously used to provide OpenCL support, but this is now replaced by packages from the Ubuntu Package archive to simplify the installation. This has the additional benefits of providing the OpenCL C++-bindings [Opea], and support for OpenCL 1.2 (previously OpenCL 1.1).

A great amount of effort has been put into not making more changes to the back end than necessary when upgrading the OS version (beyond the changes described previously in this section) in order to create a *baseline* that is comparable to the old system when performing the measurements.

The upgrade of the server OS was mostly similar to the upgrade of the back end OS. The server side code mostly depends on Python, and has fewer OS-specific

requirements. The server must use the same compiler version and OpenCL bindings as the back end for the server-side compilation steps to work correctly.

4.2 Implementing Tests on the Back End

No tests were implemented for the back end previous to this thesis. It was of high priority to implement tests verifying correctness of the scripts running on the back end to ease development and improve maintainability. These tests can be used to detect and avoid errors in related parts of the code base when adding new features.

The code on the back end consists mostly of Bash scripts, with some Python scripts for data manipulation and calculation. The combination of languages being “glued” together this way makes testing harder, but implementing and maintaining the data manipulation and calculation scripts in Python is easier, and outweighs the drawbacks. Three approaches to testing were considered: manually implementing tests in Bash and/or Python, low-level testing using a unit testing framework for Python and/or Bash, or high-level testing using Python. These three approaches are discussed below.

The first approach, *implementing tests manually*, would have the benefit of being able to test functions and scripts at an appropriate level, especially if implementing tests in both Bash and Python. However, not using test frameworks adds some overhead to developing and running tests, and might become unmanageable when a large number of tests are implemented. Some common benefits of test frameworks such as only running parts of the test suit (e.g., only run tests with the tag “OpenCL”), automatic test discovery, or only running the tests that failed last time (i.e., test failure tracking) would not be possible with this approach without “reinventing the wheel”. Since the thesis work has to be completed in a rather short amount of time and this would require more time to implement with no additional benefits, this approach was rejected.

The second approach, *implementing low-level unit tests with an appropriate test framework*, would provide most or all the benefits of using a test framework, depending on the test framework chosen. The unit testing framework *Bats: Bash Automated Testing System* for Bash was installed and tested for this purpose. However, the Bash scripts were written in a *monolithic* style, only having a single, long main function (if any function at all). The scripts have complex, and (previously) undocumented pre- and postconditions, requiring files in specific locations in specific formats. This makes simple unit testing hard; requiring complex and hard-to-maintain code to generate, parse and verify complex input/output file formats such as JSON using Bash and shell utilities. Implementing unit tests in Python would make it easier to work with complex file formats using simple, maintainable code, but would not be able to easily test single Bash functions when

refactoring the code. The process of testing every single script (in either Bash or Python) would also be rather time-consuming, but would make it easier to track down errors, compared to higher-level testing.

The third approach, *implementing high-level integration tests*², does not test a single unit (e.g., a script or a function), but the interface to the back end. The tests would use the same workflow as the server when it starts evaluation of programs on the back end: copy the required files (program source files, input files, answer to the correctness test and the Makefile) to the correct directory, run the entry-point script (`runscript_v2.sh`) and finally retrieve the output from the script. The tests then assert that the output is on a correct format and contains the expected result. Large parts of the code base can be covered by tests with relatively few tests when using this approach.

It is, however, harder to track down errors to a single file, function or script when using high-level testing. Making the scripts log extra information to the standard error stream (`stderr`) that can be used to locate the error help mitigate this problem. Testing from this abstraction level also ensures that the communication protocol between the server and the back end does not change (API compatibility).

The third approach was adopted in favor of the second approach because of the benefit of being able to implement tests covering large parts of the code base in a shorter amount of time, moreover, the second and third approach are not mutually exclusive; unit testing the code incrementally when adding new features in future is possible, even recommended [HT99; McC04]. Tests would then be written in an appropriate unit-testing framework for the language of the script or module to be tested.

The *pytest* framework [Pyta] is used to implement tests. This framework is fully compatible with Python's built-in unit test library, *unittest* [Uni], but have better automatic test discovery, and a test runner `pytest` that supports a wide range of options when running tests, such as selecting tests by tag or by name. Other important features of the *pytest* framework that are not part of the *unittest* library are flexible *test fixtures* and *test parametrization*. Test fixtures are functions or objects initializing the system to a predetermined state before testing; `setUp` and `tearDown` functions are examples of simple fixtures. Test parametrization is used to define multiple sets of input arguments to a test function or class.

Tests are written using multiple C++ programs implemented to test some specific property of the system (e.g., one test program tests if OpenMP works correctly, another runs in an infinite loop to test if the system stops programs that spend too much time). Programs testing error conditions also fail at different stages of the evaluation process; some programs fail during the correctness test, and some

²“Integration testing is the combined execution of two or more classes, packages, components, or subsystems [...]” [McC04, p. 499]

fail during the measurement test to properly check that errors are caught in both stages.

Several bugs related to error handling and faulty formatting in the existing system were fixed in the process of creating tests for the system. The tests also ensured that the existing features did not fail while developing new features, providing a quick and simple way to ensure correctness without spending time testing the system manually.

Flags to control the behavior of the program evaluation was added while developing tests (described in more detail in Section 4.4.4). One such flag was the `--no-energy` flag. This flag disables energy measurement (by returning the “dummy value” zero), making it possible to run the back end tests on a system without power monitors (or with different power monitors). This reduces the need of testing the code on the Odroid-XU3. The code can be tested on the developer’s computer during development, and tested again on the Odroid-XU3 after the development has been completed. This minimizes the time used for testing while developing new features.

4.3 Automating Installation on the Back End

In order to more easily install and configure software on the Odroid-XU3 boards an installation script that automates most of the steps required to set up a fully working back end was created. The main motivation for this effort was to reproduce the back-end setup faster and easier in a less error-prone way. The Odroid-XU3 connected to the production server still has artefacts (i.e., temporary files, packages and utilities installed, but eventually not used) from the work done by multiple previous developers on the CMB project. A simpler way to reinstall the system encourage frequent fresh installs of the system when changes are made, and avoids unnecessary files and packages clogging the system.

The steps to fully setup a working system as described in [Mag16; Ing17] are rather long, hard to reproduce, and partly no longer correct after upgrading to a new OS version. Some important security features such as properly setting the file permissions in order to restrict the *worker* user from reading and writing to files not needed are described in [SF15], but no steps to reproduce this are provided. The installation script tries to automate the setup of these security features.

The installation script is implemented in Bash and tries to group the setup process into functions (e.g., the steps to setup the *Uncomplicated Firewall* (`ufw`) utility is grouped in a single function). This makes it easier to see why dependencies are required, and simplifies removing programs or utilities and their dependencies in the future. The installation script is committed to the *cmb-board* Git repository as `install.sh`.

To speed up the installation process, the SciPy [Scia] and NumPy [Num] packages have been pre-built and uploaded to <http://folk.ntnu.no/okpeders/archives/>. This is not a feasible long-term solution, since this is a personal web page and only editable by the owner and not future developers on the project. A more permanent solution is discussed in Section 7.1.1.

4.4 Program Evaluation Improvements

The following subsections explain the implementation of the different approaches to improving the evaluation of programs on the back end. These approaches are divided into four categories: measurement stability, security, run-time error feedback, and refactorings and bug fixes.

Approaches in the measurement stability and security categories may affect the measurement stability of the system. These changes are therefore implemented as a series of experiments, described further in Section 4.5. Approaches trying to improve measurement stability that did not actually give any benefit were reverted, but are listed here and the results are shown in Chapter 5. All other improvements listed here are not reverted.

This leads to the following hypotheses, which is treated individually for each approach in the respective categories:

It is assumed that changes that are outside the “critical section” of the code base do not affect the measurement stability, and no experiments are performed to check the impact of these improvements. The critical section of the code base is the section in which changes might affect measurements, in other words, from clearing of the cache and temperature adjustments to the end of the measurements. Changes related to run-time error feedback and other refactorings and bug fixes are outside the critical section.

4.4.1 Measurement Stability

Five approaches to improve measurement stability have been tried: removing Fail2Ban, managing CPU governors, implementing measurements in C++ and the `taskset` and `nice` utilities. This subsection explains the implementation details of each of these approaches.

4.4.1.1 Removing Fail2Ban

The CMB system is secured from remote login attempts using two mechanisms: Uncomplicated Firewall (UFW) [Ufw] and Fail2Ban [Fai]. UFW can allow or deny specified IP ranges or ports to connect to the subsystem (i.e., either the server or the back end). The back end blocks all connection attempts from outside of the NTNU network using UFW.

Fail2Ban scans log files for IPs that repeatedly fail to connect, and IPs that show signs of malicious intent are banned. The intent of Fail2Ban is to reduce the risk of attackers getting access to the system it is installed on. The Fail2Ban program runs in the background and continuously scans log files, possibly introducing a source of instability to the measurements.

Since the back end already blocks all connection attempts from outside the NTNU network, and the devices on the NTNU network are generally trusted, Fail2Ban can be removed without introducing much risk to the system. It does however serve as an extra layer of protection and can therefore be useful (e.g., if UFW is misconfigured in some way).

Removing Fail2Ban is done by uninstalling the `fail2ban` package using `apt-get`, or, for a fresh install, not installing it at all.

This leads to the following hypothesis:

Hypothesis 1 (Fail2ban) *Removing Fail2ban improves the measurement stability of the CMB system.*

4.4.1.2 CPU Governors

CPU governors are implemented by the Linux Kernel. The governors control the CPU frequency (but not voltage), setting the frequency of each core in response to increase or decline in CPU load. There are multiple different governors, each implementing a different scaling algorithm for controlling the frequency.

The default governor for the installed variant of Ubuntu 18.04 is `ondemand`. The `ondemand` governor estimates the CPU load based on the ratio of idle time between two consecutive invocations of the governor [Wys17]. The `ondemand` governor might cause measurement instabilities due to uncontrollable frequency changes during measurements.

The `performance` governor sets the CPU frequency to the maximum frequency possible. By having a constant frequency the measurements are likely to be more

stable. This governor is used by default when setting up the back end using the installation script.

This leads to the following hypothesis:

Hypothesis 2 (Performance Governor) *Using the performance governor improves the measurement stability of the CMB system.*

4.4.1.3 Implementing Measurements in C++

The CMB system previous to this thesis used the `date` utility to measure the duration of the program. This utility can give timestamps with high precision, but invoking the program might introduce overhead that is uncontrollable and unpredictable. Such overhead might be introduced by `date` having more system calls than strictly necessary, having to parse the custom format string or transforming the output to the correct format. The `date` utility also does not give the caller any control over which clock source is used to measure the time.

In order to try to control, and reduce, the overhead related to timing programs the timing functionality was wrapped in a C++ program, named `run_command`. `run_command` uses the system call `clock_gettime(2)` before and after the program to measure the duration of the program. `CLOCK_MONOTONIC_RAW` is used as a clock source, a clock that always increases and is unaffected by clock adjustments [Clo]. The command to invoke the program is unchanged, but is now invoked using `std::system [Sysb]`; effectively launching a child process.

Energy measurements have previously been performed by running an executable in a background process (described in Section 2.1.5). The executable is compiled from C++ sources and has an unnecessary dependency on QT and `qmake`. By moving this code into `run_command` the measurements are done in the same place, and the unnecessary dependencies can be removed. The energy measurement is wrapped in a class that launches the measurement in a separate thread using `std::async` with the `std::launch::async` flag to ensure that the code is not lazily evaluated [Asy]. Aside from removing the dependencies and wrapping the energy measurement, the original sources for `EnergyMonitor` were kept unchanged to avoid introducing bugs or affect measurements.

This leads to the following hypothesis:

Hypothesis 3 (C++ measurements) *Performing measurements in C++ improves the measurement stability of the CMB system.*

4.4.1.4 `taskset` Command

When running a program on the Odroid-XU3, the Linux kernel scheduler is free to assign the program to any core. This might lead to unpredictable measurements if the scheduler assigns the program under evaluation to a small rather than a big core for a single measurement, or moves the program between cores during execution.

In order to control what core the program under evaluation is run on, the `taskset(1)` command is used to set the CPU affinity of the process, essentially binding the process to a given set of cores [Tas]. The program is bound to the first “big” core, using the affinity mask `0x10`, guaranteeing that it is not migrated to other cores by the scheduler. This does not, however, disallow other programs or processes (e.g., kernel worker processes) from executing on the same core, meaning that the program under evaluation still can be pre-empted. Ideally, all other programs and processes running on the back end should be restricted to a single core to avoid pre-emption of programs under evaluation (discussed in Section 7.4.1.3).

This approach did not work well with OpenMP programs, as it bound them to a single core. Working around this using `sched_setaffinity(2)` [Sch] to override the affinities set by `taskset(1)` allowed the programs to use multiple cores, but with sub-optimal efficiency—frequent thread migration caused a very high system call overhead—compared to running the test program without `taskset(1)`. The reason why this approach causes a high rate of thread migration for OpenMP programs is not known. Due to these issues, measurements were not made for this test program and this improvement is disabled by default. This is discussed further in Section 7.4.1.7.

This leads to the following hypothesis for non-OpenMP programs:

Hypothesis 4 (Taskset) *Setting core affinity using `taskset(1)` improves the measurement stability of the CMB system for programs not using OpenMP.*

4.4.1.5 `nice` Command

The Linux Kernel scheduler takes program priority into account when scheduling programs. Every process has a priority and a nice value. The nice value range from `-20` to `19`: `-20` gives the highest priority and `19` gives the lowest priority. The default nice value is `0` [Neg12].

By setting the nice value using the `nice(1)` command [Nic], the programs under evaluation can be assigned a higher priority. This makes it less likely that other processes interfere with the measurements (i.e., it is less likely that other processes preempts the program under evaluation).

Unprivileged users (e.g., the *worker* and *climber* users) can only alter the nice value of processes they own, and then only increase it to assign a lower priority. Superusers can change the priority of any process to any valid value [Ren]. Running measurements with a superuser is a security risk since this means that the program executed have superuser privileges. Therefore, a way of giving processes higher priorities without introducing additional risks to the back-end security is required.

The Linux kernel implements capabilities [Cap] that can be assigned using the `setcap(8)` command [Seta]. Capabilities divide privileges into units that can be independently enabled for files or threads. The `CAP_SYS_NICE` capability allows a process to set the nice value to any valid value for a process owned by any user. Assigning capabilities must be done by a superuser.

During the installation procedure (which has to be run by a superuser) the capability `CAP_SYS_NICE` is set for the `run_command` executable (described in Section 4.4.1.3). The `run_command` executable uses the `nice(1)` command to set the niceness of the program to be measured to `-15`, giving it a fairly high priority.

This leads to the following hypothesis:

Hypothesis 5 (Nice) *Setting priority using `nice(1)` improves the measurement stability of the CMB system.*

4.4.2 Security

Two new approaches to improving security have been implemented: “chroot jails” and limiting memory usage.

4.4.2.1 Implementing a Chroot Jail

Currently, the program under evaluation is run as the *worker* user and has read access to the most of the file system, execute access to many unneeded programs and utilities and write access to some directories (e.g., the `/tmp` directory). Changing permission bits to exclude the *worker* user might exclude other users that have access to these directories and break arbitrary parts of the system (e.g., many background processes depend on being able to create and write to files in the `/tmp` directory).

An exploit written by Lasse Eggen and Fanny Skirbekk (Listing 4.1) shows that it is possible to gain information about the file system. This exploit can be modified to read files that the worker user has permissions to read. Being able to inspect

```

#include <fstream>
#include <array>
#include <cstdio>
#include <iostream>
#include <memory>
#include <stdexcept>
#include <string>

// execute shell command
std::string exec(const char *cmd) {
    std::array<char, 128> buffer;
    std::string result;
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd, "r"),
                                                pclose);

    if(!pipe)
        throw std::runtime_error("popen() failed!");
    while(fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr)
        result += buffer.data();
    return result;
}

int main() {
    std::cout << exec("ls -lah /dev/") << std::endl;
}

```

Listing 4.1: A C++ program showing how to execute commands and write the output. This output is returned to the user if the correctness test fails. Based on an exploit written by Fanny Skirbekk and Lasse Eggen.

files in the file system might give contestants an unfair advantage, or give away weaknesses in the system to malicious users.

Fortunately, utilities exist to restrict programs to a given directory and only allow access to specified files and directories. The primary utility used is the `chroot(2)` system call. This system call changes the root directory (i.e., `/`) of the process that calls it to a specified directory [Chr]. The program will then use this directory as the root directory when trying to find files; effectively “jailing” the program to the file system reachable from the directory.

The programs that are evaluated are dynamically linked and have to load the libraries when they start. In order to load these libraries the programs require access to `/lib`, `/usr/lib` and similar directories in the file system, which, by

default, are not reachable from the “chroot jail”. OpenCL programs require access to OpenCL-specific directories and `/dev`.

There are multiple ways of giving access to directories outside the chroot jail, but in doing so it must be ensured that: the program cannot modify the files, and if the program can modify the files, modification does not affect the files used by other programs.

One approach to give access to files is to copy all the files needed into the chroot jail directory with the correct paths (e.g., `/lib/file` must be copied into `jail/lib/file`, where `jail` is the chroot jail directory). This approach requires finding out which libraries are required by the evaluated program (e.g., by parsing the output of `ldd(1)`), copying these into the chroot jail directory, setting permissions, and deleting the files afterwards—a rather complicated process.

A less complicated approach was adopted to save some time implementing file access. This approach uses the `mount(8)` command to create a *bind mount* for each of the needed directories. A bind mount mounts a directory in another directory, making the content accessible in two places in the file system. When using a bind mount, a read-only specifier can be used to ensure that the contents is not writeable. The bind mount has to be set up every time the system boots using an entry in `/etc/fstab`. Admittedly, this approach lets the program under evaluation read more files than strictly necessary, but is simpler to implement and does not have any significant security risks compared to the first approach.

When the chroot jail is set up with care, running programs without special permissions as an unprivileged user provides more security than before the beginning of this thesis. `chroot(2)` is not designed to be used for security purposes, and should not be treated as an impenetrable security mechanism, but merely as the first layer of security when evaluating programs.

The chroot jail is implemented by another executable, `run_chroot`, which is called from the `run_command` executable. It is given the path to the jail, input and output files, and the ID of the *worker* user. The executable then modifies the `stdin` and `stdout` file descriptors to use the input and output files respectively, before chrooting into the chroot jail directory, changing the user to the *worker* user (using `setuid(2)` [Setc]) and then using `execl(3)` to start the program to be evaluated. `execl` (and other front ends for `execve(2)`) consumes the process image of the calling process, but keeps nice value, process ID, real user ID, current working directory, root directory, resource limits, file descriptors and so on [Exea; Exeb; Exec].

Since `chroot(2)` and `setuid(2)` requires special privileges, the installation script gives `run_chroot` the capabilities `CAP_SYS_CHROOT` and `CAP_SETUID` (similar to how capabilities were given to `run_command`, as described in Section 4.4.1.5). The

capabilities are deactivated during the call to `exec1(3)`, since the program to be evaluated does not have any capabilities.

Since this approach to improve security does not aim to improve measurement stability, we hypothesize that this approach does not worsen measurement stability:

Hypothesis 6 (Chroot Jail) *Securing evaluation of programs using a chroot jail does **not** negatively affect the measurement stability of the CMB system.*

4.4.2.2 Limiting Memory Usage

According to Forišek, if the OS can be forced to run out of memory, any process can be killed if it tries to request more memory when all memory has already been allocated [For06]. This can have unpredictable results, potentially killing other processes required by the CMB system.

To mitigate such problems, memory-limiting has been implemented. Using Bash's built-in command `ulimit` it is possible to restrict many features of the process including data segment size, stack size, resident set size (in RAM), and more [Bas]. The `ulimit` command is a wrapper around `setrlimit(2)` [Setb]³.

For the CMB system this is implemented by prepending the `ulimit` command to the command starting `run_chroot` (see Section 4.4.2.1). Memory is limited to 1GB by default, but this is configurable. The effect of this change on measurement stability is not measured.

4.4.3 Run-Time Error Feedback

This thesis has made some changes to how errors are handled and generated on the back end. The error handling is moved from the entry point of the back end, the Bash script `runscript_v2.sh`, to a new Python script, `generate_error_messages.py`, which is called from `runscript_v2.sh`. This has multiple benefits: Bash support for JSON is rather poor, and generating correctly formatted JSON in Python is simpler, cleaner and more readable; error message handling, which was previously scattered in the Bash script, are moved into a script with a single purpose, resulting in DRY⁴ code; and finally, this approach simplifies changing and extending the way error messages are handled.

³After finishing the implementation, some problems with `setrlimit(2)`-based approaches for multi-threaded programs have been discovered. More information here: <http://coldattic.info/post/40/>. Alternative approaches are discussed in Section 7.4.2.4.

⁴The DRY principle: "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.", or, Don't Repeat Yourself. [HT99, p. 28]

4.4.4 Refactorings and Bug Fixes on the Back End

In addition to all the other improvements previously listed, other improvements have been made to the back end to increase system maintainability and reliability. This section list some of the more prominent changes.

The back end previously used Python 2.7. The Python code has been ported to Python 3.6 in anticipation of the upcoming EOL of Python 2.7 in 2020 [Pep]. Python 3.6 was chosen since this is the default Python 3 version on Ubuntu 18.04. For environment isolation and easier dependency tracking, Python packages are now installed in *virtual environments* using the `virtualenv` package [Vir].

To help get clean, good code conforming to best practices, linters were installed to perform analysis of the code. For Python the `pylint` package [Pyl] was adopted, and configured using a `.pylintrc` in the root directory of the `cmb-board` repository. For Bash scripts a `shellcheck` [She] was adopted. Most linter errors have been fixed.

The entry point now supports arguments that can configure the behavior when called from tests or from the server. The command line options supported are outlined in Table 4.1. If no options are specified, the script performs a measurement using default settings—maintaining API compatibility.

<code>--debug</code>	Enables debugging mode with more log messages.
<code>--memory-limit arg</code>	Sets the memory limit for a program (in kilo bytes). Default value: 1 000 000
<code>--no-energy</code>	Disables energy monitoring.
<code>--no-security</code>	Disables some of the security measures.
<code>--no-cleanup</code>	Does not remove program source files from the workspace. Useful for measurement stability tests and debugging when a program is run repeatedly.
<code>--timeout-length arg</code>	Sets the maximum amount of wall clock time a program can use during measurement. Default value: 90

Table 4.1: The configuration options supported by the entry point `runscript_v2.sh`

4.5 Measurement Stability Experiments

A measurement framework had to be implemented to quantify the effect the different approaches had on measurement stability. The measurements made for a program must be unchanged over time to be “fair” to the users; keeping the measurements stable over time is of a higher priority than using less time or energy.

A suitable statistic that quantifies measurement instabilities had to be chosen according to these priorities.

4.5.1 Measurement Stability Statistic

The *coefficient of variation*, C_v , is the chosen statistic for quantifying measurement instabilities. C_v is defined in Eq. (4.1c) as the ratio between the sample standard deviation (Eq. (4.1b)), s , and the sample mean (Eq. (4.1a)), \bar{x} .

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad (4.1a)$$

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2} \quad (4.1b)$$

$$C_v = \frac{s}{\bar{x}} \quad (4.1c)$$

N is the number of samples and x_i is sample i .

The standard deviation might have been chosen as a statistic. However, the standard deviation is unsuitable for direct comparison between two data sets with different means, because the interpretation is dependent on the mean (e.g., a standard deviation of 0.1 might be large when the mean is 1, but small when the mean is 100). The coefficient of variation, however, is independent of the mean and the unit of measurement, and can therefore be used for comparison when the means are different. It should be noted that the coefficient of variation is rather sensitive when the mean is close to zero; small variations in standard deviation have a large impact.

4.5.2 Measurement Stability Experimental Setup

For every test program described in Section 4.5.3 100 iterations of measurements are performed. The programs run using the same entry point as the server when evaluating programs on the back end (i.e., for every measurement sample, it recompiles the program, runs the small correctness test, clears the cache, adjusts the temperature and then runs the measurement test). The tests run contiguously, controlled by a script collecting and storing the measurement data.

The measurement experiment scripts are committed to the *cmb-board* repository. The experiments are controlled by two scripts: `run_measurement_tests.sh` and `measure.sh`, where the latter receives input arguments and runs a given number of iterations of a given test program and writes the measurement results to a given file, and the former script is a wrapper around the `measure.sh` script and runs this script for every test program. All tests are run while having an open SSH connection to the board, similar to the connection from the server to the board when evaluating user-submitted programs.

The experiment script `measure.sh` was originally written as a Python script, `measure.py`, but since the Python runtime might affect the measurements to a larger degree than Bash, the script was rewritten and all the experiments performed up to that point were performed once more. The original script used the JSON format, but this is not the case for the Bash implementation. Therefore a script was created, `clean`, to transform the data into JSON format.

The scripts `plot` and `calc_statistics` can be used to plot the measurements and calculate the relevant statistics shown in Chapter 5, respectively. These scripts expect the data in a JSON format.

After upgrading the back end to Ubuntu 18.04, a set of measurements were performed to create a *baseline* against which the experiments were compared. Upgrading the OS is a change that might have a large effect on measurement stability. As previously stated in Section 4.1, a great amount of effort has been put into not making more changes to the system than absolutely necessary when upgrading, in order to have a baseline that is comparable to the old system.

In order to decrease the number of necessary experiments and speed up development time, it is assumed that refactoring that does not change the implementation within the critical section of the scripts, also does not affect the measurement stability.

4.5.3 Measurement Stability Test Programs

The measurements are performed using five different test programs: Hello World, Mandelbrot (single-threaded), Mandelbrot (using OpenMP), Mandelbrot (using OpenCL), and Sort. These programs are chosen due to their different properties. The source code for all the programs are given in Appendix C.

Hello World is a simple, short program only outputting the string “Hello World!”. This program is mainly included to observe how very short programs behave in the experiments performed, but is not representative of typical problems given to the users.

Mandelbrot calculates the Mandelbrot set in the range $x \in [-2, 0.5]$, $y \in [-1, 1]$ for a maximum of 255 iterations. The input parameters (width and height) decide the resolution, in other words, how many discrete points in the 2D plane that the Mandelbrot set should be calculated for. In order to avoid that the program becomes I/O-bound, the program outputs the number of points that are inside the Mandelbrot set only.

The OpenMP and OpenCL versions of the Mandelbrot program calculates works similarly to the single-threaded version, but have larger input parameters to ensure that the programs do not run for a too short amount of time. The OpenCL version counts the number of points in the Mandelbrot set in a single-threaded manner (after generating the Mandelbrot set on the GPU in OpenCL). These programs can be used to examine how the experiments affect OpenMP and OpenCL programs.

Sort is a simple program that reads a large amount of numbers, sorts them using `std::sort` and outputs the sorted numbers. This program is used to examine how I/O-bound programs behave in our experiments.

In a single experiment a version of the Sort program that uses the implementation of `RandInt` in Listing 3.6 is also used. The goal is to examine the difference between programs reading numbers from standard input and programs generating numbers. Note that the `RandInt` version does not use the same random sequence of numbers, but uses the same amount of numbers in the same range. This leads to an additional hypothesis:

Hypothesis 7 (I/O impact) *I/O-bound programs have worse measurement stability than non-I/O-bound programs.*

4.6 Refactorings and Bug Fixes on the Server

Although the main focus of this thesis has been on the back end, some changes have been made on the server in addition to upgrading the OS to Ubuntu 18.04. Some of these changes have been listed in this section.

A `systemd` service to automatically start the Flask server after reboot has been created. This has the additional benefit of reducing restarts of the Flask server to simple `systemctl` commands if a restart (but not a reboot) is required.

Python packages have been upgraded to newer versions to get the latest security patches. Necessary updates to the code to fix breaking changes in the packages have been made.

An API performance issue has also been fixed. When requesting data from the `/api/problems` API endpoint, the full list of problems was returned. For each problem in the list of problems, the `id` and `name` fields was returned, in addition to extra, unnecessary data (including submission details for every submission made to a given problem). The extra data was never used because it was requested from other API endpoints when needed, but was requested from the database, serialized and transported to the client, which resulted in a huge delay.

By modifying the SQL Alchemy query for the `/api/problems` endpoint to only request the `id` and `name` column from the database, response time for this API endpoint was reduced from around 16 s to around 0.5 s. The size of the transferred data was reduced from 1.67 MB to 1.99 KB. These measurements were repeated three times on a laptop using the Mozilla Firefox browser, with the browser cache disabled.

Results and Discussion

This chapter presents and discusses the measurement results from the experiments performed as part of this thesis. Section 5.1 presents prior measurements performed on the CMB system to act as a baseline. The measurement results from the experiments performed as part of this thesis are given in Section 5.2, along with a discussion of each hypothesis. Section 5.3 discusses errors and threats to the validity of the results presented.

5.1 Prior Experiments Using the CMB System

This section presents results from prior measurements performed on the CMB system to act as a baseline. For each set of results, the experimental test setup used is outlined, along with comments on how the results should be interpreted.

Table 5.1: C_v (lower is better) from experiments performed by Støa and Follan [SF15].

Experiment	Time	Energy
Shortest-Path	0.15%	N/A
Blackscholes-serial	0.25%	3.8%
Blackscholes-OpenMP	0.51%	0.6%
Freqmine-serial	0.25%	1.4%
Freqmine-OpenMP	7%	11%

The results from experiments performed by Støa and Follan [SF15] are shown in Table 5.1. Their experiments used different programs than this thesis: *Shortest-Path*, *Blackscholes* and *Freqmine*. The *Shortest-Path* experiment is based on the *Shortest Path* problem available to users of the CMB system.¹ The *Blackscholes* and *Freqmine* benchmarks are part of the PARSEC benchmark suite [Bie11], and both serial and OpenMP versions were used. *Shortest-Path* and *Blackscholes* are computation-bound, while *Freqmine* is memory-bound. *Freqmine-OpenMP* is not able to use caches efficiently, leading to poor concurrent behavior [SF15].

Støa and Follan used a different experimental test setup than the experiments performed in this thesis: a `cron` [Cro] job runs the programs every 15 minutes over the course of several days. The *Shortest-Path* measurements were repeated 500 times, while the number of repetitions for the benchmarks was not stated by the authors.

Table 5.2: C_v (lower is better) from experiments performed by Ingebrigtsen [Ing17].

Experiment	Time
Shortest-Path	0.7%
Shortest-Path w/o outliers	0.15%

Ingebrigtsen [Ing17] tries to reproduce the *Shortest-Path* experiment performed by Støa and Follan [SF15]. The results are shown in Table 5.2. Ingebrigtsen removed outliers that occurred during an unexplained spike that lasted 12 hours (see original thesis for details). With outliers removed the results are equal to those presented by Støa and Follan, with $C_v = 0.15\%$. These results show that the measurement stability of the CMB system has not significantly changed from system version one (by Støa and Follan) and system version three (by Ingebrigtsen), allowing comparison of the results from Støa and Follan to the results presented in this thesis in Section 5.2.

Ingebrigtsen used a test setup similar to the original experiment by Støa and Follan. The experiment was performed using the same platform, program and test input as the original experiments. The number of repetitions is not explicitly given, but can be estimated to be more than 500 based on figures presented.²

The CMB project coordinator performed measurements on the *Sigma Unique*³ problem, which was used for the CMB Challenge 2018 competition. The measurement results are shown in Table 5.3. Programs solving the Sigma Unique problem have to read up to 65 000 numbers and perform relatively simple operations to

¹<https://climb.idi.ntnu.no/#/problem/1>

²The figures show that the measurements were repeated over more than six days. Subtracting the 12 hour period of outliers gives 5.5 days with measurements every 15 minutes. $5.5 \times 24 \times 60/15 = 536 > 500$

³<https://climb.idi.ntnu.no/#/problem/63>

Table 5.3: Sample mean, sample standard deviation and C_v (lower is better) from measurements performed by the CMB project coordinator in April, 2018.

Experiment	Time			Energy		
	mean s	stddev s	C_v %	mean J	stddev J	C_v %
Sigma_no	32.36	0.08	0.24	79.46	0.71	0.89
Sigma_s	2.34	0.01	0.58	8.75	0.65	7.41
Sigma_su	2.00	0.03	1.32	7.99	0.57	7.20

produce a result, making it a I/O-bound problem. The *Sigma_no* experiment uses a minimal solution without optimizations. The *Sigma_s* optimizes the solution by using `ios::sync_with_stdio()` to speed up I/O-operations, while the *Sigma_su* also uses more specialized data types for the problem (`uint16_t` instead of `int`). Every experiment repeated the measurements 10 times, by running the programs from the CMB website. These results give insight into the behavior of I/O-bound programs running on the CMB system prior to the upgrade.

5.2 Experimental Results

This section presents the results received from the experiments performed as part of this thesis. First, the main results will be presented, followed by one subsection for each hypothesis. The experimental test setup and the test programs were previously described in Section 4.5. The results relevant for each hypothesis will be discussed along with other relevant observations and each hypothesis will be accepted or rejected. A summary of the conclusions to the hypotheses is given in Section 5.2.2.8. Additional data gathered in the experiments that are not presented in this section are summarized in Tables D.1 to D.4 in Appendix D.

5.2.1 Baseline Comparison

The *Baseline* experiment was performed right after the upgrade to Ubuntu 18.04. The results are shown in Fig. 5.1. By comparing the results in the baseline to the prior measurements in Section 5.1 it is possible to examine changes in measurement stability introduced by the OS upgrade.

The *Mandelbrot* programs are computation-bound, and are therefore comparable to the *Shortest-Path* and *Blackscholes* experiments from Støa and Follan [SF15] presented in Table 5.1. The time C_v from the *Baseline* experiment for *Mandel-*

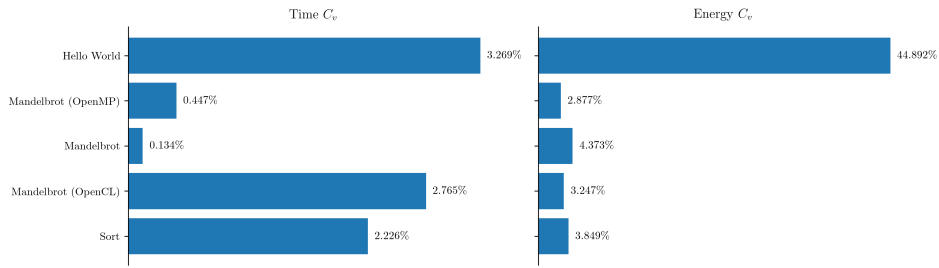


Figure 5.1: *Baseline* experiment results, showing the C_v (lower is better) for the test programs.

brod is slightly lower than the time C_v for both *Shortest-Path* and *Blackscholes-serial*, while the energy C_v for *Mandelbrot* is slightly higher than the energy C_v for *Blackscholes-serial*. When comparing the OpenMP version of the *Mandelbrot* program to *Blackscholes-OpenMP*, the time C_v is slightly lower, but the energy C_v is much higher (2.3 percentage points). It is apparent that the results from the *Baseline* experiment are similar (but with minor variations) to the results achieved by Støa and Follan for computation-bound programs—with the exception of the energy C_v for the OpenMP programs, where the source of the difference between the results is unknown.

The Sort program is I/O-bound and comparable to the measurements performed by the CMB project coordinator for the *Sigma Unique* problem. The Sort program will be compared to the *Sigma_s* because both programs use the same optimization.⁴ The Sort program has a much higher time C_v (1.6 percentage points) than *Sigma_s*, but a much lower energy C_v (3.6 percentage points). A limitation of this comparison is the large difference of I/O: the Sort program reads 40 000 000 numbers, while the *Sigma Unique* problem only has 65 000.

Since the I/O might have a large impact on the C_v for I/O-bound programs, it is excluded in the discussion and evaluation of the hypotheses, except hypothesis 7 which concerns the effect of I/O on programs. The impact of I/O on programs and hypothesis 7 is treated further in Section 5.2.2.7.

The Hello World program’s execution time is close to zero (see Table D.1), meaning small variations in standard deviation or mean have a large impact on the C_v . This is an inherent flaw of the chosen statistic. The Hello World program is—due to this flaw—excluded from the discussion and evaluation of hypotheses.

As a result of excluding the Hello World and Sort program, only the three versions of the Mandelbrot program—*Mandelbrot*, *Mandelbrot (OpenMP)* and *Mandelbrot (OpenCL)*—will be considered when discussing and evaluating hypotheses 1 to 6.

⁴Using `ios::sync_with_stdio(false)`.

5.2.2 Discussion and Evaluation of Hypotheses

The approaches described in Section 4.4 have been combined in the experiments shown in Fig. 5.2. *Baseline* corresponds to the *Baseline* experiment also presented in Fig. 5.1, *No Fail2Ban* corresponds to removing `fail2ban` (Section 4.4.1.1), *Performance governor* corresponds to using the `performance` governor (Section 4.4.1.2), *C++* corresponds to implementing measurement functionality in C++ (Section 4.4.1.3), *chroot* corresponds to using a chroot jail (Section 4.4.2.1), *taskset* corresponds to setting core affinity (Section 4.4.1.4), and *nice* corresponds to changing process priority using `nice` (Section 4.4.1.5).

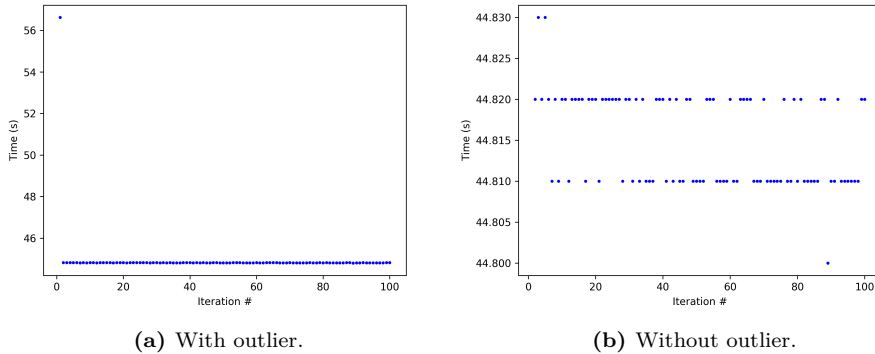
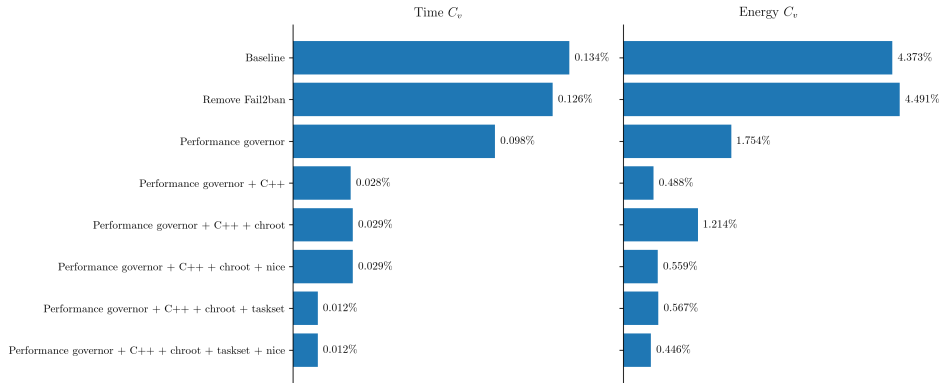


Figure 5.3: The Mandelbrot program in the *Perf + C++ + chroot + taskset + nice* experiment.

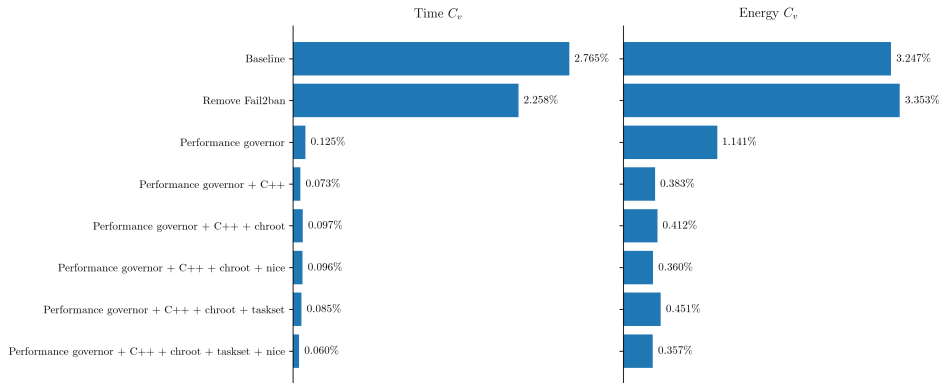
Fig. 5.3 shows that that the *Mandelbrot* program in the *Perf + C++ + chroot + taskset + nice* experiment have a single outlier. By removing the outlier more accurate results can be calculated. Fig. 5.2a shows the calculated results with the outlier removed.⁵

The rest of this subsection will consider each of the hypotheses in turn, along with the relevant results. When considering each hypothesis (except hypotheses 4 and 7) the time and energy C_v for each of the three versions will be taken into account. Hypothesis 4 considers *taskset*, for which no measurements have been performed with *Mandelbrot (OpenMP)*. Hypothesis 7 compares I/O-bound to non-I/O-bound programs, therefore both versions of the *Sort* program will be considered.

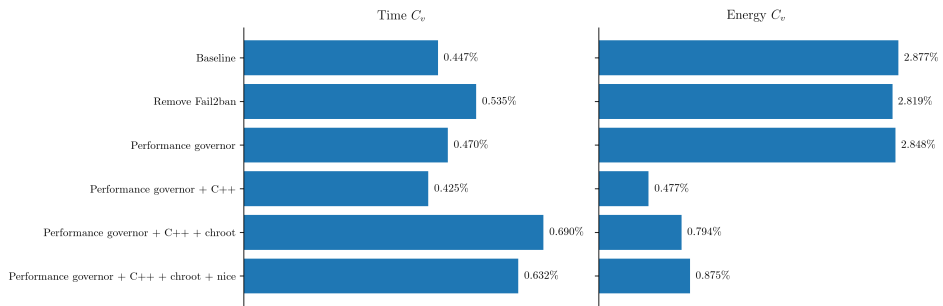
⁵Calculation using the outlier gives $C_v = 2.614$ for time and $C_v = 3.132$ for energy.



(a) Mandelbrot



(b) Mandelbrot (OpenCL)



(c) Mandelbrot (OpenMP)

Figure 5.2: The C_v results for all versions of the Mandelbrot program.

5.2.2.1 Removing Fail2Ban

Hypothesis 1 (Fail2ban) *Removing Fail2ban improves the measurement stability of the CMB system.*

The *Remove Fail2Ban* experiment tested the impact of the `fail2ban` program on the measurement stability (described in Section 4.4.1.1). When comparing the *Remove Fail2Ban* experiment to the *Baseline* experiment (Fig. 5.2) it is clear that the results are ambiguous; for each program, the C_v for either time or energy improves while the other C_v worsens, thus hypothesis 1 should be rejected.

The rejection of the hypothesis does not entail that the `fail2ban` program does not affect the measurements—Støa and Follan [SF15] showed that programs running in the background can affect measurements (e.g., *LightDM*)—but that `fail2ban` does not affect the measurements enough to be distinguishable from other (unknown) factors in the experiments performed. Since `fail2ban` has some security benefits, the removal of `fail2ban` was reverted and subsequent experiments have been performed with `fail2ban` running in the background.

5.2.2.2 Performance Governor

Hypothesis 2 (Performance Governor) *Using the performance governor improves the measurement stability of the CMB system.*

The *Performance governor* experiment tests the impact of using the `performance` governor (described in Section 4.4.1.2). When comparing the *Performance governor* experiment to the *Baseline* experiment, this has a significant impact on the *Mandelbrot* and *Mandelbrot (OpenMP)* programs, which run on the CPU. *Mandelbrot (OpenCL)* sees a slight improvement in energy C_v , while the time C_v slightly worsens.

Even if the time C_v for *Mandelbrot (OpenCL)* program slightly worsens, the results for *Mandelbrot* and *Mandelbrot (OpenMP)* strongly suggests that hypothesis 2 should be accepted.

The increase from 84 J to 114 J in mean energy consumption of *Mandelbrot (OpenCL)* is noteworthy (Table D.4) from the *Baseline* experiment to the *Performance governor* experiment. Since the OpenCL program performs the main chunk of computation on the GPU, the `ondemand` governor likely reduced the frequency of the CPU cores to save power during the *Baseline* experiment. When using the `performance` governor the cores are at max frequency and will as a result use more power, even if the cores are not utilized. However, the measurements do not collect any data

that might prove or disprove that the `ondemand` governor reduced the frequency of the cores.

5.2.2.3 Measurements in C++

Hypothesis 3 (C++ measurements) *Performing measurements in C++ improves the measurement stability of the CMB system.*

The *Performance governor + C++* experiment tests performing measurements in C++ as described in Section 4.4.1.3. The measurement stability improves for all programs when comparing the results in Fig. 5.2 for the *Performance governor + C++* and *Performance governor* experiments, which strongly suggests that hypothesis 3 should be accepted.

Moving energy measurements from a process to a more lightweight thread, as well as using system calls over command line utilities to improve measurements improves the mean running time and energy consumption of all versions of the *Mandelbrot* program, with one exception: the energy consumption of *Mandelbrot (OpenMP)* doubles. This is a surprising result, and the cause is not known.

Moving measurements to C++ also mitigates another issue with the energy measurements that can be observed in Fig. 5.4: Fig. 5.4a shows that the measurements divide into two groups where one group has a relatively higher energy consumption than the other—this grouping was also present in the *Baseline* and *Remove Fail2Ban* experiments. This grouping disappears in the *Perf + C++* experiment (Fig. 5.4b) and subsequent experiments.

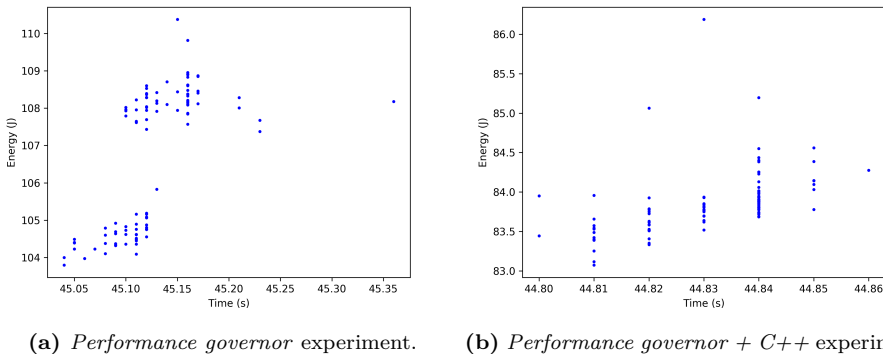


Figure 5.4: Time-energy plots for the Mandelbrot program showing that the grouping of measurements disappears when implementing measurements in C++.

5.2.2.4 Chroot Jail

Hypothesis 6 (Chroot Jail) *Securing evaluation of programs using a chroot jail does **not** negatively affect the measurement stability of the CMB system.*

The *Performance governor + C++ + chroot* experiments tests the impact of securing measurements using a chroot jail. When comparing the *Performance governor + C++ + chroot* to the *Performance governor + C++* experiment, the evidence strongly suggests that hypothesis 6 should be rejected as the C_v for all the versions of the *Mandelbrot* program worsens.

In our opinion, the security benefits of the chroot jail outweighs the drawbacks of less stability in measurements. The chroot jail is therefore enabled by default, and used in subsequent experiments.

After completing the experiments, an improvement to the current implementation of chroot jails was discovered that could give more precise time measurements and mitigate—if not completely remove—the negative impact on measurement stability. This improvement was discovered too late in the thesis work, and is therefore outlined in Section 7.4.1.1.

5.2.2.5 Nice

Hypothesis 5 (Nice) *Setting priority using `nice(1)` improves the measurement stability of the CMB system.*

Giving higher priority to the program under evaluation using `nice`, was tested in two separate experiments: *Performance governor + C++ + chroot + nice* and *Performance governor + C++ + chroot + taskset + nice*.

When comparing the *Performance governor + C++ + chroot + nice* experiment to the *Performance governor + C++ + chroot* experiment in Fig. 5.2 only the energy C_v for *Mandelbrot (OpenMP)* worsens. The energy C_v for *Mandelbrot* improves significantly, while the C_v is equal or improves slightly for the other measurements.

When comparing the *Performance governor + C++ + chroot + taskset + nice* experiment to the *Performance governor + C++ + chroot + taskset* experiment in Fig. 5.2, the C_v for both time and energy improve for the *Mandelbrot* and *Mandelbrot (OpenCL)* programs. Measurements involving `taskset(1)` were not performed for *Mandelbrot (OpenMP)*, as explained in Section 4.4.1.4.

The combined evidence from both experiments using `nice(1)` suggests that hypothesis 5 should be accepted.

5.2.2.6 Taskset

Hypothesis 4 (Taskset) *Setting core affinity using `taskset(1)` improves the measurement stability of the CMB system for programs not using OpenMP.*

The `taskset(1)` program is used to bind the program under evaluation to a given core in two experiments: *Performance governor + C++ + chroot + taskset* and *Performance governor + C++ + chroot + taskset + nice*.

When comparing the *Performance governor + C++ + chroot + taskset* experiment to *Performance governor + C++ + chroot* in Fig. 5.2, the time C_v improves for both *Mandelbrot* and *Mandelbrot (OpenMP)*. The energy C_v improves for *Mandelbrot*, but worsens for *Mandelbrot (OpenMP)*.

When comparing the *Performance governor + C++ + chroot + taskset + nice* to the *Performance governor + C++ + chroot + nice* experiment in Fig. 5.2, the C_v is either equal to or improves for both time and energy for *Mandelbrot* and *Mandelbrot (OpenMP)*.

The combined evidence from both experiments using `taskset(1)` suggests that hypothesis 4 should be accepted. However, due to the problems with `taskset(1)` with OpenMP programs, `taskset(1)` is not used by default.

5.2.2.7 The Effect of I/O on Measurement Stability

Hypothesis 7 (I/O impact) *I/O-bound programs have worse measurement stability than non-I/O-bound programs.*

From Fig. 5.5 it is clear that the computationally bound *Sort w/RandInt* program is significantly more stable than the I/O-bound *Sort* program. This suggest that hypothesis 7 should be accepted.

Comparing the *Sigma_su* program from the CMB Challenge 2018 (in Table 5.3) to the *Sort* and *Man* from the *Baseline* experiment (in Fig. 5.1) suggest that the amount of I/O is correlated to the degree of measurement instability for I/O-bound programs. No conclusions can be drawn here, but the impact of I/O on measurement stability and possible mitigations to reduce the impact of I/O should be studied further.

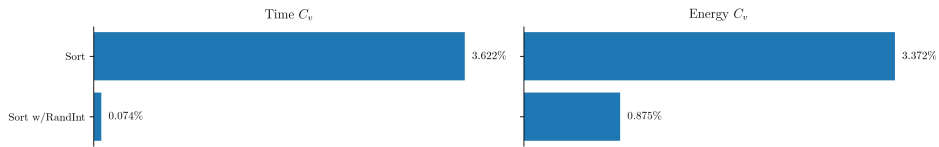


Figure 5.5: Comparison of the I/O-bound program *Sort* to the computation-bound program *Sort w/RandInt* in the *Performance governor + C++ + chroot* experiment.

The measurement stability issues related to I/O-heavy problems should affect the design of future problems for the CMB system. All students answering the questionnaire after CMB Challenge 2019 (Section 3.4) preferred generating random numbers over reading numbers, which suggests that random number generation should be the default method for problems requiring large amount of input.

5.2.2.8 Summary

For all hypotheses the relevant results have been presented, and the hypotheses have been discussed and concluded. A summary of the hypothesis conclusions is shown in Table 5.4.

Table 5.4: A summary of the conclusion to the hypotheses.

Hypothesis	Conclusion
Hypothesis 1 (Removing Fail2Ban)	Rejected
Hypothesis 2 (Performance governor)	Accepted
Hypothesis 3 (C++ measurements)	Accepted
Hypothesis 4 (Taskset)	Accepted
Hypothesis 5 (Nice)	Accepted
Hypothesis 6 (Chroot jail)	Rejected
Hypothesis 7 (I/O impact)	Accepted

5.3 Errors and Threats to Validity

The Hello World and Sort programs was removed from the evaluation of the hypotheses. The C_v is flawed when the mean is close to zero, which is the cause for the Hello World program: the mean time and energy consumption is close to zero, causing small variations to have a large impact on the C_v . For the Sort program, we've seen that I/O has a significant impact on the measurement stability. We argue that exclusion of these programs from the analysis of hypotheses 1 to 6

strengthens the *internal validity*, since we eliminate factors that causes uncontrolled variations in the C_v .

Errors in the experimental setup might also have affected the internal validity of these results. In Fig. 5.6 we observe that the energy in the *Perf* and *Perf + C++* experiments decreased in the first couple of measurements. The Mandelbrot program was the first program to be run, so it might have been affected by the tasks performed before starting measurements—usually running the test suite, or rebooting. If the starting conditions had been set up properly the previous task should not have had any effect on the measurements.

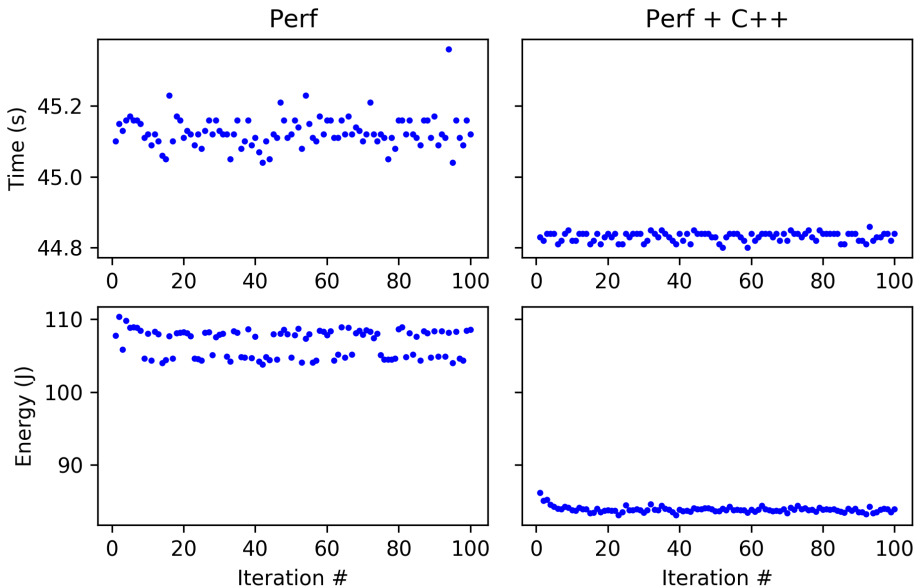


Figure 5.6: Plot of the time and energy measurements for the Mandelbrot program in the *Performance governor* and *Performance governor + C++* experiments that shows that measurements might be affected by tasks performed prior to the beginning measurements.

The current method of adjusting temperatures is imprecise, and testing has shown that a temperature difference of 8°C or more between two or more cores occurs frequently. Experiments performed by Cebrián and Natvig suggests that a temperature increase of 8–10°C increases energy consumption on second generation Intel Core processors by 5% on average [CN13]. The Odroid-XU3 has ARM cores that likely exhibit similar behavior. The temperature adjustment is therefore likely a source of accuracy loss—although it’s likely better than running on “cold” cores—and could be improved.

Implementation details might affect the actual measurements (e.g., invoking `nice(1)` command instead of the `nice(2)` system call). These differences were not considered, but should be subject to further experimentation.

Evaluation and Conclusion

6.1 Evaluation

This section discusses whether the objectives of the problem statement have been fulfilled.

Main Objectives

1. Improve feedback given to CMB users about typical compilation and runtime errors.

Partially implemented by this thesis. Due to the extra unknown complexity of this and other objectives, this objective was deprioritized. Changes to improve run-time errors are described in Section 4.4.3. Testing for correct error messages is covered by Section 4.2. Compilation errors are not implemented, but suggestions to implementing this is described in Section 7.3.1.

2. Implement mitigations for security challenges related to evaluating user code and giving feedback, and (optionally) any other part of the system.

Considered covered by Sections 4.1, 4.2 and 4.4.2. The security features focus mainly on the program evaluation step, but are implemented to be extensible to compilation and profiling steps in the future.

3. Improve the stability of measurements, especially focusing on energy measurements, but also time measurements.

Considered covered by Sections 4.4.1 and 4.5. The improvements were implemented as a series of experiments, and the hypotheses were discussed and concluded in Chapter 5.

4. Continuously refactor and improve the quality of the code base during the thesis work, including implementing more tests and improving logging, in order to increase system maintainability and reliability.

This is considered covered by Chapter 4. Implementation choices have been made to benefit the CMB projects in the long term. The back end now also provides logs that can be read and stored by the server.

Secondary Objectives

A. Conduct a user experiment to evaluate how improved feedback affects usability.

Not covered by this thesis, since improving feedback (objective 1) was de-prioritized.

B. Propose improvements to the testing of submitted programs in order to give better feedback to the user.

Considered covered by Section 7.4.2.1.

C. Propose solutions for implementing support for multiple languages and compilers.

Considered covered by Section 7.4.3.1.

D. Propose solutions for handling multiple XU3-boards and different execution platforms (back ends).

Considered covered by Section 7.4.3.2.

E. Suggest general improvements and bug-fixes to improve any other aspect of the CMB project.

Considered covered by Chapter 7.

F. Develop a command line client for automated uploads to the CMB system from the user's command line (instead of using the website).

Not considered covered by this thesis. Suggestions on how to implement this are given in Section 7.2.4.

G. Implement some of the proposed solutions after approval by CMB project's coordinator.

Considered covered by Sections 4.4.4 and 4.6.

6.2 Conclusion

The purpose of this thesis was to improve and add features to the CMB system, mainly related to measurement stability and program evaluation security, but progress have also been made on CMB system maintainability and user feedback. The OS used by the CMB system has been upgraded to a newer version.

The measurement stability has been improved after the OS upgrade. The coefficient of variation has improved from 0.13% to 0.012% for time and from 4.4% to 0.45% for energy for computation-bound, single-threaded programs. The energy measurement stability for computation-bound OpenMP programs has been improved significantly, at the cost of a slight decrease in time measurement stability. Progress has been made in understanding the behavior of multi-threaded programs running on the CMB system. The measurement stability of OpenCL programs has been improved by an order of magnitude for both time and energy measurements.

The program evaluation security has been improved. The primary improvement restricts the access of the program under evaluation to only library files, and makes execution of shell programs impossible. Progress on limiting resource usage of programs under evaluation has been made, and further improvements in this area have been suggested.

The maintainability and reliability of the CMB system has been improved. The back end has an installation script that ensures that the back ends are set up in the exactly same way every time, and implementation of tests on the back end has uncovered several bugs that have been fixed, and made future changes simpler.

Improvements have been made on feedback given to users through changes in how feedback messages are generated, resulting in more consistent formatting and more verbose error messages.

In conclusion, this thesis has contributed toward improving measurement stability, program evaluation security and maintainability of the CMB system. Moreover this thesis has contributed feature proposals, possible improvements and implementation details for future development of the CMB system.

Future Work

7.1 Project Management and Development Process

7.1.1 Change Repository Hosting

BitBucket is currently used as host for the repositories used in the CMB project. However, another competitor, GitHub, have features that could be useful for the CMB project. Integrated project management tools, wikis, team discussions and release management are notable features that are useful for the CMB project. This makes it easier to create documentation and track issues and ideas in a single place. Due to the nature of the CMB project, where all development is done during a master's thesis, a central place for documentation is especially important. Release management can be used to create and store builds of open source software packages that are specially built for the Odroid-XU3 in order to speed up the installation process (see also Section 4.3).

7.1.2 Continuous Integration

The CMB system previously used Jenkins [Jen] as a continuous integration (CI) tool, as described by Magnussen [Mag16] and Støa and Follan [SF15]. However, at the beginning of this project the CI tool did not work as previously described. Some efforts were made to revive the CI tool, but no progress was made. Reconfiguring

and setting up Jenkins from scratch was not prioritized during this thesis. A CI tool simplifies automatic testing (and possibly also automatic deployment) and an effort to setup a new CI tool should therefore be prioritized.

7.2 Front end

7.2.1 Upgrade or Rewrite Web Application

The application is written in AngularJS 1.4. The Angular 2 (and later versions) are based on entirely different abstractions, making upgrading a hard process. Completely rewriting the front end in another framework might be a more feasible approach, depending on the knowledge of the developer and how much the site will change.

7.2.2 Improve HowTo

The CMB website has a HowTo on the front end. This page should be updated with newer developments of the CMB system, possibly adopting a more “wiki-like” structure. It should also have more relevant information to get users started with the CMB system. For instance, examples on how to use OpenMP or OpenCL, and more technical information about compilers and how programs are compiled on the back end. More technical details are also relevant for researchers using the system.

7.2.3 Contest Features

The current group functionality is used for running contests on the system, and has also been used previously for managing programming classes. However, some central features for contest management are missing from the group functionality, such as score boards, countdown timers and competition rules. Splitting the functionality into separate entities customized for their intended use case would be better than a general, “one-size-fits-all” solution.

7.2.4 Command Line Client

A command line interface (CLI) would benefit power users and researchers using the CMB system. The CLI would use the server API directly, instead of the website,

allowing users to upload submissions and download results. The CLI could be characterized as a second front end for the CMB system.

To make a CLI maintainable the server API should be stabilized and versioned. Since the CLI script is stored at the users' computers is not necessarily updated frequently, changes in the API can break the CLI. Making sure the API is always backwards-compatible will ensure that the CLI always works, even if it does not have access to the newest versions.

7.3 Server

7.3.1 Improving Compilation Error Messages

Many users of the CMB system use Windows or macOS. These OSs may use different compilers than the one used by the CMB system: Windows users often use the MSVC compiler, while macOS users using Xcode use the LLVM Clang compiler. Compilers often implement features that do not strictly adhere to the C++ standard. When users upload code using such features, the compilation will fail on the CMB system, even if it compiles correctly on their local computers. Implementing error messages designed to identify these compiler-specific errors would be helpful to users.

Parsing error messages could be difficult depending on the features of the compiler. The GCC 9.1 compiler, released in May, 2019, can present error messages in a JSON format, which makes parsing easier.¹

Compilation error messages could be used by users to gain access to restricted files. Therefore, properly securing the compilation step and filtering of compilation messages are important.

7.3.2 Upgrade to Python 3

The server is implemented in Python 2, and should be upgraded to Python 3, since Python 2 reaches EOL in 2020. This would also give access to more modern features and abstractions in the language.

¹<https://gcc.gnu.org/onlinedocs/gcc/Diagnostic-Message-Formatting-Options.html#index-fdiagnostics-format>

7.3.3 Logging

Improving logging would make it easier to track down errors. A systematic logging scheme would also make searching and navigating logs easier. Errors and system crashes should be handled specially, by storing details and possibly notifying system developers about errors.

7.3.4 Securing Compilation

The compilation on the server is not secured and susceptible to attacks. DoS attacks against the server-side compilation step would cause harm to all users requesting data from the server. A possible fix would be moving the compilation step to the back end, and secure it properly there, as described in Section 7.4.2.3.

7.4 Back End

7.4.1 Improving Measurement Stability

7.4.1.1 Reducing the Size of the Critical Section

By moving the start of the timing into the `run_chroot` executable, the critical section would be reduced, and the timings would be more accurate. This implementation detail was an oversight during the development work performed as part of this thesis, as discussed in Section 5.3.

7.4.1.2 Collect and Store Additional Measurement Data

Collecting and storing additional data when running measurements would give more insights into what affects the measurement stability. Such data could include CPU core temperatures, RAM and cache usage, running programs, CPU frequency, and power usage measured by each power monitor over time. Data that is collected when a program runs should be enabled as an “analysis mode” (e.g., enabled by a flag or as an extension to profiling), so that it does not interfere with measurements.

7.4.1.3 Limiting Impact of Other Programs

In order to limit the effect of other programs running on the back end (e.g., kernel workers), all programs could be limited to only run on a single, small core using `cgroups(7)`. These programs would possibly interfere less with the programs involved in the measurement (e.g., no pre-emption). Also, temporarily disabling programs or services (e.g., `unattended-upgrades`) not needed during measurements is another way of limiting impact of other programs.

7.4.1.4 Improve Temperature Control

Temperature differences up to 8°C or more have been observed. The temperature can have a significant impact on energy consumption [CN13], and should be controlled more carefully.

The speed of the fan on the Odroid-XU3 can be controlled manually. The fan should be set to a constant speed during measurement to remove possible effects on measurement stability.

7.4.1.5 Limit Effects of I/O Operations

Performing I/O operations to and from RAM instead of disk could improve the speed and reduce measurement stability. To implement this `tmpfs(5)`, or other lower-level tools such as `memfd_create(2)` could be used.

7.4.1.6 Improve Energy Estimation

Energy estimation is done by sampling certain registers at a frequency of 100 Hz. However, the registers are updated at an unknown frequency that is less than 100 Hz, resulting in several identical entries in a row when measuring energy. Looking more closely at the inner workings of the INA231 sensor [Tii] can give insights about energy measurements to improve the accuracy of the energy estimates.

7.4.1.7 OpenMP and Multi-threading

The behaviour of OpenMP and multi-threaded programs should be subject to further studies. During our measurements, the OpenMP program doubled the energy

consumption when performing energy measurements in C++. No experiments have been performed using the built-in thread functionality in C++.

The problematic behavior of the OpenMP program when run with `taskset(1)` resulted in disabling running of programs using `taskset(1)` by default. A possible workaround is giving users control of which cores programs should run on (e.g., only run on small the cores or on a single big core), or what type of program it is (e.g., single-threaded programs are run on one core by default using `taskset(1)`, and programs using OpenMP have access to all cores). Possible solutions that work for both OpenMP, and other multi-threaded programs should be studied.

7.4.2 Improving Security

7.4.2.1 Improving Correctness Testing of Programs

Currently the system only has one hidden test case. As illustrated in Listing 3.5 this can give students insights about details of the test case, resulting in solutions that do not work in general.

To mitigate this problem, multiple hidden correctness tests should be implemented. These should test different edge cases of the program to ensure that the programs work correctly.

7.4.2.2 Further Restrict Access in the Chroot Jail

Two way approaches to give access to necessary files in the chroot jail were described in Section 4.4.2.1. The approach not implemented, which gave access to fewer files could be implemented to strengthen security.

7.4.2.3 Compilation and Profiling in Secure Environments

Attacks on the system can also occur during compilation and profiling of programs. The chroot jail can be modified to be used for securing these steps. This would make the system more secure to attacks.

7.4.2.4 Reimplementing Resource Limitations

Memory limitations were implemented using `setrlimit(2)`. However, this approach restarts accounting for every child process spawned, meaning the implemented memory limiting only is effective for single-threaded programs. However, `cgroups(5)` can be used to limit resources for *process groups*.

7.4.3 Other

7.4.3.1 Supporting Multiple Languages and Compilers

Support for multiple languages can be implemented by creating abstractions that do not depend on a specific language. The judges in Section 2.2.3 use abstractions called *Runner* (or something similar), where all language- and compiler-specific commands are collected in a single object.

To implement this, the database must add support for different languages and compiler versions. The front end must have the ability to select language when uploading, and scoreboards must show the languages used.

The improvements to measurement stability and security that are implemented in this thesis are implemented in a language-agnostic way, and could be adapted for use with other languages with minimal modifications.

7.4.3.2 Supporting Multiple Back Ends

Supporting multiple back ends have to separate use cases, which can be solved using the same tools: supporting multiple *different* back ends, and supporting multiple *equal* back ends.

By using a tool to manage queues and a broker to distribute submissions to back ends, both use cases can be covered. The queue management tool would have one queue for each type of back end, and the broker would pop submissions from a queue and run them on the respective back end—preferably running multiple submissions on multiple back ends asynchronously.

When multiple back ends of the same type are connected, the broker can run different submissions on different back ends. However, great care must be taken to ensure that the back ends produce equal results.

Chavez [Cha16] implemented a prototype broker for the CMB system, and his work should be considered if implementing support for multiple back ends.

7.4.3.3 Installing a Newer Compiler

Installing a newer version of the GCC compiler can give access to newer features of the C++ language. Most notably, GCC 9.1 provides *execution policies* which is interesting when studying energy-efficiency for multi-threaded programs.

GCC 9.1 can be installed by following the instructions here: <https://launchpad.net/~jonathonf/+archive/ubuntu/gcc-9.1>. The scripts compiling programs must be updated to use the new version.

7.4.3.4 Implementation Language

Magnussen [Mag16] suggested that all scripts should be implemented in Python for easier debugging and testing. This was partially implemented by Ingebrigtsen [Ing17], but code that could interfere with measurements was still written in Bash. Scripts in different languages make the system harder to maintain. This thesis has moved much of the code related to measurements to C++.

It is unclear whether script implemented in Python can interfere with measurement stability. However, Python lacks good interfaces to system calls and implementing all scripts in Python would be impossible.

We suggests implementing all functionality on the back end in C++. This provides a good compromise: C++ is more well behaved than Bash, making it easier to debug; it has good testing frameworks, it has access to lower level utilities of the Linux kernel through system calls and it should have minimal impact on measurement stability compared to Bash and Python.

Bibliography

- [AER18] Tab Atkins, Elika Etemad, and Florian Rivoal. *CSS Snapshot 2018 W3C Working Group Note*. Working Group Note. W3C, 2018. URL: <https://www.w3.org/TR/css-2018/>.
- [ALSU07] A Aho, M Lam, R Sethi, and J Ullman. *Compilers: Principles, Techniques and Tools, 2nd Editio*. Pearson Higher Education, 2007.
- [Ang] *AngularJS*. URL: <https://angularjs.org/> (visited on 11/19/2018).
- [Asy] *std::async*. URL: <https://en.cppreference.com/w/cpp/thread/async> (visited on 07/18/2019).
- [Bas] *bash (1) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man1/bash.1.html> (visited on 07/18/2019).
- [Bie11] Christian Bienia. “Benchmarking Modern Multiprocessors”. PhD thesis. Princeton University, Jan. 2011.
- [Big] *big.LITTLE*. URL: <https://developer.arm.com/technologies/big-little> (visited on 11/21/2018).
- [Bra17] Tim Bray. *The JSON Data Interchange Syntax*. Standard 404. Geneva, CH: ECMA International, 2017.
- [Cap] *capabilities (7) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man7/capabilities.7.html> (visited on 07/18/2019).
- [Cha16] Christian Chavez. “Climbing Mont Blanc and Scalability”. MA thesis. IDI, NTNU, 2016.

-
- [Chr] *chroot (2)* — *Linux man pages*. URL: <http://man7.org/linux/man-pages/man2/chroot.2.html> (visited on 07/18/2019).
- [CKC12] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. *Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology*. White Paper. Samsung Electronics Co., Ltd., 2012. URL: https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf.
- [Clo] *clock_getres (2)* — *Linux man pages*. URL: http://man7.org/linux/man-pages/man2/clock_gettime.2.html (visited on 07/18/2019).
- [CN13] J. M. Cebrián and L. Natvig. “Temperature effects on on-chip energy measurements”. In: *2013 International Green Computing Conference Proceedings*. June 2013, pp. 1–6. DOI: 10.1109/IGCC.2013.6604484.
- [Coda] *Codecademy*. URL: <https://www.codecademy.com/> (visited on 11/27/2018).
- [Codb] *Codewars*. URL: <https://codewars.com/> (visited on 12/07/2018).
- [Cora] *Cortex-A15*. URL: <https://developer.arm.com/products/processors/cortex-a/cortex-a15> (visited on 12/01/2018).
- [Corb] *Cortex-A7*. URL: <https://developer.arm.com/products/processors/cortex-a/cortex-a7> (visited on 12/01/2018).
- [Cppa] *Pseudo-random number generation*. URL: <https://en.cppreference.com/w/cpp/numeric/random> (visited on 06/25/2019).
- [Cppb] *Standard for the C++ Language*. Standard. Geneva, CH: International Organization for Standardization, Dec. 2017.
- [Cro] *Cron Howto*. URL: <https://help.ubuntu.com/community/CronHowto> (visited on 07/22/2019).
- [DDC+19] Marc Duranton, Koen De Bosschere, Bart Coppens, Christian Gamrat, Madeleine Gray, Harm Munk, Emre Ozer, Tullio Vardanega, and Oliver Zendra. *The HiPEAC Vision 2019*. Tech. rep. HiPEAC, 2019.
- [DDG+17] Marc Duranton, Koen De Bosschere, Christian Gamrat, Jonas Maebe, Harm Munk, and Olivier Zendra. *The HiPEAC Vision 2017*. Tech. rep. HiPEAC, 2017.

-
- [Din70] E. A. Dinic. “Algorithm for solution of a problem of maximum flow in networks with power estimation”. In: *Soviet Math. Doklady* 11 (1970), pp. 1277–1280. URL: <https://ci.nii.ac.jp/naid/10021311931/en/>.
- [Doc] *Docker*. URL: <https://www.docker.com/> (visited on 12/08/2018).
- [Doma] *DOMjudge*. URL: <https://www.domjudge.org/> (visited on 11/27/2018).
- [Domb] *DOMjudge (GitHub repository)*. URL: [DOMjudge\(GitHubrepository\)](#) (visited on 11/27/2018).
- [EK72] Jack Edmonds and Richard M Karp. “Theoretical improvements in algorithmic efficiency for network flow problems”. In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264.
- [EKN+11] Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann. “Five years with kattis—using an automated assessment system in teaching”. In: *Frontiers in Education Conference (FIE), 2011*. IEEE. 2011, T3J-1.
- [Ene] *Hardkernel EnergyMonitor (GitHub repository)*. URL: <https://github.com/hardkernel/EnergyMonitor> (visited on 11/27/2018).
- [Exea] *exec (3) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man3/exec.3.html> (visited on 07/18/2019).
- [Exeb] *exec (3p) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man3/exec.3p.html> (visited on 07/18/2019).
- [Exec] *execve (2) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man2/execve.2.html> (visited on 07/18/2019).
- [Fai] *Fail2Ban*. URL: https://www.fail2ban.org/wiki/index.php/Main_Page (visited on 07/18/2019).
- [FEL+17] Steve Faulkner, Arron Eicholz, Travis Leithead, Alex Danilo, and Sangwhan Moon. *HTML 5.2 W3C Recommendation*. Standard. W3C, 2017.
- [Flaa] *Flask*. URL: <http://flask.pocoo.org/> (visited on 11/19/2018).
- [Flab] *Flask-SocketIO*. URL: <https://flask-socketio.readthedocs.io/en/latest/> (visited on 06/11/2019).
- [For06] Michal Forišek. “Security of programming contest systems”. In: *Information Technologies at School* (2006), pp. 553–563.
-

-
- [FT02] Roy T. Fielding and Richard N. Taylor. “Principled Design of the Modern Web Architecture”. In: *ACM Trans. Internet Technol.* 2.2 (May 2002), pp. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: <http://doi.acm.org/10.1145/514183.514185>.
- [Gcc] *Using the GNU Compiler Collection (GCC)*. <https://gcc.gnu.org/onlinedocs/gcc/index.html>. Free Software Foundation, Inc. 2019.
- [Gev] *gevent*. URL: <http://www.gevent.org/> (visited on 06/20/2019).
- [Goo] *Google Analytics*. URL: <https://www.google.com/analytics> (visited on 11/19/2018).
- [Guna] *Gunicorn*. URL: <https://gunicorn.org/> (visited on 11/19/2018).
- [Gunb] *Gunicorn Architecture*. URL: <https://docs.gunicorn.org/en/stable/design.html> (visited on 06/20/2019).
- [Hac] *HackerRank*. URL: <https://www.hackerrank.com/> (visited on 12/08/2018).
- [HHSR13] Steven Halim, Felix Halim, Steven S Skiena, and Miguel A Revilla. *Competitive Programming 3*. Lulu Independent Publish, 2013.
- [HT99] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999. ISBN: 0201612622X, 9780201616224.
- [Icp] *International Collegiate Programming Contest (ICPC)*. URL: <https://icpc.baylor.edu/> (visited on 12/08/2018).
- [Idi] *IDI Open 2015*. 2015. URL: <https://idiopen.idi.ntnu.no/open15/> (visited on 04/17/2019).
- [Inga] *INGInious*. URL: <https://inginius.org/> (visited on 12/08/2018).
- [Ingb] *INGInious GitHub repository*. URL: <https://github.com/UCL-INGI/INGInious> (visited on 12/08/2018).
- [Ing17] Fredrik Pe Ingebrigtsen. “Climbing Mont Blanc – Back-end Improvements”. MA thesis. IDI, NTNU, 2017.
- [Jen] *Jenkins CI*. URL: <https://jenkins.io/> (visited on 12/04/2018).
- [Jut] *Jutge.org*. URL: <https://jutge.org/> (visited on 11/27/2018).
- [Kat] *Kattis*. URL: <https://kattis.com/> (visited on 12/07/2018).
- [LM16] Johannes Omber Lier and Thea Christine Mathisen. “Experiments towards digital exam with auto-grading in C++ programming courses”. MA thesis. IDI, NTNU, 2016.
- [Mag16] Sindre Magnussen. “Improving System Usability of Climbing Mont Blanc – An Online Judge for Energy Efficient Programming”. MA thesis. IDI, NTNU, 2016.

-
- [McC04] Steve McConnell. *Code Complete*. 2nd ed. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 0735619670, 9780735619678.
- [Min] *Minimal — Ubuntu Wiki*. URL: <https://en.cppreference.com/w/cpp/numeric/random> (visited on 07/10/2019).
- [Mon] *The Mont-Blanc project*. URL: <http://montblanc-project.eu/> (visited on 11/16/2018).
- [Mys] *MySQL*. URL: <https://www.mysql.com/> (visited on 11/19/2018).
- [Neg12] Christopher Negus. *Linux Bible*. 8th ed. John Wiley & Sons, 2012.
- [NFS+15] L. Natvig, T. Follan, S. Støa, S. Magnussen, and A. Garcia Guirado. “Climbing Mont Blanc - A Training Site for Energy Efficient Programming on Heterogeneous Multicore Processors”. In: *ArXiv e-prints* (2015). arXiv: 1511.02240.
- [Ngi] *NGINX*. URL: <https://nginx.com/> (visited on 11/19/2018).
- [Nic] *nice (1) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man1/nice.1.html> (visited on 07/18/2019).
- [Nod] *Node.js*. URL: <https://nodejs.org/> (visited on 11/16/2018).
- [NSLH19] Lasse Natvig, Magnus Själander, and Magnus Lie Hetland. *Climbing Mont Blanc – A Case Study in Challenging the Most Eager Students in a Large Programming Class*. This report presents the status of the project per June 2018. July 2019. DOI: 10.5281/zenodo.3345829. URL: <https://doi.org/10.5281/zenodo.3345829>.
- [Num] *NumPy*. URL: <https://numpy.org> (visited on 07/18/2019).
- [Odra] *Odroid Wiki*. URL: <https://wiki.odroid.com/> (visited on 12/12/2018).
- [Odrb] *Odroid XU3*. URL: <https://www.hardkernel.com/shop/odroid-xu3/> (visited on 11/21/2018).
- [Opea] *OpenCL C++ Bindings Documentation*. URL: <https://github.com/khronos.org/OpenCL-CLHPP/> (visited on 07/15/2019).
- [Opeb] *OpenMP Compilers & Tools*. URL: <https://www.openmp.org/resources/openmp-compilers-tools/> (visited on 08/02/2019).
- [Pep] *PEP 373 — Python 2.7 Release Schedule*. URL: <https://www.python.org/dev/peps/pep-0373/> (visited on 12/05/2018).

-
- [PGR12] Jordi Petit, Omer Giménez, and Salvador Roura. “Judge.Org: An Educational Programming Judge”. In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE ’12. New York, NY, USA: ACM, 2012, pp. 445–450. ISBN: 978-1-4503-1098-7. DOI: 10.1145/2157136.2157267. URL: <http://doi.acm.org/10.1145/2157136.2157267>.
- [Pyl] *Pylint*. URL: <https://www.pylint.org/> (visited on 07/18/2019).
- [Pyta] *Pytest*. URL: <https://docs.pytest.org/en/latest/> (visited on 07/18/2019).
- [Pytb] *Python*. URL: <https://www.python.org/> (visited on 11/19/2018).
- [Ren] *renice (1) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man1/renice.1.html> (visited on 07/18/2019).
- [Rev] *What is a reverse proxy server?* URL: <https://www.nginx.com/resources/glossary/reverse-proxy-server/> (visited on 11/19/2018).
- [RML08] Miguel A. Revilla, Shahriar Manzoor, and Ruijia Liu. “Competitive Learning in Informatics: The UVa Online Judge Experience”. In: *Olympiads in Informatics 2* (2008), pp. 131–148.
- [Sch] *sched_setaffinity (2) — Linux man pages*. URL: http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html (visited on 07/18/2019).
- [Scia] *SciPy*. URL: <https://www.scipy.org/> (visited on 07/18/2019).
- [Scib] *scipy.integrate.simps*. URL: <https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.integrate.simps.html> (visited on 11/26/2018).
- [Sel] *SELinux*. URL: https://selinuxproject.org/page/Main_Page (visited on 12/08/2018).
- [Seta] *setcap (8) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man8/setcap.8.html> (visited on 07/18/2019).
- [Setb] *setrlimit (2) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man2/setrlimit.2.html> (visited on 07/18/2019).
- [Setc] *setuid (2) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man2/setuid.2.html> (visited on 07/18/2019).
- [SF15] Simen Støa and Torbjørn Follan. “Climbing Mont Blanc – A Prototype System for Online Energy Efficiency Based Programming Competitions on ARM Platforms”. MA thesis. IDI, NTNU, 2015.

-
- [She] *ShellCheck*. URL: <https://www.shellcheck.net/> (visited on 07/18/2019).
- [Soc] *Socket.io*. URL: <https://socket.io/> (visited on 11/19/2018).
- [Sqla] *SQLAlchemy*. URL: <https://sqlalchemy.org/> (visited on 11/19/2018).
- [Sqlb] *SQLite*. URL: <https://sqlite.org/> (visited on 11/19/2018).
- [Sysa] *Generic Thermal Sysfs driver How To*. URL: <https://www.kernel.org/doc/Documentation/thermal/sysfs-api.txt> (visited on 07/28/2019).
- [Sysb] *std::system*. URL: <https://en.cppreference.com/w/cpp/utility/program/system> (visited on 07/18/2019).
- [Tas] *taskset (1) — Linux man pages*. URL: <http://man7.org/linux/man-pages/man1/taskset.1.html> (visited on 07/18/2019).
- [Tdta] *TDT4102 coursepage*. URL: <https://www.ntnu.edu/studies/courses/TDT4102> (visited on 11/16/2018).
- [Tdtb] *TDT4120 coursepage*. URL: <https://www.ntnu.no/studier/emner/TDT4120> (visited on 12/08/2018).
- [Tdtc] *TDT4200 coursepage*. URL: <https://www.ntnu.edu/studies/courses/TDT4200#tab=omEmnet> (visited on 04/17/2019).
- [Ter17] Brian Terlson. *ECMAScript 2018 Language Specification*. Standard 262. Geneva, CH: ECMA International, 2017.
- [Tii] *INA231 High- or Low-Side Measurement, Bidirectional Current and Power Monitor With 1.8-V I2C Interface*. SBOS644C. Rev. C. Texas Instruments. Mar. 2018.
- [Ufw] *Uncomplicated Firewall*. URL: <https://wiki.ubuntu.com/UncomplicatedFirewall> (visited on 07/18/2019).
- [Uhu] *uHunt — UVa Hunting*. URL: <https://uhunt.onlinejudge.org/> (visited on 11/27/2018).
- [Uni] *unittest*. URL: <https://docs.python.org/3.6/library/unittest.html> (visited on 07/18/2019).
- [Uva] *UVa Online Judge*. URL: <https://uva.onlinejudge.org/> (visited on 11/27/2018).
- [Vir] *Virtualenv*. URL: <https://virtualenv.pypa.io/en/stable/> (visited on 07/18/2019).
-

-
- [WAB+18] Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. “A Survey on Online Judge Systems and Their Applications”. In: *ACM Comput. Surv.* 51.1 (Jan. 2018), 3:1–3:34. ISSN: 0360-0300. DOI: 10.1145/3143560. URL: <http://doi.acm.org/10.1145/3143560>.
- [Wys17] Rafael J. Wysocki. *The Linux kernel user’s and administrator’s guide — CPU Performance Scaling*. 2017. URL: <https://www.kernel.org/doc/html/v4.14/admin-guide/pm/cpufreq.html> (visited on 07/16/2019).
- [YL06] Tatu Ylonen and Chris Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. IETF, Jan. 2006.

Appendix A

Installation Instructions

A.1 Back End

1. Download the Ubuntu 18.04 Minimal image from https://wiki.odroid.com/odroid-xu4/os_images/linux/ubuntu_4.14/ubuntu_4.14. Flash the image to a SD-card or eMMC-card (e.g., by using `dd(1)`).
2. On first boot, some configuration is done automatically before the board shuts down. The board is shut down when no LEDs are blinking (only a constant red LED). Start the board again.
3. To locate an Odroid connected to the internet, use `arp-scan`. Note that the computer running the command must be wired to the same switch (or router) as the Odroid.

```
sudo arp-scan --interface=eth0 --localnet
```

The Odroid will have a MAC address starting with `00:1e:06` in the list of discovered devices.

4. Connect to the board by using SSH. The default user/password combination is `root/odroid`.

```
ssh root@<IP-address>
```

5. Change the default password:

```
passwd
```

-
- The MAC address of the board can be changed by (replace XX with wanted number):

```
echo '00:e1:06:61:7a:XX' >> /media/boot/boot.ini.default
/usr/share/bootini/bootini-persistence.pl
```

- Install git, clone the repository and install the system:

```
apt-get install git
git clone <repository-url>
cd cmb-board
PASSWORD=<climber-password> ./install.sh
```

A.2 Server and Front End

For the server and front end set up, the instructions from Magnussen [Mag16] can be followed with a few exceptions:

- Use a Ubuntu 18.04 Minimal disk image.
- Add the following content to `/lib/systemd/system/cmb.service`:

```
[Unit]
Description=CMB system
After=nginx.service

[Service]
User=climber
Type=forking
EnvironmentFile=/srv/climber/cmb/server/cmb-flask/env
ExecStart=/srv/climber/cmb/server/cmb-flask/scripts/init_cmb.sh start
ExecStop=/srv/climber/cmb/server/cmb-flask/scripts/init_cmb.sh stop
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

- Enable and start the service using with the following commands:

```
sudo systemctl enable cmb
sudo systemctl start cmb
```

Appendix **B**

CMB Challenge 2019

B.1 General information, rules and technical information

This document contains general information, rules and technical information about the CMB Challenge 2019 competition. This was published to the students on the course page the day before the competition started.

CMB Challenge 2019

Introduction

There are 6 problems in the competition. All problems are unrelated and can be solved in any order. Problems 1-5 make use of randomly generated numbers, which is explained in the task document (you can find it on Blackboard when the competition has started). Problem 6 only uses input from `stdin`.

In order to participate in the competition you have to sign up on the CMB website (climb.idi.ntnu.no), and join the group “TDT4102 CMB Challenge 2019”.

If you need some help to get started, the HowTo-page on the CMB website might give useful hints. The Piazza-forum in the course has a tag for CMB-related questions, please do not hesitate to ask questions by posting on piazza. If you experience technical difficulties with the system, e.g., system crash or messages to contact the system administrator, you can send an email to okpeders@stud.ntnu.no — please give enough details (username, problem name, code) about your problem in order for us to help you faster.

Thanks for helping to improve the CMB system, and good luck!

General information

Specification of the problems will be published to the folder “CMB Challenge” on Blackboard when the competition starts. There will be a prize for the three best programmers, but the prizes have not yet been decided.

Competition rules

- Only for students in the TDT4102 course spring 2019
- Competition starts on the 13th of March, 12:00.
- Competition ends on the 3rd of April, 12:00.

Scoring rules

- a) 1 point for every problem with a valid solution.
- b) 3 points for the fastest solution to a problem, 2 points for the second fastest solution and 1 point for the third fastest solution to a problem (except for problem 6, see below).
- c) 3 points for the most energy-efficient solution to a problem (measured by the EDP-value), 2 points for the second most energy-efficient solution to a problem and 1 point for the third most energy-efficient solution to a problem (except for problem 6, see below).
- d) For problem 6, the best, second best and third best solutions will get 6, 4 and 2 points, respectively. The “pareto-metric” $EDP * distance$ is used here, where the distance is the distance of the path generated by the submitted solution.
- e) Points can be shared among multiple students if the difference in one of the competition metrics is smaller than the precision of that CMB metric. E.g., if no 1 and no 2 are very close, the judges might decide that both will get $(3 + 2)/2 = 2.5$ points.

Technical note

The system will test your uploaded solution by running two separate test cases, one big and one small test case. The energy and time measurements are performed while running the program on the big test case. The results will be displayed if you pass both test cases.

The CMB system runs on an Ubuntu Linux system and compiles the code using a gcc/g++ compiler. The compiler uses the C++11 standard, so code using C++14/17/20 features will not compile.

Previously, some users received compilation errors on the system, but not locally. MS-VS and Xcode sometimes have a couple of extra (read: non-standard) features in the language, and these will usually not compile on the system. Also, these IDEs sometimes include some extra headers for you, but the compiler on the system doesn't necessarily include the same headers. Make sure that you include all of your headers explicitly to avoid errors (include too many, rather than too few).

The `std.lib.facilities.h`-header used in this course is **not** available in the system, so you will receive compilation errors if you try to use it.

B.2 Problem descriptions

This document contains the problem descriptions for the problems given in the competition.

CMB Challenge 2019 — Problems

March 13, 2019

About pseudo-random numbers

In problems 1-5 you have to generate random numbers using the code below. The code below gives the same random numbers on all systems, **do not** use `rand()` or `<random>` for these tasks!

```
#include <algorithm>
using namespace std;
class RandInt {
private:
    static const unsigned int INCREMENT = 0xC39EC3;
    static const unsigned int MULTIPLIER = 0x43FD43FD;
    unsigned int m_nRnd;
public:
    RandInt(unsigned int nSeed) : m_nRnd(nSeed) {};
    int getInt(int nFrom, int nTo) {
        if (nTo < nFrom)
            swap(nTo, nFrom); // include algorithm
        else if (nTo == nFrom)
            return nTo;
        m_nRnd = (m_nRnd*MULTIPLIER + INCREMENT) & 0xFFFFF;
        float fTmp = (float)m_nRnd / 16777216.0;
        return (int)((fTmp*(nTo - nFrom + 1)) + nFrom);
    }
};
```

The class `RandInt` has a constructor taking a seed to initialize the random number generation. It also has a member function `getInt` which takes two numbers, *from* and *to* and return a random integer in the closed range [*from*, *to*], (i.e., a number *n* such that $from \leq n \leq to$).

Example:

A typical task will require you to read some numbers (including a seed) from `stdin` (you can use `cin` for this), and then do some calculations before you write the answer to `stdout` (you can use `cout` for this). The program below demonstrates how you can read variables from `stdin`, generate `N` numbers in the closed range `[0, 100]` (i.e., both 0 and 100 are in the range), and write the sum of these to `stdout`.

```
#include <algorithm>
#include <iostream>
using namespace std;
class RandInt {
    // skipped for brevity, same as above
};

int main() {
    int N, S; // N, and the seed
    cin >> N >> S;
    RandInt generator(S);
    int sum{0};
    for (int i = 0; i < N; i++) {
        int random_number = generator.getInt(0, 100);
        sum += random_number;
    }
    cout << sum << '\n';
}
```

Input:

Output:

1 To Quote Hamlet..

A lot of seabirds, like puffins and certain types of seagulls, are considered to be endangered or threatened, due to human interference. To gain a better understanding of the situation, as well as observing how effective the attempts to make up for the damage are, special methods for counting and observing the birds has been developed.

You are going to observe a part of a large seabird sanctuary, and you will have to walk for several hours to get there. You are joined by another observer, Dr. K. Meis, to make sure the resulting data is as accurate as possible. Dr. Meis is really talkative, and can't seem to stop quoting more or less brilliant fictional characters. You start counting who Dr. Meis is quoting as a way of preparing your brain for the bird counting. To make it easier you give each person Dr. Meis is quoting a number, see the table below.

0	Hamlet	10	Mrs Hudson
1	Macbeth	11	Luna Lovegood
2	Lady Trent	12	Ferguson Bishop
3	Professor McGonagall	13	Gandalf the Grey
4	Mycroft Holmes	14	Cruella de Vil
5	Pippi Longstocking	15	Sherlock Holmes
6	Vera Stanhope	16	Professor Dumbledore
7	Hercule Poirot	17	Professor Trelawney
8	Professor Kroll	18	Lindelin Rosenquist
9	Veronica Mars	19	Keith Mars

Input:

The program is supposed to read two integers from standard input. They are both situated on the first line. The first number, N , is the number of quotes. The second number, S , is a seed used to generate the needed data. (See the section about random numbers at the beginning of this document). Generate N integers in the closed interval $[0, 19]$. Every number represents a quotation, and the numbers corresponds with the numbers in the table above. You can assume that $0 < N \leq 1\,000\,000\,000$ and $0 < S \leq 2\,000\,000\,000$.

Output:

The program is supposed to write the name of the person Dr. K. Meis quoted the most, followed by how many times this person was quoted to standard output.

For format see example below. Make sure to use the exact same format with uppercase/lowercase letters in the right places. Every place containing white characters (space) is to contain exactly one white character.

Example:

Input:

23 11

Output:

Lady Trent 3

2 Pirates and probability

There is a pressing need to find a safe place to hide your amazing treasure after yet another successful raid. As captain, deciding where to hide it is your responsibility. This is a huge responsibility, and making the right choice is crucial. But the list containing all the possible hiding places seems endless, how in the name of Jacquotte Delahaye “Back from the Dead Red” are you going to pick the best place? After hours of frowning, headache, and pondering you are about to give up. You climb the main mast in a last desperate attempt to sort this mess out. And that’s when it hits you: There are only three factors you need to take into account when picking the perfect hiding place. These three are:

1. The probability that you will be able to recover it.
2. The probability that NO ONE ELSE will find it.
3. The probability that animal life and the environment WILL NOT be negatively affected by it. A habitable planet is, after all, crucial to pirate activity. And there is no planet B, not even for pirates.

After this epiphany you proceed to go through the list of hiding places and write the probability (in percent) for each of the three factors stated above.

The three factors are weighted as follows: factor 1 has weight 0.4, factor 2 and 3 are both weighted with 0.3 (i.e. $w_1 = 0.4$, $w_2 = w_3 = 0.3$).

Pleased with this brilliant method for solving the most difficult problem of the week, you decide to write a little program to solve it for you. The list is, after all, incredibly long...

Input:

The program is supposed to read two integers from standard input. They are both situated on the first line. The first number, N , is the number of hiding places. The second number, S , is a seed used to generate the needed data. (See the section about random numbers at the beginning of this document). You can assume that $0 < N \leq 5\,000\,000$ and $0 < S \leq 2\,000\,000\,000$.

When generating the numbers:

- All numbers are to be generated in the closed interval $[0, 100]$.

- The N first values are the probability for factor 1, the N next values are the probability for factor 2 and the last N values are the probability for factor 3. (Where the first value for factor 1 is the factor 1 value for place 1, etc).

The first place on the list is called 1, the second is called 2, \dots , the N^{th} is called N . Calculate the score for each place using the probabilities and the weights.

When sorting, comparing or printing numbers you should use integers. Use floating point values in intermediary calculations if necessary.

Output:

Your program should write the following to standard output:

A list of the five places with the highest score, and a list of the five places with the lowest score. The list of the five best places should be sorted accordingly: The places with the highest score are to be written before the places with lower score. If several places have the same score, the place with the lowest number (name) is to be written first.

The list of the five places with the lowest score should be sorted accordingly: The places with the lowest score are to be written before the places with higher score. If several places have the same score, the place with the lowest number (name) is to be written first.

For format see example below. Make sure to use the exact same format with commas, dots and uppercase/lowercase letters at the right places, as well as correct sorting. Every place containing white characters (space) is to contain exactly one white character.

p1 is the probability for factor 1, p2 is the probability for factor 2, p3 is the probability for factor 3.

Example:

Input:

50 8

Output:

```
The 5 places with the highest score are:  
place: 23, score: 89, p1: 92, p2: 75, p3: 100.  
place: 4, score: 82, p1: 94, p2: 95, p3: 53.  
place: 47, score: 79, p1: 93, p2: 82, p3: 57.  
place: 42, score: 76, p1: 97, p2: 44, p3: 79.  
place: 45, score: 76, p1: 65, p2: 87, p3: 80.  
The 5 places with the lowest score are:  
place: 34, score: 17, p1: 15, p2: 30, p3: 6.  
place: 37, score: 22, p1: 24, p2: 14, p3: 28.  
place: 2, score: 27, p1: 37, p2: 30, p3: 9.  
place: 30, score: 27, p1: 58, p2: 8, p3: 5.  
place: 28, score: 30, p1: 30, p2: 57, p3: 3.
```

3 The Huckybucky forest

Climate change is one of the biggest issues humanity is currently facing. The increase in the emissions of greenhouse gases increases the greenhouse effect, which in turn leads to global warming. To try and calculate how much an individual, country or event emits, one often uses carbon footprint. To compare the emissions from the different greenhouse gases one usually converts the values to CO₂-equivalents. When calculating someone's or something's carbon footprint one adds all the CO₂-equivalents emitted by this person/thing. The emissions can be both direct and indirect carbon emissions; direct emissions can be a result of for instance transport, indirect emissions are usually a result of the products we consume, such as food and clothes. Studies have shown that a meat-based diet emits more CO₂-equivalents than a plant-based diet.

You have decided to reduce your carbon footprint by eating more plant-based food. Two seconds later you realise that this is the perfect moment for moving into the Huckybucky forest (Norwegian: Hakkebakkeskogen), because their new law makes it illegal to eat meat. But then you start wondering; will moving into the Huckybucky forest actually reduce your carbon footprint? The emissions resulting from your food will most definitely decrease, but will the emissions resulting from other parts of your life (such as transport and heating) increase, decrease or stay stable?

Calculate the amount of CO₂-equivalents emitted as a result of your life in the city and your life in the Huckybucky forest.

Input

Your program is supposed to read three integers from standard input. All three are situated on the first line. The first number, N , is the number of emissions per month. The second number, M , is the number of years. And the third number, S , is a seed used to generate the needed data. (See the section about random numbers at the beginning of this document). You can assume that $0 < N \leq 500\,000$, $0 < M \leq 2\,000$ and $0 < S \leq 2\,000\,000\,000$.

When generating the numbers:

- All values are to be generated in the closed interval $[0, 1000]$.
- Two sets of values are to be generated, generate the set for the city first, then the set for the Huckybucky forest.

In each set the N first values are the emissions in the first month of the first

year, the N next are the emissions in the second month of the first year, . . . , the N last are the emissions in the 12th month of the M th year. The sets represent the CO₂-equivalents you would emit if you lived in the city and the Huckybucky forest respectively.

When sorting, comparing or printing numbers you should use integers. Use floating point values in intermediary calculations if necessary.

Output

Your program should write the following to standard output:

The first line of your output should include the total increase in CO₂-emissions from the M years, comparing the emissions from your new life in the Huckybucky forest with the emissions from life in the city, i.e. the increase in CO₂-emissions as a result of you moving to the Huckybucky forest. The value is to be given as percent. A decrease in emissions is to be stated with a minus-sign (−) in front of the percentage-value, as you can see from the example.

Your output should also include a list with one line for each year, stating which months living in the Huckybucky forest decreased your CO₂-emissions. If living in the Huckybucky forest didn't decrease your CO₂-emissions in any month in a given year, this year is to be excluded from the list.

For format see example below. Make sure to use the exact same format with dots and uppercase/lowercase letters in the right places, as well as correct sorting. Every place containing white characters (space) is to contain exactly one white character. Please note that the first year is referred to as year 0 in the output.

Example:

Input:

```
20 3 8
```

Output:

```
A total of -3% more CO2 than usual was emitted.  
Living in the Huckybucky forest reduced CO2 emissions in:  
March April June August September October in year 0  
January March June July September October November in year 1
```

January February March April July August September November in year 2

4 In Ventus

Everyone who has walked outside on a windy day knows that the wind is full of power. Wind turbines make it possible to convert the energy in the wind into electrical energy. And seeing as wind power doesn't consume any water, uses little land, is renewable, clean, doesn't emit any greenhouse gases during operation, has declining installation costs and wind is plentiful, it comes as no surprise that this is a sustainable power source growing in popularity.

A new wind turbine is going to be connected to the grid. There are, however, several vertices (nodes) between the turbine and the main grid. The power will have to flow via several of these vertices, and the vertices themselves are connected in a somewhat chaotic manner. The lines (edges) connecting the vertices all have different capacities, so the flow they can carry varies. Find the maximum possible flow from the wind turbine to the main grid with the three following constraints:

1. Flow on an edge (line) doesn't exceed the given capacity of the edge.
2. Incoming flow is equal to outgoing flow for every vertex except the wind turbine and the final vertex.
3. For a given edge, with start vertex u and end vertex v , flow can only be transported over the edge from u to v and not from v to u .

Input:

The first line of in standard input contains four integers: E , V , S and C in that order.

- $0 < E \leq 9\,000\,000$, is the number of lines (edges).
- $0 < V \leq 10\,000$, is the number of vertices (nodes).
- $0 < S \leq 2\,000\,000\,000$, is the seed used to generate the needed data.
- $0 < C \leq 10\,000$, is the maximum capacity of any line (edge).

See the beginning of this document for information about generating random numbers.

You are to generate a maximum of $3 \cdot E$ numbers because each edge has both a start vertex, end vertex and capacity. When generating numbers, do the following:

1. Generate the start node of the first edge, u , in the closed interval $[0, V - 2]$.
2. Generate the end node of the first edge, v , in the closed interval $[1, V - 1]$.
3. If, and only if, u and v are different nodes, generate the capacity, c , in the closed interval $[1, C]$. Edges which starts and ends in the same node (“loops”) are considered non-existent, meaning the generated network may contain fewer than E edges.
4. There can be multiple edges with different capacities between two nodes. These can be considered as a single “large” edge by summing the capacity of all the “smaller” edges between the two nodes.

The wind turbine is located at node 0, and the main grid is located at node $V - 1$.

Output:

Your program should write one line containing the maximum flow from the wind turbine to the main grid to standard output. For format see example below. Make sure to use the exact same format. Every place containing white characters (space) is to contain exactly one white character.

Example:

Input:

```
10 4 897 10
```

Output:

```
max flow 14
```

5 Flower power

Due to climate change and human interference, many species are threatened with extinction. Ecosystems are fragile structures, thus losing one species might upset the entire system. There are several different ways of preventing this from happening. One can, for instance, make an area of importance to wildlife, flora or fauna into a nature reserve, i.e. a protected area. This enables the species to live with as little human interference as possible, hopefully preventing them from going extinct.

A rectangular shaped area of size $w \cdot h$ is going to be made into a nature reserve in order to prevent a rare flower from going extinct. The nature reserve is to be somewhere inside the much larger area $W \cdot H$, with w parallel to W and h parallel to H . You join a group of scientists, and together you survey the area $W \cdot H$. Every observed individual of the endangered species, in this case the rare flower, is counted and the coordinates of the observed individual is added to the list. After hours of hard work, you have finally finished this part of the job. Now you'll have to try and find suitable candidates (areas of size $w \cdot h$) to become a nature reserve. It has been decided that an area is suitable if there are at least k individuals of the endangered flower inside it, i.e. at least k points.

Find an area (rectangle) $w \cdot h$ with at least k points inside it, i.e. a suitable candidate to become a nature reserve. In reality, this problem is not always solvable, but you can assume that the given input makes it solvable.

Input:

Your program is supposed to read seven integers from standard input ($N S W H w h k$). They are all situated on the first line. The first number, N , is the number of points, i.e. observations of the rare flower, in $W \cdot H$. The second number, S , is a seed used to generate the needed data (See the section about random numbers at the beginning of this document). The third number, W , is the width of the area. The fourth integer, H , is the height of the area. The fifth number, w , is the width of the nature reserve. The sixth number, h , is the height of the nature reserve. And the seventh number, k , is the number of points required for an area to be considered a suitable candidate to become a nature reserve. The origin is in the top left corner, and only integers are used, i.e. no coordinates are floating point values. You can assume that $0 < N \leq 5\,000\,000$, $0 < S \leq 2\,000\,000\,000$, $0 < W \leq 40\,000$, $0 < H \leq 40\,000$, $0 < w \leq W$, $0 < h \leq H$ and $0 < k \leq N$.

You are to generate $2 \cdot N$ numbers because there are N points and each point has both a x - and a y -coordinate. The first generated value is the x -coordinate

of the first point, the second value is the y -coordinate of the first point, the third value is the x -coordinate of the second point ... the $2N^{\text{th}}$ value is the y -coordinate of the N^{th} point. The x -coordinates are to be generated in the closed interval $[0, W - 1]$, the y -coordinates are to be generated in the closed interval $[0, H - 1]$.

Output:

Your program should write one line containing the coordinates of the top left corner of the nature reserve to standard output. For format see example below. Make sure to use the exact same format. Every place containing white characters (space) is to contain exactly one white character.

Example:

Input:

```
10 4 10 12 2 3 3
```

Output (one valid solution based on the given input):

```
x: 7 y: 8
```

6 There and Back Again

Cars using fossil fuels contribute to both global and local pollution, because they emit greenhouse gases and contribute to particulate matter. Thus, using other means of transportation is usually preferable from an environmental point of view. This is, sadly, not always possible. One can, however, try and make sure that the emissions are as small as possible. For instance, the quickest and shortest route is often better than the long one, because the amount of fuel used depends on the distance travelled.

You are going on an adventure. To make your adventure as environmentally friendly as possible you have decided to spend some time on picking the best travelling route. This took longer than expected, thus you decide to implement an algorithm to decrease your workload. You are pleased to notice that this is a version of the travelling salesperson problem (TSP) – a classic NP-hard combinatorial optimization problem – though it in this case should be called the travelling adventurer problem instead, or maybe the travelling burglar problem? Find the shortest closed walk in a complete undirected graph with the requirement that you visit every vertex/city exactly once.

Input:

Note: This task does not use random numbers!

Read from standard input. The first character in a line defines the type of information you are given.

c are comments
p n contain the number of nodes in the graph
v describe the location of a node
q describe the starting node

They have the following format:

c text just skip these
p n $0 < n \leq 10\,000$ is the number of nodes/vertices
v $t\ x\ y$ t is the number given to a node ($1 \leq t \leq n$)
 x and y are the coordinates of the node, $0 \leq x, y \leq 10\,000$
q b b is the source (and destination) node

All the numbers in the input are integers.

Output:

The output contains a single line with the order the nodes should be visited:

```
s v1 v2 ... vn vn+1
```

The line must contain all the nodes in the dataset, including both source (v_1) and destination (v_{n+1}). Note that since you have to get back to the starting city, $v_1 = v_{n+1}$.

Answers that produce a valid solution will be accepted, and the total distance of the given solution is calculated by the CMB-system and shown on the scoreboard (i.e., you don't have to produce the optimal solution). The big correctness test is so large that trying all possibilities is not a feasible solution.

Example:

Input:

```
c This is a comment, ignore me
p 8
v 1 2104 1968
v 2 1401 1968
v 3 295 1968
v 4 1235 1956
v 5 1347 1962
v 6 1401 1944
v 7 1211 1944
v 8 1211 1932
q 5
```

Output:

This is one of many possible answers, and might not be the optimal solution.

```
s 5 2 6 4 7 8 1 3 5
```

B.3 Questionnaire

This questionnaire was digitally distributed to the students using Google Forms after the competition ended.

CMB Challenge 2019 - feedback

Spørreundersøkelse for studenter i fag TDT4102 som deltok i CMB Challenge 2019. Svarene som samles inn vil kun bli brukt som feedback til CMB-prosjektet og vi bli anonymisert.

Skjemaet har bare 10 spørsmål og kan fylles ut på under 5 minutter, men bruke gjerne lenger tid i kommentarfeltet dersom du har lyst --- all feedback er meget nyttig for prosjektet!

* Required

1. Hvor mye bakgrunn har du i programmering ut over IT-GK ved NTNU (eller tilsvarende) og dette kurset TDT4102 (C++) ? *

Mark only one oval.

- Ingen
- Litt
- Programmering som hobby mer enn 1 år
- Har hatt deltidsjobb eller jobbet på programmeringsprosjekter på fritiden
- Enda mer, spesifiser gjerne på siste side (andre kommentarer)

2. I CMB-challenge hadde vi to ulike typer oppgaver. (a) Frittstående "tall-oppgaver" som involverer summering eller telling (eks. "To Quote Hamlet...", "Pirates and Probabilites" eller "The Huckybucky Forest") (b) Klassiske algoritmisk problem med mange anvendelser (eks. "In Ventus" = Maximum Flow, "There and Back Again" = Travelling Salesperson Problem) Hvilken av de to ulike oppgavetyperne likte du best ?--- prøv å se bort i fra at de hadde ulik vanskelighetsgrad *

Mark only one oval.

- (a) Frittstående "tall-oppgaver"
- (b) Klassiske problem med mange anvendelser

3. Hvor kjent er du med bruk av Kattis eller andre websteder for trening el. konkurranse i programmering? *

Mark only one oval.

- Har aldri brukt det
- Begynte å bruke det imens jeg har tatt faget, har brukt det LITT
- Begynte å bruke det imens jeg har tatt faget, har brukt det MYE
- Begynte å bruke det før jeg begynte i dette faget (TDT4102), har brukt det LITT
- Begynte å bruke det før jeg begynte i dette faget (TDT4102), har brukt det MYE

4. Det er tradisjon i programmingskonkurranser at oppgaver ofte er formulert med en god del ekstra tekst (ofte med et litt humoristisk preg) og informasjon som en ikke har bruk for. Slik har vi gjort det også for de fleste oppgavene i denne konkurransen. En annen type oppgaver er mer korte og presise, kan være uten spesiell anvendelse (såkalte "toy examples") og spesifiserer bare det du må vite. En tredje type er reelle problemer med beskrivelse av anvendelsen (f.eks. "In Ventus"), men ikke mer tekst enn nødvendig. Hvilken type oppgaveformulering liker du best? *

Mark only one oval.

- Oppgaver med ekstra (humoristisk) tekst (slik som de fleste oppgavene i denne konkurransen).
- Korte og presise, med minst mulig anvendelser.
- Oppgaver om "anvendelser" (F.eks. "In Ventus")

5. Andre kommentarer eller ideer om *oppgavetyper* for CMB?

6. I de fleste oppgavene i denne konkurransen genererte du tallene selv. Hvor godt synes du dette fungerte? *

Mark only one oval.

- Veldig bra
- Bra
- Middels
- Dårlig
- Veldig dårlig

7. Vi har i denne konkurransen brukt generering av randomiserte tall (a). En annen måte å hente inn input på kan være å lese det inn fra stdin (b) (dvs. bruke cin eller tilsvarende). Hvilken måte foretrekker du? *

Mark only one oval.

- (a) Generering av randomiserte tall
- (b) Lese input fra stdin.
- Other: _____

8. Hvor brukervennlig synes du CMB er? *

Mark only one oval.

- Veldig bra
- Bra
- Middels
- Dårlig
- Veldig dårlig

9. Har du kommentarer eller forslag på hvordan CMB kan forbedres mhp. brukervennlighet og ev. tekniske løsninger?

10. Har du gjort ***andre*** CMB-oppgaver enn de som ble gitt i konkurransen? *

Mark only one oval.

- Nei
- Ja, prøvd meg på andre oppgaver ETTER at konkurransen startet
- Ja, prøvd meg på andre oppgaver FØR (og evt. etter) at konkurransen startet

11. I hvilken grad synes du det å få feedback på kjøretid og energi-effektivitet på kode du har skrevet gir deg økt forståelse for din egen kode? *

Mark only one oval.

- I svært stor grad
- I meget stor grad
- I noen grad
- Ikke i det hele tatt
- Det bare forvirrer meg

12. Kommentarer og meninger om bruk av CMB i faget TDT4102 og hvordan det påvirker motivasjon for faget og interesse for videre studier?

Appendix C

Test Programs

This appendix displays the test programs used during experiments. The *Mandelbrot (OpenCL)* program uses depends on two files, while the remaining programs only have a single file.

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!\n";
}
```

Listing C.1: The *Hello World* test program.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    vector<long> vec;
    long x;
    while (cin >> x) {
        vec.push_back(x);
    }
    sort(begin(vec), end(vec));
    for (auto& x : vec)
        cout << x << '\n';
}
```

Listing C.2: The *Sort* test program.

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <random>
#include <limits>
using namespace std;

using engine = linear_congruential_engine<unsigned, 0xFD43FD,
                                           0xC39EC3, 0x1000000>;

class RandInt {
    engine m_eng;
    constexpr static auto range = engine::max() - engine::min() + 1;
public:
    RandInt(unsigned int nSeed)
        : m_eng(nSeed)
    {}

    int getInt(int nFrom, int nTo) {
        int num = m_eng();
        float fTmp = (float)num / range;
        return (int)((fTmp * (nTo - nFrom + 1)) + nFrom);
    }
};

int main() {
    ios::sync_with_stdio(false);
    unsigned N, S;
    cin >> N >> S;
    RandInt rng{S};
    vector<int> vec;
    for (unsigned i = 0; i < N; i++) {
        vec.push_back(rng.getInt(-20000000, 20000000));
    }
    sort(begin(vec), end(vec));
    cout << vec[N/2] << '\n';
}

```

Listing C.3: The *Sort w/RandInt* test program.

```

#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdint>
#include <cmath>
#include <sys/time.h>
#include <vector>

using namespace std;

#define MAX_ITER 255

int main(void)
{
    /* Width and height of the Mandelbrot data (i.e., # pixels) */
    int32_t width, height;
    cin >> width >> height;

    vector<uint8_t> image(width*height);

    for (int32_t y = 0; y < height; y++) {
        for (int32_t x = 0; x < width; x++) {
            // We're only interested in x \in [-2, 0.5], y \in [-1, 1]
            float initialReal = -2 + (x / (float)width * 2.5f);
            float initialImaginary = -1 + (y / (float)height * 2);

            float real = initialReal;
            float imaginary = initialImaginary;
            int iterations = 0;
            while (iterations < MAX_ITER &&
                real*real + imaginary*imaginary <= 4.0f ) {
                iterations++;
                if (iterations > MAX_ITER) {
                    break;
                }
                float oldReal = real;
                real = real*real - imaginary*imaginary + initialReal;
                imaginary = 2 * oldReal * imaginary + initialImaginary;
            }
        }
    }
}

```

```
        image[y*width + x] = iterations;
    }
}

int32_t count = 0;
for (int32_t i = 0; i < height*width; i++) {
    count += image[i] < MAX_ITER;
}
printf("%d\n", count);
}
```

Listing C.4: The *Mandelbrot* test program.

```

#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <sys/time.h>
#include <vector>
#include <sched.h>
#include <omp.h>

using namespace std;

#define MAX_ITER 255

int main(void)
{
    /* Width and height of the Mandelbrot data (i.e., # pixels).*/
    uint32_t width, height;
    cin >> width >> height;

    vector<uint8_t> image(width*height);

#pragma omp parallel for shared(image, height, width) \
    collapse(2) schedule(dynamic)
    for (uint32_t y = 0; y < height; y++) {
        for (uint32_t x = 0; x < width; x++) {
            // We're only interested in x \in [-2, 0.5], y \in [-1, 1]
            float initialReal = -2 + (x / (float)width * 2.5f);
            float initialImaginary = -1 + (y / (float)height * 2);

            float real = initialReal;
            float imaginary = initialImaginary;
            int iterations = 0;
            while (iterations < MAX_ITER &&
                real*real + imaginary*imaginary <= 4.0f ) {
                iterations++;
                if (iterations > MAX_ITER) {
                    break;
                }
                float oldReal = real;
                real = real*real - imaginary*imaginary + initialReal;
                imaginary = 2 * oldReal * imaginary + initialImaginary;
            }
        }
    }
}

```

```
    }  
  
    image[y*width + x] = iterations;  
  }  
}  
  
uint32_t count = 0;  
for (uint32_t i = 0; i < height*width; i++) {  
    count += image[i] < MAX_ITER;  
}  
printf("%d\n", count);  
}
```

Listing C.5: The *Mandelbrot (OpenMP)* test program.

```

#include <CL/cl.hpp>
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstdlib>
#include <cmath>
#include <sys/time.h>
#include <vector>

using namespace std;
using namespace cl;

void errorAndExit(string error_message) {
    cout << error_message << endl;
    exit(1);
}

int main(void)
{
    // get all platforms (drivers)
    vector<Platform> all_platforms;
    Platform::get(&all_platforms);

    if (all_platforms.size() == 0)
        errorAndExit("No platforms found.\n");

    Platform default_platform = all_platforms[0];

    vector<Device> all_devices;
    default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);

    if (all_devices.size() == 0)
        errorAndExit("No devices found.");

    Device default_device = all_devices[0];

    Context context({default_device});

    Program::Sources sources;
    ifstream ifs("mandelbrot.cl");
    if (ifs.fail()) errorAndExit("Failed to open mandelbrot.");
    string kernel_source {
        istreambuf_iterator<char>(ifs), istreambuf_iterator<char>()
    };
    sources.push_back({kernel_source.c_str(), kernel_source.size()});

```

```

Program program(context, sources);
if (program.build({default_device}) != CL_SUCCESS)
    errorAndExit("Error building program!");

/* Width and height of the Mandelbrot data (i.e., # pixels). */
cl_int width, height;
cin >> width >> height;

/* The output buffer is the size of the Mandelbrot data. */
std::size_t bufferSize = width * height * sizeof(cl_uchar);

/* Create an output buffer for final data. */
Buffer image_buffer(context,
                    CL_MEM_WRITE_ONLY | CL_MEM_ALLOC_HOST_PTR,
                    bufferSize);

CommandQueue queue(context, default_device);

cl::Kernel mandelbrot(program, "mandelbrot");
mandelbrot.setArg(0, image_buffer);
mandelbrot.setArg(1, width);
mandelbrot.setArg(2, height);

queue.enqueueNDRangeKernel(mandelbrot, NullRange,
                           NDRange(width/4, height), NullRange);

vector<u_char> image(width*height);
queue.enqueueReadBuffer(image_buffer, CL_TRUE, 0,
                        bufferSize, image.data());

int32_t count = 0;
for (int32_t i = 0; i < height*width; i++) {
    count += image[i] < 255;
}
printf("%d\n", count);
}

```

Listing C.6: The *Mandelbrot (OpenCL)* test program.

```

#define MAX_ITER 255

float4 createStartX(int x)
{
    return (float4)(x, x + 1, x + 2, x + 3);
}

__kernel void mandelbrot(__global uchar* restrict output,
                        const int width, const int height) {
    int x = get_global_id(0) * 4;
    int y = get_global_id(1);

    float4 initialReal = -2 + (createStartX(x) / (float)width * 2.5f);
    float4 initialImaginary = -1 + (y / (float)height * 2);

    float4 real = initialReal;
    float4 imaginary = initialImaginary;

    int4 iterationsPerPixel = (int4)(0, 0, 0, 0);
    int iterations = 0;
    int4 mask;

    do {
        iterations++;
        if (iterations > MAX_ITER) break;

        float4 oldReal = real;
        real = real * real - imaginary * imaginary + initialReal;
        imaginary = 2 * oldReal * imaginary + initialImaginary;

        float4 absoluteValue = real * real + imaginary * imaginary;
        mask = islessequal(absoluteValue, (float4) 4.0f);
        iterationsPerPixel -= mask;
    } while(any(mask));

    vstore4(convert_uchar4(iterationsPerPixel), 0,
            output + x + y * width);
}

```

Listing C.7: The kernel used with the *Mandelbrot (OpenCL)* test program.

Appendix D

Additional Data

This appendix presents additional measurement data not presented in Chapter 5. Tables D.1 to D.4 show mean, standard deviation and C_v for all experiments performed as part of this thesis.

Table D.1: Sample mean time, sample standard deviation and C_v (lower is better) for Hello World, Sort and Sort w/RandInt.

Experiment	Hello World			Sort			Sort w/RandInt		
	mean	stddev	cv	mean	stddev	cv	mean	stddev	cv
	s	s	%	s	s	%	s	s	%
Baseline	0.090	0.003	3.269	39.332	0.876	2.226			
Remove Fail2ban	0.090	0.004	4.585	39.069	0.834	2.134			
Performance governor	0.071	0.004	5.313	38.236	0.661	1.728			
Performance governor + C++	0.079	0.004	4.767	38.690	2.239	5.786			
Performance governor + C++ + chroot	0.020	0.000	0.000	37.016	1.341	3.622	8.159	0.006	0.074
Performance governor + C++ + chroot + nice	0.080	0.002	2.140	35.371	0.187	0.529			
Performance governor + C++ + chroot + taskset	0.020	0.001	4.950	35.242	0.358	1.017			
Performance governor + C++ + chroot + taskset + nice	0.080	0.003	3.675	35.240	0.286	0.811			

Table D.2: Sample mean time, sample standard deviation and C_v (lower is better) for the Mandelbrot programs.

Experiment	Mandelbrot			Mandelbrot (OpenMP)			Mandelbrot (OpenCL)		
	mean	stddev	cv	mean	stddev	cv	mean	stddev	cv
	s	s	%	s	s	%	s	s	%
Baseline	45.213	0.061	0.134	82.873	0.370	0.447	37.853	1.047	2.765
Remove Fail2ban	45.212	0.057	0.126	82.597	0.442	0.535	37.892	0.855	2.258
Performance governor	45.127	0.044	0.098	82.648	0.388	0.470	36.392	0.046	0.125
Performance governor + C++	44.832	0.013	0.028	76.148	0.324	0.425	36.020	0.026	0.073
Performance governor + C++ + chroot	44.774	0.013	0.029	76.288	0.527	0.690	35.964	0.035	0.097
Performance governor + C++ + chroot + nice	44.839	0.013	0.029	76.749	0.485	0.632	35.958	0.034	0.096
Performance governor + C++ + chroot + taskset	44.764	0.005	0.012				35.913	0.031	0.085
Performance governor + C++ + chroot + taskset + nice	44.815	0.006	0.012				36.015	0.021	0.060

Table D.3: Sample mean energy consumption, sample standard deviation and C_v (lower is better) for Hello World, Sort and Sort w/`RandInt`.

Experiment	Hello World			Sort			Sort w/ <code>RandInt</code>		
	mean	stddev	cv	mean	stddev	cv	mean	stddev	cv
	J	J	%	J	J	%	J	J	%
Baseline	0.233	0.105	44.892	128.002	4.927	3.849			
Remove <code>Fail2ban</code>	0.237	0.105	44.225	128.298	5.124	3.994			
Performance governor	0.108	0.029	26.843	127.612	4.489	3.518			
Performance governor + C++	0.120	0.058	48.324	109.632	4.004	3.652			
Performance governor + C++ + <code>chroot</code>	0.017	0.006	36.039	106.458	3.590	3.372	20.478	0.179	0.875
Performance governor + C++ + <code>chroot</code> + nice	0.180	0.016	8.776	100.866	0.620	0.615			
Performance governor + C++ + <code>chroot</code> + <code>taskset</code>	0.024	0.009	35.054	101.669	2.929	2.881			
Performance governor + C++ + <code>chroot</code> + <code>taskset</code> + nice	0.154	0.024	15.306	102.370	3.509	3.428			

Table D.4: Sample mean energy consumption, sample standard deviation and C_v (lower is better) for the Mandelbrot programs.

Experiment	Mandelbrot			Mandelbrot (OpenMP)			Mandelbrot (OpenCL)		
	mean	stddev	cv	mean	stddev	cv	mean	stddev	cv
	J	J	%	J	J	%	J	J	%
Baseline	107.208	4.688	4.373	159.575	4.591	2.877	83.547	2.712	3.247
Remove <code>Fail2ban</code>	106.624	4.788	4.491	162.948	4.593	2.819	84.023	2.817	3.353
Performance governor	106.733	1.872	1.754	157.436	4.484	2.848	114.471	1.306	1.141
Performance governor + C++	83.863	0.409	0.488	339.712	1.621	0.477	96.520	0.370	0.383
Performance governor + C++ + <code>chroot</code>	83.716	1.016	1.214	336.922	2.676	0.794	96.264	0.397	0.412
Performance governor + C++ + <code>chroot</code> + nice	83.700	0.468	0.559	334.318	2.924	0.875	95.952	0.346	0.360
Performance governor + C++ + <code>chroot</code> + <code>taskset</code>	83.257	0.472	0.567				95.827	0.432	0.451
Performance governor + C++ + <code>chroot</code> + <code>taskset</code> + nice	83.152	0.371	0.446				96.044	0.342	0.357

Appendix **E**

Digital Appendix

The digital appendix included contains the *cmb-board* (back end), *cmb-flask* (server) and *climbing-mont-blanc* (front end) repositories.