Analysis of Energy Efficient Compilation
Comparison of Machine Learning techniques for energy efficiency in compiler optimisations

Munaaf Ghumran

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

May 2014
Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Munaaf Ghumran, May 2014
Executive Summary

In this dissertation I am to provide an insight into which machine learning algorithms allow us to learn how to choose compiler optimisations that reduce energy use. The work aims to further research done in 2009 by the MILEPOST project on a similar idea, but for execution time, and see if the same idea can also be applied to energy use. There are strong industrial links with a project titled MAGEEC who hope to base their implementation design on the outcome of this research.

Current research is lacking in targeting energy as most work has always focused on simply improving execution time or code size. I hope to provide some idea of how current optimisation levels are suboptimal and that there often are better optimisations out there, particularly to reduce energy usage. This shows that there is an open market for developing a framework that can use machine learning to predict better optimisation flags than current state of the art passes do, such as GCC and the -O3 optimisation level.

I also explain the methods in which this work can be further worked on as a research proposal, with multiple avenues that can lead on from what has been found.

Summary of Achievements

- Creating an open source benchmark data suite of 84 programs from a variety of sources due to the lack of any current freely available open source suites
- I spent over 250 hours collecting data comprising of 500 unique compilation passes on 84 different programs
- A further 40 hours of deductive analysis on this data to bring it to a machine learning state
- Created an automated framework in Python that synchronises data over a network to allow minimal lag in starting the energy monitor readings
- I implemented a representation of this data that was then used by WEKA to create machine learning models
- Used eight machine learning algorithms to train and test on, storing the accuracy and rate of false negatives in each one along with time taken to build the models and predict classes for the test cases
- Analysed and visualised the final results to compare and conclude with reasoning which algorithms performed better
Supporting Technologies

The following list highlights any third party resources that I have used during my project:

- I used the WEKA framework to support the choices of machine learning algorithms and implementations under Java. \(^1\)
- I used a Pandaboard supplied by the Microelectronics Department to record the training data.
- I have used three STMF24-Discovery boards supplied by the Microelectronics Department to record the energy readings.
- I used and adapted a Python framework developed by PhD student James Pallister to record energy data from these boards.
- I made use of the Python Socket library when working over a network. \(^2\)
- I have used the Eclipse IDE to work with when implementing under Java.

\(^1\)WEKA: [http://www.cs.waikato.ac.nz/ml/weka/](http://www.cs.waikato.ac.nz/ml/weka/)

\(^2\)Python Socket: [https://docs.python.org/2/library/socket.html](https://docs.python.org/2/library/socket.html)
## Notation and Acronyms

The following list details any meanings of any notations or acronyms that I have used in my dissertation:

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MILEPOST</td>
<td>MachIne Learning for Embedded PrOgramS opTimizations</td>
</tr>
<tr>
<td>MAGEEC</td>
<td>MAchine Guided Energy Efficient Compilation</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>BEEBS</td>
<td>Bristol/Embecosm Embedded Energy Benchmark</td>
</tr>
<tr>
<td>-O3</td>
<td>Optimisation level in GCC</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low Level Virtual Machine</td>
</tr>
<tr>
<td>NN</td>
<td>Nearest Neighbour</td>
</tr>
<tr>
<td>WEKA</td>
<td>Waikato Environment for Knowledge Analysis</td>
</tr>
<tr>
<td>ARFF</td>
<td>Attribute Relational File Format</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal Component Analysis</td>
</tr>
<tr>
<td>SMO</td>
<td>Sequential Minimal Optimisation</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Machine</td>
</tr>
<tr>
<td>J48</td>
<td>Decision Tree Algorithm</td>
</tr>
<tr>
<td>IID</td>
<td>Independant and Identically Distributed</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>PhD</td>
<td>Doctor of Philosophy</td>
</tr>
<tr>
<td>IPV6</td>
<td>Internet Protocol Version 6</td>
</tr>
<tr>
<td>DDR2RAM</td>
<td>Double Data Rate Random-Access Memory</td>
</tr>
<tr>
<td>CO2</td>
<td>Carbon Dioxide</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma Separated Values</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank my friends and family who have supported me to get to where I am and without them I would be nobody. Getting through this degree required more than academic support, and for all those who I have met over the past four years, I express my gratitude for allowing me to spend time in their presence.

Finally a thank you to my brother for his endless support when I have needed him, my sisters for always being there to cheer me up and my mother and father for having brought me up to be here and sticking with me through the ups and downs.
## Contents

1 Contextual Background 1
   1.1 Background ................................................................. 1
   1.2 Motivation ................................................................. 2
   1.3 Central Challenges ....................................................... 3
   1.4 Aims and Objectives ..................................................... 4

2 Technical Background 5
   2.1 Explanation of Problem .................................................. 5
   2.2 Related Work ............................................................. 5
   2.3 Supporting Technologies ................................................ 8
   2.4 Approach ................................................................. 9

3 Project Execution 11
   3.1 Design of hardware ...................................................... 11
   3.2 Collection of Data ....................................................... 13
   3.3 A Priori Analysis of Data .............................................. 17
   3.4 Machine Learning Stage ............................................... 19
   3.5 A Posteriori Analysis of Results ..................................... 21

4 Conclusion 27
   4.1 Summary of Achievements ............................................. 27
   4.2 Project Status .......................................................... 27
   4.3 Future Plans ............................................................. 28

A MILEPOST Feature Vector 31
List of Figures

2.1 MILEPOST Framework .............................................. 7
2.2 Iterative Compilation Interface Framework .......................... 7

3.1 STM32F4-Discovery Boards and Energy Monitor attached ........... 11
3.2 Pandaboard .................................................................. 12
3.3 Small sample of features used in MILEPOST feature vector .......... 13
3.4 Physical Setup ................................................................ 14
3.5 CSV Output of Energy Monitor ......................................... 15
3.6 Example of an ARFF file type .......................................... 18
3.7 Results of decision tree algorithms .................................... 21
3.8 Results of lazy learner algorithms ..................................... 23
3.9 Results of SMO algorithm .............................................. 24
3.10 Results of all algorithms tested ....................................... 25
Chapter 1

Contextual Background

1.1 Background

A project called MILEPOST [4] ran in 2009 which worked on automating the choice of compiler optimisations in an effort to see if they could outperform the standard optimisation flags that exist in compilers today. Their work included the creation of a feature vector which summarises a given program into a numerical representation based on 56 features, as well as machine learning models that conclusively beat the standardised optimisation level -O3 by over 10%. This work however, focused on optimisations for code speed/code size or execution time, no options existed to optimise for energy efficiency.

While one can argue correctly that there exists a natural correlation between a code size and energy used, this link is not always clear. Sure, making a program execute 100 less instructions might in general mean its working less hard, for a shorter amount of time and therefore using less energy - however there remain intricacies between which parts of a processor are in use as different parts utilise more energy than others. Work concluded from Valluri [20] showed an example of this in the case of loop unrolling. While an optimisation that itself reduced execution time, it also increased energy dissipation leading to the conclusion that the answer isn’t simply clear cut.

My dissertation work has close ties with MAGEEC, a new UK-based project1[3]. I worked with the company running this (Embecosm2) in conjunction with the University of Bristol over the summer of 2013 as an intern. Their goal is to complete an infrastructure with an automatically chosen set of compiler flags using machine learning for a given program to reduce energy use, which mimics a similar framework such as the one MILEPOST created but for energy. My work will aim to aid in the finding of a direction for the machine learning implementation.

The bottleneck over summer was a lack of open source programs to be used in the analysis stage. Although numerous proprietary solutions exist, such as CoreMark3 and SPEC4, the project requires all the work used to be licensed freely. The fact that an open source solution isn’t out there readily available has led to this being a part aim of my project as I would be required to gather a set of programs at the start of the project to collect energy data from.

There needs to be work done in collating many open source programs from what is out there and having one set that can be used to work off. The summer work showed that this itself is a time consuming task and would be a fundamental part of the final project itself.

1MAchine Guided Energy Efficient Compilation: www.mageec.org
2Embecosm: http://www.embecosm.com/
3CoreMark: https://www.eembc.org/coremark/
4SPEC - Standard Performance Evaluation Corporation
1.2 Motivation

The motivation for this work comes from the fact that there hasn’t been any known research aimed specifically at energy efficient compilation to date. Previous work, as mentioned focused heavily on code speed and code size optimisations which are the staple of most current optimisation flags created in compilers. There is a chance that work could be ongoing in this area currently by others, which motivates us to be the first to release our findings.

There is so much information that is still unknown on this topic, and I believe the many more questions will come out of this than are answered. An example of this is the fact that there are so many optimisation flags these days, yet so little is known about how they interact. Flag ordering itself can affect the outcome of a series of optimisations, for example an ordering of A-B-C may result in an energy reduction of 5% but the same flags in order B-C-A may only reduce energy by 2%. Answers to questions such as these are perfect examples of where future research could go on from my work. This is because knowing more about those interactions would lead to changes in the fundamental learning that my machine learning would use, a machine learning algorithm that could incorporate an understanding of the ordering of flags would be the next stage of a system that automatically chooses better flags. Currently this is too complex to be able to to implement due to the sheer number of flags and possible combinations that exists. Having answers that can help narrow this search space down means we could be able to find better search space solutions as opposed to sampling a small space randomly which is the approach taken and explained later in my profiling section.

Furthermore, more motivation comes from the fact we are living in an age where there is now a demand to be energy efficient. We are no longer in a time when this work would be seen as an extra, instead, it is now needed. With so much focus constantly on the depleting energy resources around the world and consumer driven focus on energy efficient solutions on everyday items, companies are now being required to work towards this goal. So this is a huge part of everyday life in most people - being able to target and improve an area that is in such wide and constant use (applications and computing) is enough motivation to make an impact on a large scale.

With the open source nature of GCC, further motivation comes from the fact that future work could lead to a global framework implemented in the next GCC upgrade, meaning every user could utilise an automated energy efficient flag that targets energy reduction of a given program. This leads to a potential global reduction in energy use in places where compilation happens simultaneously on such large scales such as data centres. Imagine every data centre in the world now compiling programs and running them targeting energy reduction, the scope of reduction in CO2 emissions from this alone would be in the factor of tens of metric tonnes.

Finally one of the direct motivations of this project’s outcome is that when the MAGEEC project completes and the framework is all automated, there is a direct benefit to embedded devices such as mobile phones and anything that runs off a battery. This will lead to potentially improved battery life in those types of embedded systems which is beneficial in a variety of situations.

---

5There exist over 100 unique optimisation flags in GCC, giving a full space of $2^{100}$ possible combinations (turned on/off) and 100! unique orders between this space

6Based on reported 170Mt from US data centres of energy use [2]
1.3  CENTRAL CHALLENGES

1.3.1  Dataset

The first challenge that has been identified is the sourcing of the open source programs and creation of this data set that will be used to collect energy data from. There are a variety of old systems and small out of date suites such as Dhrystone, Whetstone, Linpack and LAPACK. But these suites consist of either specifically targeted programs, or very few programs in general. Work from the MILEPOST team included creating and releasing their own data set titled CBench\textsuperscript{7} which includes 32 programs that were derived based on the popular MiBench\textsuperscript{8} open source suite.

However, from a machine learning point of view, 32 programs is not really enough to gauge a wide variety of information that will be useful to learn off of. So our aim was to take some of these benchmarks from CBench, and a variety of other benchmarking suites to create our own one. Work on this falls under the title BEEBS\textsuperscript{9} and some work has been done by James Pallister on maintaining this set, with further programs sourced by myself. Between the both of us, we aimed to have a set of around 80-100 programs, expanding from the 10 that BEEBS began with. This number of programs would be closer to what we would need to have a wide enough representation covered for the machine learning.

1.3.2  Data collection phase

This challenge is about making sure that the data collected is reliable, correct and in a usable format. Compilation and data collection of these programs is arguably going to be one of the longest parts of this project. With the size of the data set and the varying time each program takes to compile just once, scaled up to the number of unique compilations to sample the flag space, the collection phase has the potential to take nearly two weeks alone.

As I haven’t worked on electronics or the hardware that monitors energy usage prior to this project, I will be seeking some advice and help from the creator of the framework and PhD student James Pallister on this. He will be able to guide me into getting the framework set up correctly and then my work will continue by adapting his profiling code to suit the topic at need. The energy board hasn’t been used on a Pandaboard before this, so it isn’t confirmed whether or not it will work correctly or reliably, but it is believed to be suitable for this board and should in theory provide us with correct results.

Lastly one needs to make sure that all the information collected from this is stored in such a way that it is usable. The data needs to contain the optimisation flags that were used in each specific run, the corresponding energy use and peak energy use as well as the other information that is gained from the profiler (voltage, current etc). This needs to be stored in a CSV format and structure that it can be easily analysed after to go into the machine learning representation.

1.3.3  Machine Learning Representation

The next challenge to this project is faced in the actual machine learning. Here, the representation and analysis of energy data collected needs to be defined early on. As mentioned above, once we know the CSV layout of the results, they have to be analysed (partially manually) to determine a subset of flags which are deemed to be most influential for each program.

This concept will be explained further in the execution part, but the challenge here is ensuring that we are able to have a program, and corresponding flags that have been determined to be most influential or ‘good’ given the results from the data collection phase. This is the subset data that will then be used in the representation. So a program with its known ‘good’ flags along with the feature vector information that describes the program (from MILEPOST work) aim to become the full representation the machine learning will learn from.

\textsuperscript{7}CBench: http://ctuning.org/wiki/index.php?title=CTools:CBench
\textsuperscript{8}MiBench: http://www.eecs.umich.edu/mibench/
\textsuperscript{9}Bristol/Embecosm Embedded Energy Benchmark Suite: http://www.cs.bris.ac.uk/Research/Micro/beebs.jsp
CHAPTER 1. CONTEXTUAL BACKGROUND

1.4 Aims and Objectives

The high-level objective of this project is to aid a direction of machine learning via analysis of several algorithms to lead to energy efficient compilation. More specifically the concrete aims are:

1. Investigate suitability of Pandaboard as a system to test on
2. Run and collect energy readings under 500 compilation settings for each program on the Pandaboard with energy measurement boards attached
3. Utilise WEKA framework as an input for the machine learning representation to utilise multiple algorithms
4. Have a data set of many different programs along with the energy readings of each for archive use
5. Compare and analyse results of various algorithms to see which perform well on this problem
Chapter 2

Technical Background

I will talk about previous work that has been done in this field as well as explaining the tools and third party resources that are used. This is to ensure that everything mentioned hereafter is self contained in a way that little to no external reading is required to understand the motivations and aims behind this project and decisions made during it.

2.1 Explanation of Problem

The project aims to compare existing machine learning algorithms on energy information for multiple programs under many different compilation settings in order to (in the future) implement an effective machine learning algorithm to automatically predict flags to reduce energy use in a given program. Energy use in programs is something that’s been under studied over the years despite it’s rising importance with the high energy demands of the world these days. The problem in this project lies in gathering this data and representing it as well as then comparing the results that are obtained in order to reason why one algorithm may have performed better than another. This explains what I want to achieve, and I follow on by looking at what work has been done around this area.

2.2 Related Work

2.2.1 Compilation work

I begin by looking into what previous work has been done around this area. As I have mentioned, this is a relatively under-researched area, with little done in terms of similar types of projects. There has been work done around the area, for example not just looking into using machine learning to predict compiler flags, rather just looking into energy use under certain compilations and identifying if there are any patterns in these results. For example, there is work that looks at the link between compiling for performance and for energy by Valluri[20], techniques for low energy use from Tiwari [18] to the start of automated techniques from Gheorghita et al [5] and Patyk et al. [15].

One can start by looking at the works that initially targeted execution time [6, 9], and the later research that tries to link to link in execution time to a reduction in energy use [17, 7, 20]. These studies aim to show that compiling for improvements in execution time also lead to a reduction in energy use. This can be thought of in a simpler way, if the code segment A is being optimised to perform fewer instructions for example, naturally this will lead to less work put on the processor and memory management areas of the CPU, leading to a reduction in energy use, after all energy = power * time, so reducing the time element should reduce energy use. Of course there are cases where this is not so simple, where reducing the instructions executed and time taken may lead to parts of the system being used that are more power hungry, for example more memory operations may take place, or under a less efficient part of the processor itself, increasing power used despite reducing time - hence actually increasing energy used.
All of the works mentioned look at optimisation levels as a whole, without looking individually at single optimisations, and the effect of each flag in essence. This highlights an area where there’s been a lack of research in itself. Some work has been done on a few of the more common flags, but on the whole, this area remains insufficiently researched.

2.2.2 Energy Optimising

Out of the work that has looked specifically at the energy optimisations under compilers, one of the most effective optimisations turned out to be the scheduling of instructions. This work is documented in Parikh et al. [14] who find that reducing inter-instructional energy cost by the scheduling of instructions specifically reduces the energy consumption.

Another key optimisation is in the form of reducing memory access, that works by owing less in the form of access paths from CPU to memory, and rather having more in CPU local memory such as the cache reducing the changes in hardware that are required to be accessed.

2.2.3 Machine Learning methods

Some iterative learning methods have been experimented with for compiler optimisation selection. Gheorghita [5] and Patyk [15] both did work using an iterative compilation method. A drawback to this method is that it requires multiple compilations of course, increasing the compilation time. However, it proved effective in reducing both energy use as well as an increase in performance, so remains an effective method.

Lin et al. [9] have a method that used genetic algorithms to automate optimisation choices. This is a method that hasn’t been tried often in this area, and I believe Lin’s work to be the only known published genetic algorithm work in this area. While other work exists in this area using genetic algorithms, none other target only energy optimisations.

2.2.4 MILEPOST

MILEPOST [4] is the most similar work that exists using a machine learning approach on optimising compiler flags. A simple explanation of how this works is a step by step as follows:

- Store information about many programs run under multiple compilation settings
- Extract program features which numerically represent a given program
- Use the machine learning framework to best pick optimisations based on that previous information learnt

The benefit to this is that it requires just the source code, as the intermediate representation works off the source code. That is, the feature extractor is completely dependent on the source code itself. With the use of iterative compilation in the training phase it is able to generate a large database to then compare new programs against. Figure 2.1 and 2.2 visually shows how this works from the paper.
2.2. RELATED WORK

Figure 2.1: MILEPOST Framework

Figure 2.2: Iterative Compilation Interface Framework

Feature Extractor

The feature extractor is part of the work developed by the MILEPOST team. It takes a program and analyses it under some Datalog\(^1\) rules to numerically represent a program based on the features that are present. The full list of features is shown in Appendix A, but an example of the types of things that are matched are the number of instructions executed, number of unconditional branches in methods and number of calls that return an integer. The origin of these lists of features and their reasoning remains unknown from the published work, but it has been continually updated and expanded and is reputable given the popularity of the research work completed by the MILEPOST team.

\(^1\)Datalog is a logic based programming language
CHAPTER 2. TECHNICAL BACKGROUND

Machine Learning

MILEPOST combine two methods of learning. One uses the notion of distribution over good solutions and the other uses an Independent and Identically Distributed model that uses the k-nearest neighbour algorithm to compare the unseen program to the training set in terms of the feature space. The feature space they use compromises of a training program $T$, its optimisation passes $x$ and execution times $y$. This is going to be the same type of feature space I will aim to use, with energy use replacing execution times. Their prediction model then attempts to find a solution that maximises the result of searching over the solution with an unseen program. The good thing about their system is that the database is collectively updated, and these new data sets that are updated automatically are used by the prediction model, so their model doesn’t need to change even if the training set increases.

2.3 Supporting Technologies

2.3.1 WEKA

WEKA$^2$ is an open source collection of machine learning algorithms. It has the ability to do pre-processing, classification and allows visualisation of clustering algorithms amongst other features. The reasons I have opted to use this third party resource as part of my project are as follows: its framework is well documented and, it is a popular choice amongst the keenest of machine learners. Also, its extensive choice of algorithms gives me access to a wide set of machine learning methods all under one framework. I have used WEKA a few times before, so it is beneficial that I am not completely new to it, and this experience will be helpful when it comes to applying algorithms to the training data and learning various models to compare.

ARFF

The library has it’s own specific format, the ARFF data type, which it uses to keep data in one form and uses the relational information contained in this format to allow multiple algorithms to execute on the same data set. ARFF stands for Attribute-Relation File Format and is the representation all inputs are standardised to when using WEKA. This is the standard format that my machine learning representation will be in the form of. This will allow me to gain access to the wide amount of algorithms in WEKAs source to compare the effectiveness of chosen algorithms against each other.

2.3.2 Python

I will be programming partly in Python during the project. Python has a brilliant library for communicating over networks, and I will be using this sockets library to manage information between the hardware running code and my laptop that controls when the energy monitor begins running. It is also going to be used to automate the entire data collection process. The code to use the energy monitors has already been coded in Python, so I will take this code and modify it to perform the variation in task I require.

2.3.3 Java

Java is my chosen programming language to process the machine learning data and apply algorithms onto the data. I will be using the Eclipse IDE to work in Java to utilise its advantages. This Java work will consist of a class that automatically loops through all the data to be learnt, and creates models accordingly. This is in a way such that each model represents a single flag, and the algorithm determines if for a given program, that flag should be turned on, or off.

2.3.4 Energy Measurement Board

This board is a uniquely created board that is able to take up to three simultaneous measurements, allowing up to three parts of a target board to be measured. This is useful if you want to get readings from more than just the CPU cores, however for my project I focus on just using one of these

measurement points. This is the point that connects onto the Pandaboard and reads in information on the following measurement points:

<table>
<thead>
<tr>
<th>Measurement Point</th>
<th>Inductor</th>
<th>Power Rail</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP1</td>
<td>L19</td>
<td>VDD_VCORE2</td>
</tr>
<tr>
<td>MP2</td>
<td>L20</td>
<td>VDD_VMEM</td>
</tr>
<tr>
<td>MP3</td>
<td>L17</td>
<td>VDD_VCORE1</td>
</tr>
</tbody>
</table>

The energy readings I will be collecting will come from these three measurement points on one single measurement board. The reason I have chosen these measuring points is that MP3 uses the power rail VDD_VCORE1, which consists of the actual Cortex-A9 core chip [13], and MP2 uses the rail VDD_VMEM meaning that energy readings are coming from the memory on the board too. I feel like these are probably the two parts of the board that we will most likely be using when we execute a program, so we’d be aiming to monitor for changes in and around those areas during execution.

There were two other boards that also fed onto other parts of the Pandaboard, however these power rails did not really target parts of the board that I felt were necessary to monitor for changes, such as the audio chip and further memory.

The energy measurement board is capable of measuring at 2,000,000 samples/second [11] and provide information on energy used, time taken, current and voltage. The boards come with a firmware developed for them that will allow this to work under Linux immediately. I will be flashing those firmwares onto the board myself prior to using them.

2.4 Approach

Out of all of the related work that we have seen, the MILEPOST project is the most relevant and similar to what they have done for execution time. As this is a known successful project, for this stage and project we will look to utilise the work they have done and attempt to compare algorithms for energy use instead of execution time. The feature extractor that was created as part of MILEPOST is going to be used as part of this projects representation. The feature vector will represent a program, so that the source code isn’t required, as it acts as an intermediate representation that summarises the program numerically instead. From this we hope that these features, combined with pre determined knowledge from multiple compilation runs of each program, acting as prior information will aid in creating good machine learning models to test unseen programs on.

I will begin by sourcing the programs to run on the Pandaboard. This will be a time consuming task as I will only use open source programs, and will aim to get a wide variety in types of programs, so that they do not all perform similar functionality. Once this data set has been created, I aim to take these programs and execute them each under 500 different randomly generated optimisation flags, such that a sample space of the flag space is modelled. Using all of the energy readings for the programs under these different flags, I plan on using some analysis over this data to pick out flags that seem to appear commonly in the top solutions as well as bad flags that appear overly commonly in the worst solutions. This information will form the machine learning representation that I aim to then compare multiple algorithms against and create many models as well as provide test cases to see how well the predictions are.
Chapter 3

Project Execution

3.1 Design of hardware

3.1.1 Energy Monitor

The beginning of the project required interacting with the Pandaboard and the energy boards to instrument the device and be able to collect energy information while each program was executed. Recent work by Dr. Simon Hollis in the micro electronics lab led to an energy measurement board that could instrument and relay information about a board it was connected to. This consists of an STM32F4-Discovery board (Fig 3.1) and an attachment board that does the profiling which is built in such a way to require no soldering and simply plugs on to the discovery board.

Next, a custom made energy measurement board created by Dr. Simon Hollis is attached on top of the STM32F4-Discovery board to allow the hardware to interact and measure the device it will be connected to. This board is designed specifically to provide power sensing for ARM Cortex M4F hardware systems[10]. Figure 3.1 shows what these two components look like.

(a) STM32F4-Discovery Board  (b) Board with Energy Monitor

Figure 3.1: STM32F4-Discovery Boards and Energy Monitor attached
### 3.1.2 Pandaboard

The hardware that is chosen to best represent embedded systems is the Pandaboard. There are a variety of reasons this board was chosen for this project. The board choice is important in the overall result of the project, so getting this right was important. There needed to be a board that represented an embedded system quite well with stripped functionality and simple input/output. While a fully embedded system that had zero operating system functionality was preferred, the influence for the final decision came from limitations on the data set.

Finding a data set with full C support (such as printf) was hard enough as described, whereas having to find benchmark programs that would fit on a board with smaller memory capacity, and having to rewrite/structure each benchmark to remove additional dependencies (such as `#include<stdio.h>`) would be nearly impossible.

My beginning research led me to the conclusion that there would not be enough benchmarks available to run on a board with smaller memory or CPU. I did begin work and research at the start to see if this was feasible, before changing my decision to the Pandaboard. The added benefit of extra memory and access to system resources of a full operating system (Ubuntu) was a trade off that was acceptable to make. The Pandaboard ES Rev B model runs under a Cortex-A9 1.2GHz Processor and 1GB DDR2RAM. This produces a powerful board capable of running every benchmark that a normal laptop could run, meaning there was a larger availability of transferring the found benchmarks from standard laptop use to the board.

The STM discovery boards are then connected to the Pandaboard by several wires that touch end points on the Pandaboard. This exact configuration was aided by PhD student James Pallister who has a higher depth of knowledge in micro electronics than myself.
3.2 Collection of Data

3.2.1 Benchmarks

I began creating the benchmark suite by researching available suites that exist already. There are a few commonly known suites such as the Worst Case Execution Time (WCET) \(^1\) and the popular LLVM Test Suite \(^2\). After many hours of searching for more alternatives, I came across the following:

- Whetstone
- Drhystone
- StateChart
- Public Domain Code
- Shootout
- Sieve of Eratosthenes
- Stanford

There exist plenty of industrial propriety test suites such as CoreMark and SPEC which can cost upwards of $2000+ to purchase benchmark programs. The alternative is the open source market, which remains quite wide and scattered, but serves as a fine alternative to source programs. Collating programs from these various open source suites allows the creation of a specialised data set that can be associated with the project and possibly useful for external parties looking for anything similar. The trade off in the time that it takes to locate, verify and ensure each chosen program is suitable was one I decided to make for obvious cost reasons, but more so for the end goal of having this set available for others after.

All of the programs were due to be run under the Ubuntu distribution of the Linux operating system, so had to be non platform dependant programs. Generally, this was the case as I was only searching for C program files and not any other format due to its popularity and suitability being small, compact and low level, the perfect combination to run on a board such as the Pandaboard. C is still a very popular programming language, and as many of the existing suites were created 20 or 30 years ago either directly as a C implementation then, or have been converted to this format over the years.

Another key point to make sure was filled was that these benchmarks covered a wide range of applications, i.e. from memory or CPU intensive programs to highly mathematical or loop intensive programs. The reason for this is to create a data set that will have a wide set of Feature Vector results when the MILEPOST feature extractor is executed as part of the Machine Learning representation later on. As this feature vector is essentially a list of 56 features (See Appendix A for full list) that range from the number of basic blocks in methods, number of instructions to number of edges in the control flow graph, from a machine learning point of view, it is important to try to supply the learning phase with feature vectors that cover and represent each of these features, with a wide variety on the frequency of each result. A total of 84 programs have been collected and used across these open source packages. A link is in the appendix to view and download these.

---

\(^1\)Created by the Mlrdalen research group - See http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

\(^2\)LLVM - Low Level Virtual Machine is the second largest compiler in use
3.2.2 Compilation

Python Framework

After generating the data set of 84 programs above, the next stage was to run these programs on the Pandaboard and collect the energy readings for each program under various compiler flag settings.

The setup of the Pandaboard, energy monitors and my laptop is shown below. The three STM boards are all connected directly onto the Pandaboard, and the connection runs back to the laptop, which acts as an external source that can control the timing of starting and stopping the energy monitor. An alternative method is to press a blue button on the board which also starts/stops the energy reading, but this method would factor in human error and reaction time error in a scenario where the data we are trying to capture is so minuscule that the error will often be longer than the time a program takes to execute. For this reason, naturally I opted to use the code based solution that can be triggered remotely.

![Physical Setup](image)

The key information to take from the setup is that there is no link between the laptop, and the Pandaboard. This provides an issue in the sense that while the energy monitor can be remotely started and stopped via the laptop, there remains no way to link between a program running on the Pandaboard, and this remote code executing at the same time on the laptop due to the separate nature of the systems. Because of this, I needed to think of a solution that would allow me to somehow make these two events occur simultaneously.

There were a couple of solutions to this. One that was looked into was using the MQTT [19] protocol via Ethernet. This would work by sending messages over the two systems to instruct when it was suitable for execution to occur. This library is available under a variety of languages, so I could have implemented it in Python, C++ or Java for example. Working with this, I encountered a few issues under the Python language, which led me to look for another solution. I had issues connecting through my home network under the machines which were both on the network using IPV6. These problems lead me to look for another solution, instead over Wifi, and the next one that seemed viable was the Python Socket library.

This library works relatively simply, it was fairly straightforward to implement a server and a client that could connect to each other and transfer messages. I needed to be able to communicate with the
3.2. COLLECTION OF DATA

client (Pandaboard) and then initiate the energy monitor to begin monitoring when the client sent a message indicating the program was about to execute. However, the time delay in receiving messages and then executing the monitoring code was too high using sequential code, which is a problem in itself still. I wanted to try multithreading the python server code to see if that could minimise the lag, so that we could queue messages on one thread as they are continually received, and on another thread the monitoring would begin and save. This minimised the lag considerably, and was the best result I could hope for. Now I had the ability to communicate between the two systems, and instruct the monitoring to begin when the program began executing on the Pandaboard.

Finally the last thing I did to reduce the effect this small delay had on the results was that for each program that was executed, it would be executed 20 times, and the energy board would record during all of these executions, and then stop and average the final result to get an average reading for one execution. I think this is an acceptable approach, particularly in small programs which run in a fraction of a second.

All of this work was done under version control using Git, due to the constant changes in code and slight variations I felt the need for version control here was paramount.

Approach for optimisation settings

There have been several methods used to sample the optimisation space. GCC has 82 optimisation flags in the -O3 optimisation level itself [1], leading to an unattainable $2^{82}$ possible configurations if explored fully. Research conducted by Dr. Simon Hollis and James Pallister [8] used a Fractional Factorial Design to explore this space in a much shorter amount of compilations. Given that our target is to analyse the results of this space, and not fully explore or attempt to explore a huge section of it, the approach used in this project seems simple. It seems simple because it is just taking 500 random combinations of the 82 flags used in the -O3 level of GCC’s optimisations, and executing each program under random flag settings. This approach is fine because so many of these `random` solutions actually result in better solutions than -O3 by itself. -O3 is a generic optimisation, so for each program its often suboptimal as an optimisation parameter. An example of this is that one of the programs run under 500 random flag settings outperformed -O3 on 88 of the 500 compilations, which is 17.6% of all solutions.

A limitation to bear in mind is that this needs to run on just the Pandaboard, and that some programs might not take long to run, but some take over a minute each. Scaling this up shows that for the large programs that take perhaps two minutes to execute each time, collecting the energy readings under 500 combinations leads to roughly 1000 minutes of execution, the equivalent of a tad under 17 hours. This puts huge strain on the time that is needed to do this phase of the project. If there are just ten programs of the 84 that last this long, that is nearly 10 days of just collecting the data. This limitation is a large reason why I could not search a larger space than 500 combinations.

This was implemented in the clients Python code of the framework mentioned above. To summarise, it works by iterating through a directory of programs, then generating a random combination of 82 flags turned on, compiling under these flags, executing the program, and repeating this 500 times with a new set of flag combinations. The server side then collects energy readings for each compilation, stores this information and when each program concludes writes it out in a CSV style format that allows the data to be easily imported and worked on after. An example of the output is shown below where 1 represents a flag turned on, and 0 a flag turned off, then the energy readings beside this.

Figure 3.5: CSV Output of Energy Monitor
Each program had its own file containing all 500 flat settings and corresponding energy readings, which leads me onto the next stage where some a priori analysis needs to be done on all of this data before being used in the machine learning stage later on.

This part of collecting data took nearly 7 days of non stop compilation running and saving recorded information about the energy usage. This was quite a long time dedicated to just collecting data for the training phase.
3.3 A Priori Analysis of Data

3.3.1 'Good' Flags

Machine learning needs some prior information applied to it so that irrelevant information is discarded and only the most important information is kept to learn from. This is where some preprocessing comes in to place. The results we attain before have a lot of extra information that cannot go directly into a machine learning representation, we will not learn anything useful if we simply tell an algorithm that here is program X and here are 500 combinations of flag settings for it. There is no way it would be able to produce a result predicting good flags for an unseen program.

Instead, what we must do is take that information, and literally filter it. The idea used is to filter it in such a way that the end result looks like having program X, flags A,B..N that are deemed most influential and combining this with the feature vector representation of the program so that the learning knows for program X, the representation is in the form of a feature vector, that it can associate with it, as well as N amount of flags that are deemed influential for that specific program only. This is a much more effective way of learning as the extra information is taken out regarding individual flag settings and results, but is retained in the form of a structured set that informs us that we know these flags are good for this program. Now our machine learning will be able to take this information, and apply it to an unseen program by various techniques such as looking for which feature vector closest represents the unseen program from the training set, and attempting to use the 'good' flags from those programs.

What is a 'Good' flag?

Sorting the energy results by least energy used, you can immediately see how many of the 500 runs outperform the -O3 level. From this information, I began by summing up how often each flag occurred over these better solutions as a number, and how often the same flags occurred in the rest of the solutions.

An example is, for 500 runs of prog1.c, 88 results out performed -O3. In these 88 results, the flag -fomit-frame-pointer appears in 85 solutions. On the contrary, out of the 412 worse solutions -fomit-frame-pointer appears in only 158. By taking the ratio of the occurrence in better solutions and dividing it by the occurrence in worse solutions, one can determine by the resulting ratio, if a flag appears more often in better solutions, leading to an expectation that this flag is influential in reducing energy use. In general over a 500 randomly chosen sample space, if a flag has no effect, it should lean towards or close to a 1.0 ratio. For our above example, this ratio results in 2.5477 which is far above and beyond our expected value. It is by this metric that one can filter some influential flags out of the energy data.

This analysis is then done for each program over its set of results to result in a list of programs plus influential flags. This is the same metric that is believed to have been used by the MILEPOST team, although it was not specifically declared in their published work.

3.3.2 Bad Flags

The same metric is used to classify bad flags for programs alongside the good ones. So using the ratios described above, I also compute the ones that are significantly under the average value of 1.0. I choose to use the threshold value of 0.85, as I believe that allows it to factor in results that by chance were away from this average value, but by a large enough difference to make it significant indicating its more likely to be due to the flags influence rather than it being unusually picked more frequently in bad solutions by the random generator.

I decided to include this information so that the machine learning would know bad examples as well as good examples. The logic being that the learning would be able to distinguish between the 56 features not only what makes the flag good, but some rules why it may be bad by the negative examples. I didn’t want there to be too few instances to learn anything significant from, which was the case when it is based on just 'good' flags. There end up being 3-5 instances of programs per flag.
which across 56 features I did not expect to perform too well. Having this extra information allowed the data to be less sparse and potentially uncover some better learning rules.

### 3.3.3 Representing the Data

All this information about good and bad flags that has been processed now needs to be converted into a representation that will be used by the WEKA framework. This means consolidating the information into the ARFF format, which is commonly used as a standard within WEKA.

For each of the 78 flags that are considered, it’s own ARFF file must be created. This is because each file (representing a flag) will then contain instances as training data of the feature vectors from the corresponding programs that the flags determined to be a ‘good’ or ‘bad’ flag of. To explain this visually, a snippet of one of the files -fomit-frame-pointer is shown below.

![Example of an ARFF file type](image)

Note: Some information has been edited out for visualisation purposes

This shows the entire structure of an ARFF file. This consists of a relation `@relation milepost`, attributes `@attribute ft1 numeric` along with their data type representing the feature vector and finally a nominal class label `@attribute class {0,1}`. Then the instance data follows under `@data` which shows an example of a feature vector combined with class labels.

This process of translating into the ARFF format turned out to be lengthy, and due to this - I utilised version control to have full control of reverting back to previous files, in-case of error, or any potentially worse issues. I made sure to commit changes as I worked through each program over all the flags - meaning at any stage I can revert to have training data over only the first X programs if required.

---

3ARFF - Attribute Relational File Format
3.4 Machine Learning Stage

This brings us to the machine learning stage. The data is now in a format to execute under the WEKA framework.

3.4.1 WEKA

There is a Javadoc API for using WEKA which is usually a standalone application, under the Java programming language. This is the approach I have used, incorporating the weka.jar file into my code to gain access to the internal structures containing algorithms. At this point I had to choose the algorithms that would be compared. I wanted a few different types, but to also compare under similar types of algorithms. For example, I chose decision trees as a type of algorithm, but under this broad definition chose to evaluate the J48 algorithm due to it’s popularity, and the REPTree algorithm to compare against.

The other algorithms that I have provisionally chosen to test are the following:

- REPTree
- J48
- Nearest Neighbour
- KStar
- LADTree
- SMO

Under each of these algorithms, I expect to test multiple parameters such as unpruned trees for decision trees, and varying values of K for the NN (Nearest Neighbour). It also allows a balance of some quite different types of machine learning to take place, such as Support Vector Machine (SVM) algorithms with SMO, decision tree algorithms in REPTree, J48 and LADTree, as well as some lazy learners in KStar and Nearest Neighbours. A lazy learner is an algorithm that does the work when an instance is presented, unlike other models which parametrise the space such as decision trees which would follow a built model. One more feature I am looking to use in comparing these is the time taken which measures how long is needed to build the models as well as classify all test cases.

Why Decision Trees?

Decision trees allow visualisation of the model in the form of a graph, so have an advantage over traditional black-box methods by allowing a full view of the decisions taken at each step. This allows for human interaction and external understanding of what is happening during classification. The trees also display which features are most important, or provide us with the most information - again a benefit that can benefit future learning. As the nodes in a tree are hierarchical, essentially feature selection is performed within the model, which is a technique that commonly can be done before creating a model. Feature selection is essentially selecting a subset of the existing features, often disregarding features that are irrelevant or do not provide much information in terms of entropy.

Trees are also an extremely fast method to classify with. If the tree consists of full binary splits, the time that is needed to traverse a solution is in the magnitude of $O(\log n)$. REPTree is a rapid exploration of random trees, which creates a random space filling tree to search over. This slight difference over the traditional algorithm (J48) is the reason I wanted to compare and see if this made any noticeable difference.

Why Nearest Neighbour?

The nearest neighbour algorithm is a popular algorithm when it comes to machine learning. It works by placing a test case in the hyper dimensional feature space, and finds which training example it closest matches, using the Euclidean distance metric. In cases where the feature space consists of two or three dimensions, this data can even be visualised, however in our case of a 56-dimension
hyper space, this feature of the algorithm no longer proves an advantage meaning we cannot visualise the classifications like decision trees. It becomes more of a black box type of algorithm, as in, we know what we input, and the output - but not quite the exact steps taken in between.

One reason I have decided to test this algorithm is because the work by MILEPOST also used k-nearest neighbour in their study. They used a 1-NN algorithm to find solutions over this high dimensional space, and in theory with a large enough training set, this should be a good method. The problem with this method comes when the hyperspace is sparse, then we are not sure if the nearest training example is of any use, it may be the nearest, yet actually a far representation from the test case.

One solution to this is to attempt to reduce the features into a lower dimensional space. For example, using feature extraction methods such as Principal Component Analysis (PCA) to reduce the 56-dimension hyper space into a more manageable dimension. Some of my preliminary work was on this task, and the conclusion was that, still the bottleneck for this was the lack of programs meaning not enough information in a high dimensional space to properly extract useful features into a lower dimensional space. This PCA method could actually obfuscate a good feature, and an argument can be made that instead of trying to reduce these features, work could be done on analysing the quality of them for energy optimisations instead. As these were created by a team targeting execution and code size improvements remember.

3.4.2 Building Classifiers

Implementing the classifiers to begin modelling the training data was the next step. This work was done in Java as mentioned, and I created a class that would loop through each .arff file in a folder, create one of the chosen classifiers and then build the model accordingly. Each classifier was tested by simply changing the choice of classifier, allowing quite simple access to different algorithms. In the implementation, I wanted to be able to identify how many predictions would be correct, and how many were false negatives. This is because, due to the relatively rare nature of having a flag determined 'good', I wanted to minimise the times this was incorrectly predicted to be not 'good'. We would rather have an algorithm that performs slightly worse, but never gives a false negative as that is going to be preferred. You’d rather get as many true correct answers than more false negatives when comparing the classifiers.

Each algorithm has many parameters attached to it in WEKA, so that one can modify slight variations in the algorithms. An example of this is modifying the value of $k$ in K-NN, or the Boolean value describing if a tree is unpruned or not in decision trees. These parameters will be used, and there will be results stored for a variety of parameters to see if any of those make a difference which can then be analysed after.

3.4.3 Testing Classifiers

For testing the classifiers, I opted to create 40 test instances for each flags model. We want the testing cases to be known correct outputs, so for this reason I used the first 40 programs that we collected energy readings of. For each flag, the instance included the feature vector of those programs as well as known class outputs. This method leads to a total of over 2700 test cases that are evaluated over the 78 machine learning models for each flag. This creates a vast unique test set that will evaluate the performance of each algorithm.

The implementation in Java that records correct predictions will then be used to generate the output indicating how many predictions were correct, a percentage and the number of false negatives that a part of those results. These three metrics will then form the basis of our comparison in the results section.
3.5 A Posteriori Analysis of Results

Here I will present the results and explain the findings that have come from them. I will attempt to give reasoning to any results that are out of the ordinary, or unexpected. I plan to explain through each type of learning algorithm tested first, presenting the results and then comparing them to each other. For the following results there were a total of 2730 test cases and 149 of these cases represented the true positive values, i.e. values of class 1. These are the values the percentages are based out of as a fraction.

3.5.1 Decision Trees

There were three decision trees tested, the J48 classifier, REPTree and LADTree. Figure 3.7 shows the results from the findings between these trees.

![Decision Trees](image)

**Figure 3.7:** Results of decision tree algorithms

**REPTree**

The results show that the REPTree algorithm performs with 86.08% accuracy over the 2730 test cases presented. This is a very high accuracy rate, and in fact, the highest of all decision trees. There are a few reasons why this is higher than the others when we begin to look at the false negative rates. It manages to mis-classify a huge 81% of all the true cases marked with a 1 in our tests. This number is way too large to consider this algorithm good for the given task, we want to minimise the amount of false negatives that are produced so that we are rarely incorrectly classifying true results as negative results.

Given this high rate of false negatives as well as the high accuracy rate, the conclusion to the results shown here indicates that this algorithm tended to classify towards class label 0 more than class label 1, which explains why it has a high false negative rate. It also explains why the accuracy seems high, because remembering that in our test cases, of the 2730 instances, only 149 are labelled as class 1, so an algorithm tending to predict class 0 would seemingly result in high accuracy here. Having said this, we can also see that it was the fastest by some measure in classifying all the instances, an advantageous property of decision trees in general.
CHAPTER 3. PROJECT EXECUTION

J48

Default  The J48 decision tree performed slightly worse in terms of accuracy over its compatriot REPTree, however we can see that the false negatives were at a much more respectable 11%. Although the accuracy falls to 69.63%, the reduction in false negatives indicates to us that this algorithm has likely learnt slightly more optimal rules as it is no longer continuously predicts the class to be 0 which we indicated as a problem of REPTree.

The default parameters under the J48 tree mean that the tree is pruned, i.e. a method that reduces the size of the decision tree by removing nodes that provide little information gain. We will see in the next algorithm what the difference leaving this tree unpruned makes in comparison.

Unpruned  The only difference in this result is that I enabled the setting so that the tree is left unpruned. The results remain very similar, with a slight decrease in accuracy, a decrease in false negatives as well as a slight increase in the time taken to classify. I think there may be some reasons towards this, having that slightly extra information left in the trees probably allowed it to make less errors in the false negatives because the trees would have slightly more useful information in branches that are learnt to classify an instance as 1.

However, this extra information as shown can also mean mis-classification in general because of its unreliability (its lack of information gain) resulting in a decreased accuracy rate. Out of these two J48 models there isn’t many differences so I’d argue that either would be suitable models, with the slight edge being given to the unpruned tree due to its smaller false negative rate.

The timing differences between both are minimal, with trivially expecting an unpruned tree to take longer to be searched as it would contain more branches and leaves.

LADTree

Immediately these results show that the LADTree algorithm is the longest to execute of all. This likely stems from the fact that this algorithm uses boosting to create alternating decision trees, lending to its longer learning and classification time. The boosting does lead to the case of only 1 in 149 false negatives produced, which is the advantage shown by LADTree over the other tree algorithms.

Analysis of Trees  This leads me to the conclusion that over all the decision trees, the unpruned J48 classifier is my choice on the best of the four tested. Its relatively quick, with a low false negative rate as well as a relatively high classification rate giving it the edge over both REPTree and LADTree respectively.
3.5.2 Lazy Learners

The lazy learner algorithms that were tested included the Nearest Neighbour and KStar algorithms. Figure 3.8 shows the results from the findings between these learners.

![Lazy Learners](image)

**Figure 3.8: Results of lazy learner algorithms**

**Nearest Neighbour**

The NN algorithm uses the value $k$ to determine how it classifies an instance. It determines this by finding the $k$ closest instances in the hyper dimensional space using a measuring metric which is commonly chosen to be Euclidean. That metric calculates how the distance from point A to point B using a straight line, involving Pythagorean maths. This metric is what was used in the following values of $k$.

**K=1:** 1-NN performs quite well, the results in accuracy near the values recorded by the J48 tree shown before. This case performs really well in regards to the false negatives predicted. We can see that it is slightly slower than decision trees over the time taken to build and classify all models, however 1.21s is still relatively fast and not exactly a negative. The reasoning behind why the results would be how they are could be as follows. A low false negative rate could be due to the fact that with 1-NN, the instances to be test matched up well to the training set as the closest instance matches in the training set.

**K=2:** 2-NN compares in performance to the REPTree algorithm. It is high in false negatives, and high in accuracy, with the same reasoning believed to be the logic behind the results. The reasoning is that it must be tending to prefer class 0 to have high accuracy and high false negatives. My intuition makes me think that what happens here is that in a hyper-dimensional feature space of 56 features, that space if visualised, would be incredibly sparse. So using 2 closest neighbours to classify an instance would likely result in one instance being relatively close, but the other having a large chance of being incredibly far away, despite being the next closest instance due to this sparseness.

With the cases when instance 1 has class label 1 and instance 2 has class label 0, it is not explicitly mentioned what the algorithm does to choose when it returns an equal amount of votes for each class. That is why given the results tending to class label 0, I believe the algorithm uses the fact that the majority of instances over the whole model have class label 0 and ends up voting class label 0 in tied cases. This may not be correct, however the logic behind it would make sense given the results.
KStar

KStar is the best performer of the lazy learners. The reason being its higher accuracy than 1-NN and equally low false negative reports. These combined give me confidence in saying it outperforms the NN algorithm, and compares directly to the J48 tree classifier. The only drawback of this method is the time taken to build and classify all models is nearly twice as long as the trees take, and roughly 70% slower than the NN algorithm. While on this data set this may not seem much issue, if this was expanded onto a large scale training set, it would definitely start to become a disadvantage of the algorithm. Having said all of this, I conclude the lazy learners section by saying on classification performance alone the KStar algorithm has outperformed the NN algorithm.

3.5.3 SMO

The SMO algorithm is an implementation of the SVM learning models [16]. SMO works by breaking the quadratic problem an SVM model needs to be trained into smaller quadratic problems such that it is solvable. This provides a huge performance increase over normal SVM learning methods which tend to scale between linear and cubic in relation to the training size [16].

![Figure 3.9: Results of SMO algorithm](image)

We can see from the results here that the SMO algorithm is one of the worse algorithms tested. It is not in comparison to the two we have identified so far as working well, J48 and KStar, but it is not as bad as the REPTree and 2-NN algorithms that we’ve identified to be the worst so far. The accuracy remains relatively close to some of our better models at 68%, and is its best feature. However with nearly 34% of all the true positive results being classified as false negatives, there is a worry that this would give us far too many false negative results if it was the chosen implementation. Although we mention that the SMO algorithm gives an increase in performance over the notoriously slow SVM learning methods, it is still the slowest performer of the eight algorithms tested. Taking 2.83s to create all the models and classify the data is just over a 300% increase over the fastest model (REPTree).
3.5.4 Summary

To conclude the analysis of the machine learning algorithms, we have seen that there are three potentially viable algorithms that perform well on this case. The other five are slightly less creditable and reasoning has been given for why potential results are misleading or incorrect. This brings me onto the fact that the sample size of programs that were used may possibly still be too small to fully evaluate the 56 features presented in the feature vector. I believed that there were enough when 84 programs had been assembled, however with the analysis of the 2-NN algorithm and other cases where it seems over fitting could be the case, one has to question the suitability of the range of programs used. If it was possible to expand this to over double and reach 200 programs, this could become worth looking at the performance of the learners again, as we likely obtain a vast amount of detail in each of the 56 features presented.

Given the data we do have and the results that we have seen, it leads me to believe that the best machine learning candidates fall between the three algorithms 1-NN, KStar and J48. Two of these algorithms are lazy learners and one is a decision tree, perhaps the added benefit of being able to visualise a decision tree model and its superior speed lends it to be the preferred choice. One algorithm doesn’t need to be solely chosen, so instead of picking out one of these three, I will conclude by saying that all three of these are potential candidates. There are cases for each to be used, but the aim that we set out at the start of this project was to analyse these results and compare the classifiers which has been met.

Relating the findings to the information we know about what currently exists, i.e. MILEPOST, it’s reassuring to have results where an algorithm that they used also performs well and is in the best of our results with 1-NN. They used this 1-NN algorithm as was mentioned and their results on their full solution outperformed GCC’s -O3 optimisation level by 11%.

Below in figure 3.10 is a graph that shows each algorithms performance compared to each other for full comparison.
Chapter 4

Conclusion

4.1 Summary of Achievements

There were several aims of this project that were sought out to be achieved. The most important of these aims was to conclude analysis of various machine learning algorithms in the given context, to gain an insight into which algorithms perform well. I believe that I have successfully completed this achievement and concluded the results found to show my reasoned opinion on which algorithms are good for this task.

Another aim was to investigate the suitability of using a Pandaboard as the hardware that would be the testing device. Through consultation with PhD student James Pallister, and contact with hardware experts in the team at MAGEEC, I believe that I have successfully completed this aim by concluding that the Pandaboard was a great device to test and run the programs on. The balance it produces between fully embedded systems and the ability to run an OS on it meant that it provided a platform that could expand much more than a restricted embedded system but very much represent the key elements from these.

A system has been developed and saved on line using version control of all the programs that were used in this work as well as the energy readings that were stored for those associated programs. This means I leave the project with full results stored for later use or continuing the project, or if a third party wants to use it for something similar they have some data to begin with. This is important because of the current lack of such an easy to find open source benchmark suite.

Energy readings were collected using an automated framework and hardware connected to the Pandaboard that eventually formed the input to the machine learning phase. This was a large process and step in the project, with the analysis of the data acting as a form of pre processing, the next stage that was performed before the training files were composed.

The WEKA framework was the utilised for access to a vast amount of machine learning algorithms and was used to generate the results that the analysis was based on. These results led to the conclusions that out of the eight algorithms tested and compared, three worked well enough to be considered as a chosen implementation for the MAGEEC project. These three were the 1-NN which MILEPOST also used in their work on execution time improvement, the J48 decision tree and KStar algorithm that is a lazy learning algorithm along with 1-NN.

4.2 Project Status

The current project status is that there is a working implementation that has provided results. There are 84 programs, along with a database of results that provide the compilation flags, energy used and time taken for each run to be performed. It is also in a state that the work has identified the ‘good’ flags or influential flags for each program, both good and bad ones, and the machine learning representations have been created around this.
The aim of producing results and analysing them has been achieved with evidence of this shown towards the end of Chapter 3 in sub section 3.5. Evidence of the objective to run and collect the energy readings for the programs run on the Pandaboard is also shown in Chapter 3 where the method used is explained in detail as well as the results from this which are uploaded as part of this work.

I believe that all the aims and objectives that were set out in Chapter 1 have been met through this project.

4.3 Future Plans

Though this system did produce results, I believe there is much more work that could be done on this. There is scope to go beyond looking at just compilation flags individually, and to attempt to glean some information into how the ordering of these flags affects energy use, or even execution time if that is simpler to do. Simply permuting the same flag combination could be an effective energy reduction. This would be beneficial as it has not been done before and there really is a lack of understanding in the current market about this, despite the vast amount of optimisation flags that currently exist. Looking into this would be enough for a project in itself, and the findings could have widespread benefits to existing optimisation techniques. A side project that would work well alongside this would be something that attempts to solve the problem of how complex the data could get including order combinations of flags and exploring that space efficiently.

Apart from trying to look into the inter-flag performance, this project could be extended and improved by improving the data set to be much larger and incorporating many more open source programs. This would not only improve its recognition as an all in one open source benchmark suite, but also allow much more information to be recorded about energy runs on differing types of applications, something that likely will have an effect on a future learnt model if it was updated. Increasing the programs and training set could provide information into whether the current results that I explained to be possibly affected by such a sparse hyper space could be improved, as these results were not as expected, but intuitively correct.

Expanding the database of information by working on more hardware is another viable future extension. The Pandaboard was a great system to work on, but there are also many other embedded systems that could be tested such as a BeagleBone \(^1\) or Parallela \(^2\) board. There would be room to analyse what differences come from using another system, be it based on the change in CPU, or lower memory available. Processors will all perform slightly different and the results would never be the same so there would always be an avenue to pursue here. The problem that this would face would be the availability of test programs however, as if a system was chosen that is fully embedded and could not host an OS, all system calls would have to be removed from the source code and ported so that it can run on a board with less memory or functionality. This was a major reason behind choosing the Pandaboard in the first place, and would prove as the biggest challenge in exporting this work to multiple boards.

The machine learning representation could be changed, to see if this affects what is learnt. After all, the algorithms are simply working off the knowledge supplied to them. The current system used a class measure of 1 for a ‘good’ flag, and treated everything else as class 0, including the flags that were identified as really ‘bad’. Perhaps a multi-modal class that uses a measure of 1 for ‘good’ flags, 0 for flags that seemingly are not influential, and \(-1\) for ‘bad’ flags instead may affect the results returned. I am unsure if the chosen system for this project was the most optimal way of representing the data, but I believed it to be a good method as the learners could attempt to differentiate between extreme cases of both influentially ‘good’ and influentially ‘bad’ flags.

\(^1\)Beaglebone: \url{http://beagleboard.org/bone} \\
\(^2\)Parallela: \url{http://www.parallella.org/board/}
Bibliography


### Appendix A

**MILEPOST Feature Vector**

<table>
<thead>
<tr>
<th>Feature number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>Number of basic blocks in the method</td>
</tr>
<tr>
<td>f2</td>
<td>Number of basic blocks with a single successor</td>
</tr>
<tr>
<td>f3</td>
<td>Number of basic blocks with two successors</td>
</tr>
<tr>
<td>f4</td>
<td>Number of basic blocks with more than two successors</td>
</tr>
<tr>
<td>f5</td>
<td>Number of basic blocks with a single predecessor</td>
</tr>
<tr>
<td>f6</td>
<td>Number of basic blocks with two predecessors</td>
</tr>
<tr>
<td>f7</td>
<td>Number of basic blocks with more than two predecessors</td>
</tr>
<tr>
<td>f8</td>
<td>Number of basic blocks with a single predecessor and a single successor</td>
</tr>
<tr>
<td>f9</td>
<td>Number of basic blocks with a single predecessor and two successors</td>
</tr>
<tr>
<td>f10</td>
<td>Number of basic blocks with two predecessors and one successor</td>
</tr>
<tr>
<td>f11</td>
<td>Number of basic blocks with two successors and two predecessors</td>
</tr>
<tr>
<td>f12</td>
<td>Number of basic blocks with more than two successors and more than two predecessors</td>
</tr>
<tr>
<td>f13</td>
<td>Number of basic blocks with more than 15 instructions</td>
</tr>
<tr>
<td>f14</td>
<td>Number of basic blocks with more than 15 instructions in the interval [15, 500]</td>
</tr>
<tr>
<td>f15</td>
<td>Number of basic blocks with more than 15 instructions greater than 500</td>
</tr>
<tr>
<td>f16</td>
<td>Number of edges in the control flow graph</td>
</tr>
<tr>
<td>f17</td>
<td>Number of critical edges in the control flow graph</td>
</tr>
<tr>
<td>f18</td>
<td>Number of abnormal edges in the control flow graph</td>
</tr>
<tr>
<td>f19</td>
<td>Number of direct calls in the method</td>
</tr>
<tr>
<td>f20</td>
<td>Number of conditional branches in the method</td>
</tr>
<tr>
<td>f21</td>
<td>Number of assignment instructions in the method</td>
</tr>
<tr>
<td>f22</td>
<td>Number of unconditional branches in the method</td>
</tr>
<tr>
<td>f23</td>
<td>Number of binary integer operations in the method</td>
</tr>
<tr>
<td>f24</td>
<td>Number of binary floating point operations in the method</td>
</tr>
<tr>
<td>f25</td>
<td>Number of instructions in the method</td>
</tr>
<tr>
<td>f26</td>
<td>Average of number of instructions in basic blocks</td>
</tr>
<tr>
<td>f27</td>
<td>Average of number of phi-nodes at the beginning of a basic block</td>
</tr>
<tr>
<td>f28</td>
<td>Average of arguments for a phi-node</td>
</tr>
<tr>
<td>f29</td>
<td>Number of basic blocks with phi nodes in the interval [0, 3]</td>
</tr>
<tr>
<td>f30</td>
<td>Number of basic blocks with more than 3 phi nodes</td>
</tr>
<tr>
<td>f31</td>
<td>Number of basic block where total number of arguments for all phi-nodes is in the interval [5, 10]</td>
</tr>
<tr>
<td>f32</td>
<td>Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5]</td>
</tr>
<tr>
<td>f33</td>
<td>Number of switch instructions in the method</td>
</tr>
<tr>
<td>f34</td>
<td>Number of unary operations in the method</td>
</tr>
<tr>
<td>f35</td>
<td>Number of instruction that do pointer arithmetic in the method</td>
</tr>
<tr>
<td>f36</td>
<td>Number of indirect references via pointers (** in C)</td>
</tr>
<tr>
<td>f37</td>
<td>Number of times the address of a variables is taken (&quot;&amp;&quot; in C)</td>
</tr>
<tr>
<td>f38</td>
<td>Number of times the address of a function is taken (&quot;&amp;&quot; in C)</td>
</tr>
<tr>
<td>f39</td>
<td>Number of indirect calls (i.e. done via pointers) in the method</td>
</tr>
<tr>
<td>f40</td>
<td>Number of assignment instructions with the left operand an integer constant in the method</td>
</tr>
<tr>
<td>f41</td>
<td>Number of binary operations with one of the operands an integer constant in the method</td>
</tr>
<tr>
<td>f42</td>
<td>Number of calls with pointers as arguments</td>
</tr>
<tr>
<td>f43</td>
<td>Number of calls with the number of arguments is greater than 4</td>
</tr>
<tr>
<td>f44</td>
<td>Number of calls that return a pointer</td>
</tr>
<tr>
<td>f45</td>
<td>Number of calls that return an integer</td>
</tr>
<tr>
<td>f46</td>
<td>Number of occurrences of integer constant zero</td>
</tr>
<tr>
<td>f47</td>
<td>Number of occurrences of 32-bit integer constants</td>
</tr>
<tr>
<td>f48</td>
<td>Number of occurrences of integer constant one</td>
</tr>
<tr>
<td>f49</td>
<td>Number of occurrences of 64-bit integer constants</td>
</tr>
<tr>
<td>f50</td>
<td>Number of references of a local variables in the method</td>
</tr>
<tr>
<td>f51</td>
<td>Number of references (def/use) of static/extern variables in the method</td>
</tr>
<tr>
<td>f52</td>
<td>Number of local variables referred in the method</td>
</tr>
<tr>
<td>f53</td>
<td>Number of static/extern variables referred in the method</td>
</tr>
<tr>
<td>f54</td>
<td>Number of local variables that are pointers in the method</td>
</tr>
<tr>
<td>f55</td>
<td>Number of static/extern variables that are pointers in the method</td>
</tr>
</tbody>
</table>

*Table 1: List of static program features currently available in the MILEPOST GCC*