Author:
Blackmore, Craig

Title:
Inductive Logic Programming for Compiler Tuning

Inductive Logic Programming for Compiler Tuning

By

CRAIG BLACKMORE

Department of Computer Science
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Engineering.

DECEMBER 2018

Word count: Thirty-two thousand words

Abstract

Software performance is heavily dependent on the compiler settings used to generate executables from source code. These settings have a significant impact on execution time, energy consumption and code size, and they can be modified easily without the need to alter the underlying source code nor the compiler itself.

Automated Compiler Tuning (ACT) is the practice of selecting compiler settings to improve the performance of computer software. Existing work shows significant potential gains beyond industry standard GCC `-O3`, but the search space is very large. This motivated the development of predictive machine learning approaches, most of which rely on feature vectors of simple statistics that summarise program characteristics. Inductive Logic Programming (ILP) can potentially exploit full program structure that existing methods do not consider.

This thesis applies ILP to ACT to investigate the hypothesis that exploiting full program structure increases the accuracy and effectiveness of predictive compiler tuning. It explores the design, feasibility and benefits of applying ILP to discover meaningful rules that relate effective compiler settings to program structure. During the process, assumptions in existing work are identified and challenged to increase understanding of the problem and give a robust evaluation of compiler tuning techniques.

The research is conducted on the industry standard GCC compiler and a state-of-the-art benchmark suite targeting ARM Cortex-M3 (CM3) and Cortex-A8 (CA8) processors. The ILP method outperforms two state-of-the-art approaches and also identifies settings that should always be disabled to improve performance on the target platform. This leads to the construction of two new program-agnostic optimisation levels that outperform GCC's `-O3` and are competitive with program-specific predictive approaches. Finally, a case study on an image classification application from the ARM CMSIS Neural Network library finds an 8% improvement over `-O3` on the CM3.

Acknowledgements

I would like to thank my supervisors Oliver Ray and Kerstin Eder for their valuable feedback, time and support throughout my PhD. Thank you to Simon Hollis for first introducing me to this research area.

My thanks to Jeremy Bennett, Simon Cook, Ed Jones, Graham Markall and everyone at Embecosm for hosting my internship, for valuable discussions and their continuing commitment to compiler research.

Thank you to friends and colleagues at the University including: James Pallister, Steve Kerrison, Jeremy Morse, Kyriakos Georgiou, Dejanira Araiza Illan and many more.

Thank you to my family for encouraging and supporting me throughout my life. My thanks to David Timms, Udoyon Bhaduri, James Dixon, Julie Cox, Alex Neill and all of my open mic friends for keeping my love of music alive. Finally, thank you to Kuiama Lewandowski for always knowing what to say when I needed it the most and being there through the highs and lows with endless cups of tea and moral support.

AUTHOR'S DECLARATION

Table of Contents

Page

List of Tables **xi**

List of Figures **xiii**

List of Listings **xv**

1 Introduction **1**

1.1 Research questions . . . 3

1.2 Contributions . . . 4

1.3 Related publications . . . 6

1.4 Thesis structure . . . 7

2 Background **9**

2.1 Compiler tuning . . . 9

2.1.1 Problems in compiler tuning . . . 10

2.1.2 Standard optimisation levels . . . 11

2.1.3 Iterative compilation . . . 11

2.1.4 Machine learning . . . 15

2.1.5 Scope . . . 21

2.2 Inductive Logic Programming . . . 22

2.2.1 Motivating example . . . 23

2.3 Hardware platforms . . . 24

2.4 Benchmarks . . . 25

2.4.1 Bristol/Embecosm Embedded Benchmark Suite (BEEBS) . . . 25

2.4.2 Collective Benchmarks (cBench) . . . 27

2.5 Measuring energy and time . . . 28

3 Quantifying the potential benefits of compiler tuning **29**

3.1 BEEBS data initialisation . . . 30

3.2 Evaluating configurations . . . 31

3.3 Standard optimisation levels . . . 31

3.4 Analysing the configuration space with Random Iterative Compilation 35
3.4.1 Distribution of results 35
3.4.2 Energy-time correlation 40
3.5 Finding further improvements with Combined Elimination 40
3.5.1 Comparison of RIC and CE 41
3.5.2 Impact of different baselines 44
3.6 Summary of potential gains 45

4 Re-evaluation of predictive compiler tuning approaches 47
4.1 Evaluating machine learning models 47
4.2 Milepost 48
4.2.1 Results 48
4.3 COBAYN 50
4.3.1 Choice of flags 50
4.3.2 Performance of COBAYN models 52
4.4 Summary of results 53

5 Constructing new program-agnostic optimisation levels 55
5.1 Constructing program-agnostic optimisation levels 56
5.1.1 Adapting combined elimination 56
5.1.2 Threshold trade-off in constructing `-Ocm3` 57
5.1.3 Analysis of two excluded flags 58
5.1.4 Cross-validation of `-Ocm3` 60
5.1.5 Testing on other platforms 62
5.2 Finding a small set of good configurations 63
5.2.1 Method 64
5.2.2 Results 66
5.3 Conclusion 67

6 Applying ILP to compiler tuning 71
6.1 Method 72
6.1.1 Identifying examples of significant flags 72
6.1.2 Extracting program features for ILP+FV 76
6.1.3 Extracting Intermediate Representation for ILP+IR 76
6.1.4 Learning rules for bad flags 77
6.2 Results and evaluation 78
6.2.1 ILP+FV and ILP+IR 78
6.2.2 Hybrid approach 80
6.3 Conclusion 81

7 Focusing ILP based compiler tuning 83
7.1 Limitations of initial approach . . . 83
7.2 Improved Inductive Logic Programming based compiler tuning . . . 84
7.2.1 Identifying significantly good or bad flags . . . 84
7.2.2 Restricting the search space . . . 85
7.2.3 ILP search algorithm and cost function . . . 85
7.2.4 Cross-validation of new ILP approach . . . 86
7.2.5 Insights from human-readable rules . . . 86
7.3 Constructing a new program-agnostic optimisation level . . . 87
7.3.1 Performance of `-Ocm3-ilp` . . . 87
7.4 Testing on the Cortex-A8 processor . . . 88
7.5 Conclusion . . . 89

8 Transferability of compiler tuning methods and models across platforms and to a real world application 91
8.1 How well do the compiler techniques perform on different platforms? . . . 91
8.2 Can knowledge gained on one platform help to optimise another platform? . . . 92
8.2.1 Transferability of best configurations found by iterative compilation . . . 92
8.2.2 Transferability of program-agnostic optimisation levels . . . 92
8.2.3 Transferability of machine learning models . . . 94
8.2.4 Summary . . . 96
8.3 Do the gains transfer to a real world application? . . . 97
8.4 Critical evaluation . . . 98

9 Conclusion 103
9.1 Future work . . . 104

Bibliography 107

List of Tables

Table **Page**

2.1 Hardware features of the target platforms 25

3.1 Average improvement found by RIC for execution time, energy consumption and code size 36
3.2 Average improvement made by RIC and CE combined for execution time 45

4.1 Average improvement made by Milepost and COBAYN 53

5.1 Average improvement made by program-agnostic configurations 68

6.1 Average improvement made by ILP prototype 81

7.1 Average improvement made by ILP 90

8.1 Average improvement made by transferring the best RIC configuration for each BEEBS program on the CM3 to the CA8 and vice versa 97
8.2 Average improvement made by transferring program-agnostic configurations between CM3 and CA8 97
8.3 Average improvement made by transferring ILP models between CM3 and CA8 . . . 97
8.4 Summary of improvements on neural network case study program 98
8.5 Average relative standard deviation in the RIC dataset 100

LIST OF FIGURES

FIGURE **Page**

2.1 HV metric 14
2.2 Euclidean distance between normalised feature vector of each cBench program 26
2.3 Euclidean distance between normalised feature vector of each BEEBS program 26

3.1 Execution time of standard optimisation levels (CM3 BEEBS) 32
3.2 Energy consumption of standard optimisation levels (CM3 BEEBS) 32
3.3 Code size of standard optimisation levels (CM3 BEEBS) 32
3.4 Execution time of standard optimisation levels (CA8 BEEBS) 33
3.5 Energy consumption of standard optimisation levels (CA8 BEEBS) 33
3.6 Code size of standard optimisation levels (CA8 BEEBS) 33
3.7 Execution time of standard optimisation levels (CA8 cBench) 34
3.8 Energy consumption of standard optimisation levels (CA8 cBench) 34
3.9 Code size of standard optimisation levels (CA8 cBench) 34
3.10 Execution time, energy consumption and code size of 1000 random configurations applied to each benchmark (CM3) 37
3.11 Execution time, energy consumption and code size of 1000 configurations applied to each benchmark (CA8 BEEBS) 38
3.12 Execution time, energy consumption and code size of 1000 configurations applied to each benchmark (CA8 cBench) 39
3.13 Correlation between time and energy for 1000 random configurations across all benchmarks 40
3.14 Best execution time achieved by RIC (1000 configurations) and CE (CM3) 41
3.15 Best execution time achieved by RIC (1000 configurations) and CE (CA8 BEEBS) . . 41
3.16 Best execution time achieved by RIC (1000 configurations) and CE (CA8 cBench) . . . 41
3.17 Average of best execution time for each benchmark after each configuration tested by RIC (1000 configurations) and CE (CM3) 42
3.18 Average of best execution time for each benchmark after each configuration tested by RIC (1000 configurations) and CE (CA8 BEEBS) 42

3.19 Average of best execution time for each benchmark after each configuration tested by RIC (1000 configurations) and CE (CA8 cBench) 42
3.20 Best execution time achieved after each configuration tested for 1000 random configurations, CE-all, CE-O3 and CE-random (relative to O3) 44

4.1 Performance of 1NN-best and 1NN-prob (CM3) 48
4.2 Performance of 1NN-best and 1NN-prob (CA8 BEEBS) 49
4.3 Performance of 1NN-best and 1NN-prob (CA8 cBench) 49
4.4 Performance of exhaustive search (7 flags) and RIC/CE search (133 flags) (CM3) . . . 51
4.5 Performance of exhaustive search (7 flags) and RIC/CE search (133 flags) (CA8 BEEBS) 51
4.6 Performance of exhaustive search (7 flags) and RIC/CE search (133 flags) (CA8 cBench) 51
4.7 Performance of COBAYN in 1-shot and 8-shot mode (CM3) 54
4.8 Performance of COBAYN in 1-shot and 8-shot mode (CA8 BEEBS) 54

5.1 Trade-off between thresholds for constructing `-Ocm3` 58
5.2 10-fold cross-validation of `-Ocm3`, Milepost 1NN and COBAYN 1-shot (logarithmic scale) 61
5.3 Performance of `-Oca8-beebs` and `-Oca8-cbench` on CA8 BEEBS 63
5.4 Performance of `-Oca8-beebs` and `-Oca8-cbench` on CA8 cBench 64
5.5 Performance of each method for producing a reduced set (CM3) 67
5.6 Performance of each method for producing a reduced set (CA8 BEEBS) 67
5.7 Performance of each method for producing a reduced set (CA8 cBench) 68

6.1 Comparison of flag selection methods 75
6.2 ILP+FV, ILP+IR, 1NN-prob and best of 1000 random configurations 79
6.3 ILP+IR/1NN-prob/O3 hybrid and best of 1000 random configurations 80

7.1 Performance of new ILP compared to original ILP approach and Milepost 1NN (logarithmic scale) 86
7.2 Performance of `-Ocm3-ilp` on the Cortex-M3 (logarithmic scale) 88
7.3 Performance of new ILP method and `-Oca8-ilp` on CA8 BEEBS 89
7.4 Performance of new ILP method and `-Oca8-ilp` on CA8 cBench 89

8.1 Best RIC configuration for CM3 tested on the CA8 93
8.2 Best RIC configuration for CA8 tested on the CM3 93
8.3 `-Oca8` on CM3 94
8.4 `-Ocm3` on CA8 BEEBS 94
8.5 `-Ocm3` on CA8 cBench 95
8.6 CA8 ILP BEEBS and cBench models tested on CM3 95
8.7 CM3 ILP model tested on CA8 (BEEBS) 96
8.8 CM3 ILP model tested on CA8 (cBench) 96

LIST OF LISTINGS

LISTING **Page**

5.1 The `-Ocm3` configuration 58
5.2 The `-Oca8-beebs` configuration 63
5.3 The `-Oca8-cbench` configuration 63

Chapter 1

Introduction

As Internet of Things (IoT) edge devices take on more computationally intensive tasks, the execution time and energy consumption of applications running on embedded systems becomes increasingly important. The whole system stack, from low level physical design of hardware to high level design of software and algorithms, contributes to the energy and time efficiency of an application. The compiler has a significant impact on how efficiently the software utilises the underlying hardware.

Modern compilers offer a range of standard optimisation levels (e.g. `-O1`, `-O2`, `-O3`) that are intended to progressively improve the execution time of programs at the expense of increased compile time, code size and/or conformance to software standards (Sec. 2.1.2). The most well known is the `-O3` optimisation level provided by the industry standard GCC compiler [69] (and its more recent competitor Clang [70]). Previous work [27, 19, 57, 10, 18, 3] has shown that `-O3` is far from optimal in many cases; by selectively enabling or disabling compiler flags that control optimisation settings, the compiler can be fine-tuned to improve the performance of a given program and target platform. This leads to significant gains without the need to modify the underlying source code or compiler, but it is infeasible to search exhaustively for the optimal configuration of flags due to the very large search space.

The optimal set of compiler flags is dependent on the target program, platform, compiler and metric. Furthermore, optimisations can interact in complex and unknown ways [19, 57]; for example, applying one optimisation could improve or hinder the success of a later optimisation.

Automatically tuning the selection of these flags is essential to feasibly address the diverse range of programs that can be written, tailor the compiler to its growing set of supported platforms and handle the increasing number of optimisations added to each new version of the compiler. Keeping up with embedded platforms is more difficult than for more popular x86 targets which have a much larger user base and therefore receive a lot more development effort.

Additionally, embedded platforms may not possess all of the hardware features necessary for some optimisations to be successful.

Existing work [27, 59, 71, 61, 22] uses random sampling or more complex iterative compilation methods (which evaluate the performance of a given program compiled with a large number of different configurations) to search for configurations that improve the performance of a target program. This is a time consuming task that must be repeated for each program and platform pair. The slow search time motivated other studies to use iterative compilation to train machine learning approaches to predict good configurations more quickly, at the cost of potentially reduced performance for an unseen program.

Most predictive compiler tuning approaches rely on feature vectors of simple program properties that summarise characteristics of the source code. These predefined features seem to be selected because they are simple and easy to measure rather than for their predictive power [48, 45]. Furthermore, reducing programs to these feature vectors loses valuable structural information that could be used to make more accurate models.

This thesis explores how Inductive Logic Programming (ILP) can exploit an Intermediate Representation (IR) of programs that preserves structural information in order to produce more accurate models by automatically identifying relevant program features. The proposed method uses ILP to infer logical rules that relate effective compiler flags to program features. These rules can be easily converted into sentences and thus provide a white-box approach that gives insight into the relationship between program structure and compiler flags.

In developing the ILP compiler tuning approach, several assumptions in previous work are challenged to increase understanding of the problem and give a robust evaluation of compiler tuning techniques. The choice of benchmarks, evaluation techniques and measurement setup is essential – a re-evaluation of two state-of-the-art predictive approaches shows that they may not work as well as previously thought.

The ILP approach learns several facts that suggest some flags should always be enabled or disabled for the target platform. These facts are used to construct new program-agnostic optimisation levels tailored to each target platform. In this case, the new levels are a by-product of the learned model, but other techniques have specifically targeted the goal of automatically constructing new optimisation levels [38]. Such approaches have received much less attention from the compiler tuning community than machine learning approaches and there does not exist a good comparison between the two. This thesis shows that program-agnostic optimisation can often outperform more complex machine learning techniques that target individual programs.

In industry there are benefits to seeking out program-agnostic configurations rather than applying machine learning to individual programs. The new configurations can be constructed once and distributed easily to be used by many and they do not require machine learning implementations to be installed and tuned. Furthermore, the new optimisation levels could also be tested rigorously to increase confidence in their robustness, for example, by running the GCC test suite on these configurations.

1.1 Research questions

The primary objective of this thesis is to design, implement and evaluate an ILP based compiler tuning approach to investigate the hypothesis that learning directly from IR increases predictive accuracy. The following aims and research questions were investigated in order to realise this objective.

Investigate the potential gains of compiler tuning

- What is the actual impact of the standard optimisation levels on execution time, energy consumption and code size?
- What are the potential gains from fine-tuning optimisation selection?
- What is the relationship between energy consumption and execution time?
- How can it be ensured that benchmarks provide a realistic evaluation of compiler tuning techniques?

Evaluate state-of-the-art predictive approaches on the target platforms

- How well do two state-of-the-art predictive approaches work on a diverse set of benchmarks on the target platforms?

Determine whether tuning to the platform as whole is sufficient or if program-specific optimisation is indeed required

- Are some optimisations universally good/bad for performance on a given platform?
- Can a single program-agnostic configuration perform well or is it beneficial to choose settings for each program?
- Can a small set of configurations be constructed such that there exists at least one good configuration for each program, thus providing a smaller set of configurations on which to run iterative compilation?

Design, develop and evaluate an ILP based compiler tuning approach

- How can the effects of individual optimisations within a configuration be quantified?
- Is it feasible to apply ILP to compiler tuning?

- How can ILP based compiler tuning be designed?
- How can relevant features be automatically discovered by the machine learner?

Investigate the generality and transferability of the techniques developed in this thesis

- How well do compiler tuning techniques perform on different platforms and benchmarks and how can an appropriate method be selected for a given scenario?
- Can models learned for one platform yield useful information for another?
- How well do the results seen on benchmark suites translate to real world improvements on a realistic use case?

1.2 Contributions

This section summarises the main contributions of this thesis.

Contribution to a large open source embedded benchmark suite. The Bristol/Embecosm Embedded Benchmark Suite (BEEBS) is the largest and most diverse free open source benchmark suite for resource limited embedded systems. This thesis fixes a conceptual flaw in the design of the suite to provide a robust and realistic evaluation of compiler tuning techniques (Sec. 3.1).

Analysis of potential gains of compiler tuning In this thesis, the development of compiler tuning methods is motivated by exploring the potential improvements available for execution time, energy consumption and code size (Ch. 3). Iterative compilation is used to time-intensively search the configuration space and provide a lower bound on the improvements available. These experiments find a strong correlation between energy and time on each platform. The analysis also finds that Combined Elimination (CE) [59] outperforms the commonly used Random Iterative Compilation (RIC) [27, 63, 3] in several cases, which is in contrast to a previous study [18]. The results produced in this analysis form the basis for machine learning training data in the rest of this work.

Re-evaluation of two state-of-the-art predictive compiler tuning approaches Milepost [27] and COBAYN [10] are re-evaluated on the two platforms targeted by this thesis using the BEEBS benchmark suite which is over three times as large as cBench used in the original studies. The BEEBS programs present a harder challenge to the machine learner and offer a more thorough and realistic evaluation of compiler tuning techniques. Consequently, BEEBS uncovers limitations that were not visible in previous evaluations with cBench. These re-evaluations enable the previous work to be compared directly with the methods developed in this thesis on the same experimental setup.

Construction of program-agnostic configurations For each platform, a single configuration that outperforms `-O3` across a whole set of representative benchmarks is constructed (Ch. 5). The new configurations provide significant gains and, once constructed, can be applied to any new program with no further effort. They also outperform existing more complex machine learning approaches in several cases, but they still do not capture all of the available improvements found by iterative compilation, therefore, a program-specific approach is still beneficial.

In between program-agnostic and program-specific configuration is the task of finding a small set of configurations for the platform such that, for each program, there exists at least one good configuration. This thesis presents an Integer Linear Programming solution to this problem that improves the state of the art [63] by guaranteeing the optimal reduction of a larger set of configurations. This creates a small set of configurations tailored to the target platform on which a small iterative compilation search can be performed.

Design, development and evaluation of ILP based compiler tuning This thesis explores whether an ILP based method can be developed that automatically discovers relevant program features to learn more accurate models than feature vector approaches (Ch. 6). First a prototype ILP approach that leverages IR is designed, implemented and evaluated against a feature based approach (Ch. 6). The initial approach uncovers some limitations that are addressed in Ch. 7 to create an even more accurate model that provides better performance. These centre around improving the training input for ILP and modifying the way in which the ILP system searches for and evaluates candidate rules. As part of this process, practical limitations are identified and overcome to enable the approach to be applied to large programs.

Analysis of the generality and transferability of compiler tuning approaches Compiler tuning can be applied to a range of platforms, but the model for each platform is usually trained and tested in isolation. Chapter 8 shows that knowledge gained for one platform can also improve performance for a second platform and potentially save the effort of performing a new iterative search and/or constructing a new model for the second platform.

The techniques developed in this thesis are tested on a Neural Network case study, on which program-agnostic optimisation provides the best speed-up and energy improvement, reinforcing the finding that program-agnostic configuration is competitive with more complex machine learning approaches.

In conclusion, the ILP approach developed and evaluated in this thesis outperforms existing feature vector based approaches thus confirming the hypothesis that learning directly from IR increases predictive accuracy.

1.3 Related publications

- C. Blackmore, O. Ray, and K. Eder.

  A logic programming approach to predict effective compiler settings for embedded software.

  Theory and Practice of Logic Programming, 15(4-5):481–494, 2015

  This paper introduces the design, implementation and evaluation of the prototype ILP approach. The work is presented in Ch. 6 with extra detail on design decisions. This paper also includes an early version of the Milepost re-evaluation that appears in Ch. 4.

- C. Blackmore, O. Ray, and K. Eder.

  Automatically discovering human-readable rules to predict effective compiler settings for embedded software.

  Automated Reasoning Workshop, 2017

  This extended abstract provides an interim update to the ILP approach by focusing the generation and processing of training inputs. Chapter 7 builds on this work.

- C. Blackmore, O. Ray, and K. Eder.

  Automatically tuning the gcc compiler to optimize the performance of applications running on embedded systems.

  arXiv:1703.08228 [cs.DC], 2017

  This paper demonstrates the benefit of constructing program-agnostic configurations for embedded platforms. The work also appears in Ch. 5. The paper also fixes a conceptual flaw in the design of BEEBS and provides an comparison of the RIC and CE methods – these contributions appear in Ch. 3. Although the paper is is not published in a peer-reviewed venue, the authors feel the results are worthy of dissemination.

- C. Blackmore, O. Ray, M. Kull, M. G. Rahman, P. Flach, and N. Lachiche.

  Reframing of Classification and Regression Tasks for Predicting the Effects of Compiler Settings on Multiple Embedded Systems.

  In *2nd International Workshop on Learning over Multiple Contexts*, 2015

  This paper gives an introduction to the challenges of targeting multiple platforms with compiler tuning. It inspired work in Ch. 8.

- K. Georgiou, C. Blackmore, S. Xavier-de Souza, and K. Eder.

  Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption.

In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, pages 35–42, New York, NY, USA, 2018. ACM

This paper targets optimisation sequences in Clang/LLVM. It shows how performance can be improved significantly by applying fewer of the optimisations defined in an existing standard optimisation level while preserving their original ordering. Specifically, the approach searches for the optimal point at which to stop applying the sequence of optimisations in `-O2`. The work falls outside the scope of this thesis, although it defines an interesting problem that the techniques in this thesis could be applied to in future.

1.4 Thesis structure

The rest of this thesis is structured as follows:

Chapter 2 – Background This chapter provides the background necessary to understand the scope of this thesis and the wider context of this research area. First there is a review of the various approaches to compiler tuning with particular focus on iterative compilation and predictive approaches. Then there is an introduction to Inductive Logic Programming and its contrast to propositional machine learning techniques that are used by most existing predictive compiler tuning approaches. Finally, there is a summary of the hardware platforms targeted in this research and the benchmarks used to measure the impact of different configurations.

Chapter 3 – Quantifying the potential benefits of compiler tuning To motivate the exploration of compiler tuning techniques, this section investigates the potential benefits available that compiler tuning offers on a range of platforms and programs. Iterative compilation is used to intensively search for configurations that improve execution time, energy consumption and/or code size on the experimental setup described in this chapter. The advantages and disadvantages of two iterative compilation approaches are analysed. The exploration of the compiler tuning space provides a lower bound on the available improvements for techniques developed in later chapters that, once trained, aim to quickly find a good configuration for a given program.

Chapter 4 – Re-evaluation of predictive compiler tuning approaches Two state-of-the-art machine learning compiler tuning approaches are evaluated on the two target platforms of this thesis using a much more diverse set of benchmarks than the original studies. This highlights the importance of a representative data set and a carefully designed evaluation methodology. The results provide comparative data for the methods developed in this thesis.

Chapter 5 – Constructing new program-agnostic optimisation levels This chapter seeks to find program-agnostic configurations that outperform GCC's `-O3` optimisation level. These program-agnostic configurations offer a convenient alternative to the default optimisation levels available in GCC. The fact that these program-agnostic configurations do not capture all of the potential benefits seen in Ch. 3 confirms a predictive approach is indeed required to target programs directly, as explored in the next chapter (Ch. 6).

Chapter 6 – Applying ILP to compiler tuning A new ILP-based approach for predicting effective configurations is designed, developed and evaluated. In contrast to previous work, ILP aims to exploit a relational intermediate representation of programs to relate program structure to effective compiler flags. The method outperforms existing propositional approaches and produces human-readable rules that explain why particular flags should be enabled or disabled.

Chapter 7 – Focusing ILP based compiler tuning Additional analysis of the ILP approach identifies some limitations that are overcome to produce an even more effective methodology by focusing aspects of the training phase.

Chapter 8 – Transferability of compiler tuning methods and models across platforms and to a real world application Compiler tuning techniques can be applied to a wide range of platforms and programs but each platform is usually tested in isolation. This chapter explores how well each approach performs on different platforms and benchmark suites and provides insight into which methods work well in different scenarios. The transferability of a model from one platform to another is also tested.

Finally, a key part of this chapter demonstrates that the advantages seen on benchmark programs in the previous chapters are applicable to real world applications. A case study on the CM3 shows how a Neural Network classifier can be sped-up by 8% on the Cortex-M3 using the program-agnostic configuration created in (Ch. 5).

Chapter 9 – Conclusion The impact of the thesis contributions are summarised in the conclusion chapter and directions for future work are recommended.

Chapter 2

Background

This thesis combines the research area of compiler tuning with the declarative machine learning technique of Inductive Logic Programming. Compiler tuning is the process of fine-tuning compiler settings to optimise one or more performance goals (e.g. execution time, energy consumption, code size). Inductive Logic Programming aims to infer logical rules that relate specific facts to general rules. Applying ILP to compiler tuning is of particular interest because ILP can directly exploit a relational representation of IR thus retaining important structural information that is not captured by propositional machine learning features used in existing predictive approaches.

The following sections provide the background necessary to understand the scope of this thesis and the wider context of this research area. First there is an introduction to compiler tuning and a review of existing approaches (Sec. 2.1). This is followed by an introduction to ILP (Sec. 2.2) and its contrast to propositional machine learning techniques that are used by most existing predictive compiler tuning approaches. Finally, there is a brief overview of the hardware platforms, benchmarks and measurement techniques that will be used to evaluate compiler tuning techniques in later chapters.

2.1 Compiler tuning

There is great potential for compilers to optimise code by applying different transformations that compiler writers have invested considerable effort in developing. Applying the correct set of optimisations is crucial to performance but choosing this set is challenging due to the larger number of flags available and often not well understood interactions between them [19, 57]. Furthermore, the best configuration is dependent on the program and target platform. The challenge increases as new optimisations are added to the compiler, for example, 50 optimisation flags have been added to GCC over the last three years.[1]

2.1.1 Problems in compiler tuning

Two major problems in compiler tuning are choosing which optimisations to apply (the *optimisation selection problem*) (Sec. 2.1.1.1) and selecting the order in which to apply those optimisations (the *phase ordering problem*) (Sec. 2.1.1.2). Tackling either problem can significantly improve one or more compiler tuning goals (e.g. execution time, energy consumption and/or code size) (Sec. 2.1.1.3); most work has focused on the optimisation selection problem [8].

2.1.1.1 Optimisation selection problem

The aim of the optimisation selection problem is to find the set of optimisations that provide the best performance for a given program. The amount of control over optimisations is dependent on the compiler. Each optimisation is applied during a particular stage of compilation, called a *pass*, but not all compilers allow passes to be manipulated directly, instead they can be controlled indirectly via command-line flags. The mapping between flags and passes is complex and often undocumented.

The main mechanism for controlling fine-grained optimisations in GCC is through binary flags that may enable or disable one or more optimisation passes and parameterised flags that control numerical settings for optimisations (e.g. unroll factor, loop tiling size). This thesis focuses on binary flags. Each flag begins with `-f` and most have an opposite version beginning with `-fno` that disables the optimisation (e.g. function inlining can be switched on using `finline-functions` and switched off using `-fno-inline-functions`).

There are additional flags that directly enable or disable passes (`-fenable-kind-pass` and `-fdisable-kind-pass`) but these are only intended for use by compiler developers to debug the compiler and using them can easily produce invalid combinations due to functional dependencies between passes and untested/unsupported combinations. In production, users are expected to use the normal `-f` and `-fno` flags [69]. Regardless of whether flags or passes are manipulated, the user is unable to change the order in which passes are applied; this is determined internally by GCC's pass manager.

An exhaustive search of the optimisation space is infeasible due to the large number of optimisations available. For example, GCC 4.9.3 has over 150 flags that can be enabled or disabled. This results in over 2^{150} possible configurations which would take over 10^{36} years to search exhaustively supposing optimistically that it takes a second to compile and run a program with a single configuration.

[1]This figure is based on the flags listed by `gcc -help=opt -Q` in GCC 4.9.3 (May 2015) and GCC 8.0.1 (Mar 2018).

2.1.1.2 Phase-ordering problem

The phase-ordering problem aims to find the optimal sequence of optimisations to apply. In theory, the configuration space is infinite as passes can be applied any number of times, although the benefits are likely to diminish after a while. In contrast to GCC, the LLVM optimiser `opt` gives the user direct control over the selection and ordering of optimisation passes.

Techniques addressing the optimisation selection problem can be applied to a wider range of compilers than the phase ordering problem as more compilers offer control over optimisation selection than ordering.

2.1.1.3 Compiler tuning goals

Compiler tuning can be used to target any goal that a) can be measured and b) can be influenced by the compiler. Most work has focused on improving execution time, code size and/or compilation time, including multi-objective goals that seek to improve a combination of these metrics. Recently, there has been increased interest in optimising energy consumption which is particularly important on battery powered embedded systems (e.g. IoT devices) but only a few works in compiler tuning focus on energy [57, 32].

2.1.2 Standard optimisation levels

Modern compilers provide standard optimisation levels which enable a predefined set of optimisations. GCC provides `-O0`, `-O1`, `-O2` and `-O3` which enable an increasing set of optimisations at the expense of code size [69]. There is also `-Os` which is similar to `-O2` except it disables optimisations expected to increase code size. Finally, `-Ofast` applies additional optimisations to `-O3` that do not conform to industry standards (e.g. IEEE floating point) and therefore compromise precision and reproducibility. Although these optimisation levels are convenient for the user, better settings can often be found with extra effort (Sec. 2.1.3.1 below).

2.1.3 Iterative compilation

Early work on compiler tuning focused on iterative compilation methods [43] that compile a target program with several different compiler configurations and evaluate the performance of each resulting compilation in order to find a good one. This is a time-consuming task that must be repeated for each new program and platform combination but in practice it yields significant gains. There are several approaches for selecting which configurations to test in iterative compilation; these can be split into the following three categories:

1. No prior knowledge (RIC [27, 10, 3], Fractional Factorial Design (FFD) [57]) – all configurations tested are chosen before running the experiments.

2. Feedback directed (CE [59], Optimisation Space Exploration (OSE) [71], Statistical Selection (SS) [61], Genetic Algorithm (GA) [3]) – the results of previous iterations are used to direct the search.

3. Machine learning guided [18, 10]) – a trained machine learning model is used to bias the configuration selection at each iteration.

Techniques in the first two categories can be used on any new program, do not require a model to be learned but are time-consuming as they select and evaluate a large number of configurations. The third category represents an overlap between iterative and predictive methods and will be discussed in Sec. 2.1.4.

Most iterative compilation studies have targeted the performance of individual programs (Sec. 2.1.3.1). The task of creating new optimisation levels that improve the performance of a wide range of programs has received less attention (Sec. 2.1.3.2), but as Ch. 5 concludes, solutions to this problem are competitive with state-of-the-art machine learning based approaches.

2.1.3.1 Program-specific iterative compilation

The simplest approach, *Random Iterative Compilation* (RIC), uses straight-forward random sampling of compiler flags to construct a set of configurations for evaluation. Fursin et al. reported an average speed-up of 15% on an ARC embedded processor and 33% and 40% on Intel and AMD general purpose processors respectively [27]. More advanced methods use knowledge from previous iterations to guide the search and decide which configuration to try next, however, existing studies conclude that RIC often outperforms them [3, 18].

Cooper et al. [22] used a genetic algorithm to search the optimisation space for sequences that reduce code size. Based on the GA results, a single optimisation sequence was constructed manually that performed within 6% of the best sequence found by GA on each program. Running the GA separately for each module in the program gave up to 2% improvement over using the same sequence for the whole program. The GA approach still requires a new search for each program, therefore, like other iterative methods it is time-consuming. Agakov et al. [3] later used a GA to optimise for execution time and found that RIC performed better overall.

Another approach called *Combined Elimination (CE)* seeks to analyze the effect of each flag relative to an initial baseline, which has all flags enabled, and continually updates the baseline by disabling the flag that has the largest negative impact on performance. As CE is used later in this thesis, a brief description of the CE algorithm presented by Pan et al. [59] is given as follows. The algorithm uses Relative Improvement Percentage (RIP) to measure the impact of a given flag in relation to a given configuration and target program. Let $F_1, F_2, ..., F_n$ be the set of available compiler flags. The impact of flag F_i relative to the baseline configuration B is calculated by the following:

$$RIP_{B(F_i)} = \frac{T_{B(F_i=0)} - T_B}{T_B} * 100$$

where T_B is the execution time of the target program when compiled with configuration B and $T_{B(F_i=0)}$ is the execution time given by the same configuration with flag F_i disabled. The algorithm proceeds as follows:

1. Let $S = \{F_1, F_2, ..., F_n\}$ be the optimisation search space and $B = \{F_1 = 1, F_2 = 1, ..., F_n = 1\}$ be the baseline configuration with all flags enabled.
2. Calculate the $RIP_{B(F_i)}$ for each flag $F_i \in S$.
3. Let $X = \{X_1, X_2, ..., X_m\}$ be the set of flags with negative RIPs sorted in ascending order such that X_1 has the most negative RIP.
4. If $X = \emptyset$ then terminate with B as the final configuration.
5. Remove X_1 from S and X and let $B = B(X_i = 0)$.
6. For $i = 2$ to m recalculate $RIP_{B(X_i)}$ and if $RIP_{B(X_i)} < 0$ remove X_i from S and X and let $B = B(X_i = 0)$.
7. Goto step 2.

Pan et al. [59] showed that CE outperforms two other iterative compilation approaches called Optimization-Space Exploration (OSE) [71] and Statistical Selection (SS) [61]. OSE searches a subset of the optimisation space and iteratively combines good performing configurations; and SS aims to identify whether each flag has a positive or negative effect for the target program and enable/disable them accordingly. Later, Cavazos et al. compared RIC and CE and concluded that RIC outperformed CE. Chapter 3 reaches the opposite conclusion, albeit on a different experimental setup, and the strengths and weaknesses of the two approaches are discussed.

A small number of studies have used iterative compilation to reduce energy consumption. Gheorghita et al. [32] used the Wattch simulator [17] to model energy consumption in their iterative compilation experiments which tested different parameters for loop unrolling and tiling optimisations. The study concluded that iterative compilation is also beneficial in reducing energy consumption. Configurations that improved runtime also often reduced energy consumption, but there were several cases for which this was not true. In addition, time efficient configurations were more likely to also be energy efficient than vice versa.

Pallister et al. [57] used iterative compilation to quantify the effects of individual flags on the energy consumption of embedded platforms including the CM3 and CA8 using 82 flags from GCC 4.7 and an early version of BEEBS which contained ten benchmarks. Fractional Factorial Design [33] was used to systematically select a subset of the optimisation space intended to capture interactions between flags. The study found a set of five flags for the `fdct` benchmark running on the CM3 that improved both time and energy but in disproportionate amounts. On the CA8, execution time often decreased more then energy consumption. One exception was noted where enabling `-ftree-vectorize` on `2dfir` improved energy but not execution time.

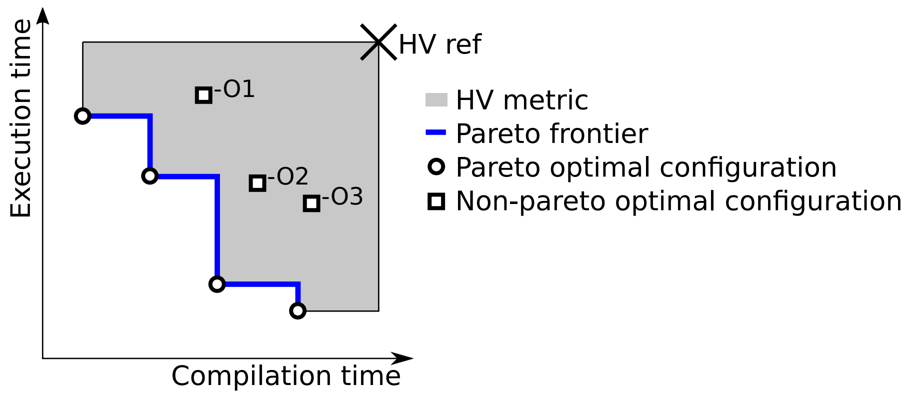


Figure 2.1: HV metric

2.1.3.2 Program-agnostic iterative compilation

Iterative compilation is usually applied to a single target program. In contrast, Hoste et al. [38] proposed Compiler Optimisation Level Exploration (COLE) to find new optimisation levels by changing the goal of iterative compilation to optimise the average performance of a benchmark suite rather than the performance of a single program. A genetic algorithm was used to search the configuration space with the multi-objective goal of reducing compilation time and execution time. The approach produced a set of eleven *Pareto optimal* configurations (a Pareto optimal configuration dominates all other tested configurations on at least one of the objectives). Of the eleven configurations, the first provides the best overall compilation time and the last gives the best overall execution time.

Overall, the majority of improvements were for compilation time rather than execution time. The best configuration for execution time made a modest 3% improvement over `-O3` on GCC 4.1.2 but with a 38% faster compilation time.

A set of Pareto optimal configurations, known as a *Pareto frontier*, is typically evaluated using the hypervolume (HV) metric which measures the objective space covered in excess of the *HV reference* (often selected as the worse performance observed for any configuration). In COLE, the HV metric of the eleven new optimisation levels was compared to that of the existing `-O1`, `-O2` and `-O3` levels.

At benchmark level, only the HV metric (Fig. 2.1) was reported, which confounds the impact of each individual configuration and obscures whether the improvements are for execution time or compilation time. It is also not clear whether some programs performed significantly poorly under the COLE optimisation levels.

COLE found several programs that were insensitive to program-specific optimisation; in these cases, targeting the program directly with iterative compilation offered little improvement over the program-agnostic configurations. There is no existing comparison to quantify the additional benefit that program-specific machine learning models offer over constructing program-agnostic optimisations. The latter has been largely overlooked in this domain and is addressed in Ch. 5.

Purini et al. [63] aimed to downsample a large set of randomly sampled configurations to a reduced set that contains at least one good configuration for each benchmark. Once generated, the reduced set can then be used for a small iterative compilation search on new programs. The study targeted the phase-ordering problem on LLVM and defined a good configuration as performing better than `-O2`. Each of the three methods tested had the primary objective of covering the most benchmarks with the reduced set (a benchmark is covered if the reduced set contains at least one good configuration for that benchmark). Such an objective treats each program as equally important and as a consequence, the approach may focus too much on programs with small available gains and not enough on those with higher potential for improvement. Chapter 5 proposes an Integer Logic Programming solution to this problem that, in contrast to [63], guarantees the best k reduction of the initial pool of configurations, with the primary objective of improving the overall performance of the benchmark suite.

2.1.4 Machine learning

Due to its time-intensive nature, it is clearly infeasible to use full iterative compilation every time a programmer wants to compile a new program. This motivated other studies to use iterative compilation data to train machine learning based approaches that seek to predict a suitable configuration to optimise a given target program. Typically, these methods train a model that takes an input describing characteristics of the target program and outputs a predicted configuration. These methods exhibit a trade-off between the time taken to find a solution and the quality of that solution.

In the *training phase* a model is learned that correlates program features to effective configurations. In the *deployment phase* the model predicts k configurations for the target program, called *k-shot prediction*. When $k = 1$, a single configuration is chosen. When $k \geq 2$, iterative compilation must be run on each predicted configuration to determine the best one, thus compromising on search time to reach further improvements. The latter mode uses machine learning to guide the iterative compilation search and has been shown to find good configurations faster than naive random iterative compilation [3, 18]. The stronger the model, the lower k should be required to achieve good performance.

Many predictive compiler tuning approaches rely on feature vectors of statistical aggregates that summarize characteristics of the target program code. These methods seek to correlate program features with effective configurations but finding the most relevant features is non-trivial (Sec. 2.1.4.4). Features are typically chosen by intuition or onerously generating and evaluating different candidates. The optimal feature set may also depend on the platform and tuning task at hand. The ILP method introduced in this thesis embeds feature construction into the machine learning process.

The rest of this section summarises various machine learning approaches to compiler tuning including 1-nearest-neighbour and decision trees (Sec. 2.1.4.1); bayesian networks (Sec. 2.1.4.2) and neural networks (Sec. 2.1.4.3). Finally there is an overview of existing methods for constructing and selecting features for predictive compiler tuning (Sec. 2.1.4.4).

2.1.4.1 1-nearest-neighbour and decision trees

Fursin et al. [27] explored 1-nearest-neighbour (1NN) and decision tree methods for reducing the execution time, code size and compilation time of programs. They created Milepost GCC which extracts features from programs for use in machine learning. The results of leave-one-out cross validation on an embedded ARC processor showed average speed-ups of 11% for 1NN and 5% for decision trees.

The study was based on 22 benchmarks from the cBench suite [1]. Training data were produced by using iterative compilation to evaluate the execution time, code size and compilation time of 1000 random configurations of 88 optimisations from GCC 4.4. Each program was flattened into a feature vector consisting of 56 features, extracted by Milepost GCC, which describe various aspects of the code such as number of basic blocks and number of conditional branches.[2]

The 1NN method finds the nearest training program T to the test program S based on the euclidean distance between their feature vectors (which are normalised by the number of instructions (feature 24)). Then one of two methods is used to select a configuration. In 1NN-best, the configuration which gave the best execution time for T during training is predicted for S. In 1NN-prob, a probability distribution is produced for the nearest neighbour by selecting the set of configurations that performed within 5% of the best found configuration and calculating the frequency that each flag appears in that set.

In more detail, the 1NN-prob prediction is the Maximum Likelihood Estimation (MLE) for the probability density function given in Eq. (2.1) (originally published in [27]). The probability function is calculated in the same way as the earlier Independent Identically Distributed (IID) approach in [3], the main difference is that the earlier work samples iteratively from the distribution whereas Milepost uses the MLE for its 1-shot prediction.

$$P(x|T^j) = \prod_{i=1}^{L} P(x_i|T^j) \tag{2.1}$$

The Milepost study concluded that 1NN-prob performed better than 1NN-best and explains that this could be because 1NN-prob is less sensitive to flags that appear in good configurations by chance that may have a negligible impact on the training program, but a more significant effect on the test program. Chapter 4 also concludes that 1NN-prob performs better than 1NN-best and the decision tree method.

The decision tree method (referred to as the 'transductive machine learning model' in [27]) frames the tuning problem differently to 1NN. Rather than predict the best set of flags, the

[2] A full list of the Milepost features is available in [27].

goal is to output the probability that a given feature vector and configuration will produce a speed-up within 95% of the best performance found by iterative compilation. For each program and configuration, the training input concatenates the feature vector with a vector of binary variables denoting whether each flag in a configuration is enabled or disabled. In deployment, the decision tree can evaluate different candidate configurations against the target feature vector and choose the configuration most likely to give at least 95% of the available speed-up.

Of particular interest is the Datalog-encoded IR that Milepost extracts and uses to calculate its features vector. This thesis seeks to learn directly from the Milepost IR, which retains more information than flattening programs into feature vectors. Moreover, the IR is based precisely on the internal representations over which the GCC optimisations themselves operate [29].

The Milepost IR consists of 48 predicates which describe the structure and properties of the code. The control-flow is encoded by the following predicates: `bb_p(BB)` defines a basic block[3], `edge_src(E,BB)` defines an edge[4] `E` from basic block `BB` and `edge_dest(E,BB)` which defines an edge to basic block `BB`. The following facts encode an edge `ed0` from basic block `bb0` to basic block `bb1`:

```
bb_p(bb0). bb_p(bb1). edge_src(ed0,bb0). edge_dest(ed0,bb1).
```

As well as describing the structure of the program, the Milepost IR also contains a range of flags that describe properties of specific parts of the program. For example, a single statement may have several flags associated with it such as the following, which means that statement `st0` contains memory operations:

```
stmt_flag(st0,has_mem_ops).
```

In order to apply propositional machine learning, Milepost extracts a feature vector for each function using a set of Prolog rules which analyse the IR to produce counts and averages that describe various aspects of the program such as number of basic blocks with a single successor (feature 2), number of conditional branches (feature 20) and number of calls with pointers as arguments (feature 42). A complete list of the 56 features is given in [27].

The following code shows how feature 2 is calculated using two auxiliary predicates `edge_src_pr2/2` (which counts the number of edges N whose source is at basic block B) and `edge_src_pr2_sel1/1` (which determines whether B has exactly one successor):

```
edge_src_pr2(B,N)     :- bb_p(B), findall(E,edge_src(E,B),L),
                         count_lst(L,N).
```

[3]A basic block is a sequence of instructions with a single entry point (the first instruction) and a single exit point (the last instruction).

[4]An edge connects a pair of basic blocks if one is the predecessor of the other. For example, if the last instruction of basic block BB0 can be followed immediately by the first instruction of basic block BB1, then BB1 is a successor of BB0 (and BB0 is a predecessor of BB1).

```
edge_src_pr2_sel1(B) :- edge_src_pr2(B,N), N=1.
ft(ft2,N)            :- findall(B,edge_src_pr2_sel1(B),L),
                        count_lst(L,N).
```

Milepost GCC was connected to the Collective Optimisation Database (COD) available at `cTuning.org`. This was a crowd-tuning database that aims to gather experimental data from users worldwide in order to produce a large collaborative training data set. The development and use of cTuning highlighted several issues with obtaining collaborative and reproducible data that led to the production of successors `cMind.org` and the latest `cKnowledge.org` [28]. Although the 1NN-best, 1NN-prob and decision tree methods are evaluated in [27], only the 1NN-best method was implemented in cTuning. Collective Knowledge is more focused on reproducibility, crowd-sourcing and iterative methods.

2.1.4.2 Bayesian networks

The recent COBAYN [10] study developed a Bayesian Network approach for the task of selecting configurations that optimise execution time. Use of the Bayesian Network seeks to capture dependencies between flags by learning conditional probabilities between them. Given a set of features that describe the target program, the learned model outputs a probability distribution from which to predict candidate configurations. The first predicted configuration is always the Maximum Likelihood Estimation (MLE) (i.e. the most likely configuration given the probability distribution) while subsequent predictions are sampled at random using the weights of the probability distribution.

The authors targeted the Pandaboard (which features a Cortex-A9 processor) and 39 benchmarks from cBench [1] and PolyBench [62]. The study tested the method in two modes: 1-shot mode (which tests the MLE configuration only) and 8-shot mode which tests the 1-shot prediction plus seven additional configurations sampled from the probability distribution.

Feature vectors describing each program were supplied in three different ways i) static features from Milepost [27] ii) dynamic features from Microarchitecture-Independent Characterization of Applications (MICA) [37] and iii) hybrid (both static and dynamic features).

The approach was tested on seven flags identified in an earlier paper [19] as having the largest impact on execution time, albeit for a very different platform featuring a dual-core Intel Xeon, which is much more powerful than the Cortex-A9 and has a fundamentally different architecture. Since the study focused on just seven flags, this allowed for an exhaustive search of the 128 possible configurations[5]. However, limiting the flag set limits the available improvements as shown in Sec. 4.3.

The COBAYN study showed that its approach outperformed the IID and Markov (MAR) models presented in [3], the former of which is closely related to Milepost's 1NN-prob Sec. 2.1.4.1

[5]Seven flags that can either be enabled or disabled yield $2^7 = 128$ possible configurations

and the latter seeks to capture dependencies between flags. On average, COBAYN performed worse than `-O3` in 1-shot mode, in fact, it took two to four iterations to start outperforming `-O3` on cBench, and five iterations for PolyBench.

The study also compared COBAYN to a predictive modelling approach [60] that predicts speed-up using dynamic features. They reproduced the approach in their experimental setup and trained models using six different machine learning algorithms configured with a range of parameters. COBAYN outperformed each of these models, with an average 1.56x speed-up in 1-shot and 1.48x speed-up in 8-shot mode compared to predictive modelling.

2.1.4.3 Neural networks

Kulkarni et al. [45] used a technique called Neuro-Evolution for Augmenting Topologies (NEAT) [67] to train a neural network to predict performance enhancing optimisation sequences for the Jikes RVM [4] Java compiler. This approach generates an initial population of neural networks and uses a genetic algorithm to evaluate and evolve new neural networks. Features were chosen based on intuition and the efficiency with which they could be calculated. Sher et al. [65] also used NEAT to learn neural networks that predict optimisation sequences for LLVM.

While Kulkarni et al. targeted a dynamic compiler, the NEAT approach is also applicable to static compilation. Sher et al. used NEAT to learn neural networks that predict optimisation sequences for LLVM and reported improvements in execution time ranging from 5% to 50% [65].

Ashouri et al. [9] proposed the MiCOMP (Mitigating the Compiler Phase-Ordering Problem) framework which simplifies the phase-ordering problem by clustering LLVM optimisation passes into promising sub-sequences that can be used as building blocks to compose longer sequences. A total of 48 passes were clustered into five sub-sequences. One group contained the majority of passes (77%) while the four remaining groups consisted of one to four passes each. Neural network, linear regression and K-star machine learning algorithms were used to predict the speed-ups of sequences generated from the sub-sequences.

2.1.4.4 Feature selection

Feature selection has a substantial impact on the performance of machine learning algorithms: choosing the most relevant features for a given problem should improve accuracy and smaller feature sets can increase the efficiency of training and classification.

The features used by compiler tuning work can be divided into two categories: *static* and *dynamic*. Static features, such as those used by Milepost [27], are extracted from the source code at compile-time and include information like number of basic blocks, edges and different types of instruction. They are inexpensive to compute as they do not require the program to be run, but unlike dynamic features they do not differentiate between cold sections of code that are rarely executed, and hot sections where the program spends most of its time.

Dynamic features are gathered by instrumenting the program for one or more profiling runs to collect runtime features such as number of cache hits/misses, number of each type of instruction executed, number of branch mispredictions. They can capture the effects of runtime inputs that are not known at compile time. On the one hand, this exposes more information about the program but the classifier may become biased to the particular input(s) used in training unless care is taken to ensure a balanced, representative set of training inputs are given for each program – this also increases training time as more inputs need to be tested during the iterative compilation phase.

Static features are more portable than dynamic features since profiling is not always possible on the target platform and the execution time may be prohibitively large to allow the extra profiling run(s). Such long runtimes also prohibit the application of iterative k-shot machine learning approaches. Instrumenting the code may also influence the behaviour of the program and dynamic features can vary between runs.

It is difficult to manually choose the right features for compiler tuning because the problem is complex and not well understood. One approach is to select a large set of features in the hope that it includes enough suitable features for the task, however, irrelevant features may confuse training especially if the machine learner assumes each feature to be equally important (as in the euclidean distance based 1NN approach in [27]). Having large amounts of features also increases the complexity of the learning problem making training and classification less efficient. A more reasonable approach is to run empirical tests to evaluate, build and prune a feature set, which is how the feature vector for Milepost GCC was created [54].

Namolaru et al. [54] devised a relational Datalog encoding of GCC's internal GIMPLE IR from which they computed numerical features (e.g. number of basic blocks and average number of store instructions in a basic block) for use in propositional machine learning. Prolog rules were written to extract numerical features that were incrementally added to the training feature vector until the resulting model achieved a desired accuracy. The final feature vector consisted of 56 features that have been adopted by further studies [27, 68, 10]. A key motivation of this thesis is to applying machine learning directly to the Datalog IR thus retaining structural information rather than downsampling the code into numerical features.

The original work by Namolaru et al. [54] constructed the feature vector on three desktop/server processors (AMD Athlon 64 3700+, Intel Xeon 2.8GHz and Itanium2 1.3GHz) and tested it on the embedded ARC 725D 700MHz processor, thus making the assumption that the set of relevant features is independent of the platform. This assumption is continued in subsequent studies [68, 10] that apply the Milepost feature vector to whatever experimental setup is chosen. An advantage of the ILP approach is that the features are automatically constructed during training and are therefore customised for the target platform.

Genetic algorithms have also been applied to the task of feature construction and selection. Tartara et al. [68] produced a grammar for generating rules utilizing a subset of the Milepost [27] feature vector and applied genetic programming to evolve better rules.

Leather et al. [48] exploited program structure to predict loop unroll factors using genetic programming. In each iteration of the evolutionary search, features were generated from a grammar and used to train a machine learner. The goal of the search was to find the features that maximise the accuracy of the machine learner. The method took two days to train and since it only targets the parameter for one flag it is unclear whether it would scale to the task of tuning with over 100 compiler flags. Investigating this would require substantial effort to replicate the original work.

Li et al. [49] created a GCC plugin which extracts hundreds of program features; they then used a genetic algorithm to identify the features that were best for predicting which optimisations to enable. The study hand-picked 15 features based on previous studies and tested the effect that adding 5 features (generated by their genetic algorithm) had on the performance of a Support Vector Machine (SVM) for predicting which flags to enable. The study showed that adding the 5 new features improved the execution time of 10 out of 15 benchmarks. They also showed that extracting a new feature vector at certain stages during the compilation process further improved predicted configurations for 12 out of 15 benchmarks.

Cummins et al. [23] introduced an end-to-end deep learning methodology, DeepTune, that was applied to two parallel computing challenges: predicting the optimal device on which to run an OpenCL program and selecting GPU thread coarsening factors. Instead of training with conventional features, the source code was normalised and encoded as input to a neural network to determine its own hidden features. Such a technique could also have a positive impact for compiler tuning, but such an investigation is outside the scope this thesis.

Blackmore [11] demonstrated how the Milepost feature vector could be used for compiler tuning in an ILP setting but this work did not utilise the full potential of ILP to learn from structured data instead of a propositional feature vector, as explored in Chs. 6 and 7.

Blockeel et al. explored the orthogonal problem of predicting a target platform's performance based on the execution times of a set of reference programs. The use case is for a user who has an application of interest and wishes to acquire a new target platform that will maximise performance for their application. Interestingly, the feature vector consisted purely of the reference program execution times, which were used to find the most similar 'predictive machine' to which the user does have access. The performance of the application on the closest machine is used to predict the performance of the target platform. Such an approach has limited value in compiler tuning as the model requires to have already run the target program on some other machine in order to make a prediction.

2.1.5 Scope

This thesis focuses on developing a new one-shot predictive approach that exploits intermediate representation to tackle the optimisation selection problem for binary GCC flags that can be enabled or disabled. The order in which optimisations are applied is determined by the compiler's pass manager, as is usual for the optimisation selection problem.

Random Iterative Compilation and CE are used to quantify the potential performance improvements available and produce training data. These methods do not rely on features (unlike predictive approaches) and the algorithms do not have parameters that require tuning (unlike GAs). Moreover, random sampling has already been shown to be competitive with more advanced approaches like CE and GA. As each method serves the purpose of finding significant improvements it is not necessary to try other iterative compilation approaches. Any user of the techniques developed in this thesis can apply their iterative compilation algorithm of choice to the task.

A recent survey of compiler tuning approaches counted 79 papers that target the optimisation selection problem and 24 that target the phase-ordering problem. There is difficulty in comparing all approaches because they were evaluated on different experimental setups (e.g. different compilers, optimisations, processors and/or benchmarks), most do not offer a publicly available working implementation and there is often not sufficient detail to reproduce the methods. As a result there is no consensus as to which method is the best and the problem worsens as more methods are proposed. To this end, this thesis compares its methods to two major works on predictive optimisation selection, Milepost [27] and COBAYN [10] and three iterative approaches; Random Iterative Compilation, Combined Elimination [59] and Purini et al [63]. To compare all published methods is non-trivial task that would require substantial further effort.

2.2 Inductive Logic Programming[6]

The aim of ILP [53] is to infer logical, human-readable hypotheses H that distinguish positive examples E^+ from negative examples E^- with respect to background knowledge B describing the problem domain (Sec. 2.2). More formally, the goal is to learn hypotheses H which, given B, cover all of the positive examples (coverage) and none of the negative examples (consistency), which can be written $B \cup H \models E^+$ and $B \cup H \not\models E^-$.

Inductive Logic Programming is a white-box method that learns declarative rules that can be translated easily into sentences to provide insight into the learned relations. The technique has already made an impact in drug design by identifying sub-molecular structural features, called pharmacophores, that are responsible for medicinal activity [26].

A key advantage of applying ILP to compiler tuning is its ability to exploit relational IR directly. Chapter 6 develops an ILP based compiler tuning approach that learns directly from the Datalog IR developed by [54] and used in Milepost GCC [27]. In contrast to [54] which created Prolog rules to calculate numerical feature aggregates, ILP can automatically construct new features directly from the Datalog IR.

[6]Work in this section also appears in [13].

2.2.1 Motivating example

Inductive Logic Programming can be applied to a problem by encoding the problem and inputting it to an ILP system that constructs and evaluates hypotheses, outputting the best generalisation that was found. This thesis uses CProgol4.4 [52] and Aleph [66] ILP systems, which both operate on a Prolog encoding of the problem.

For example, in terms of compiler tuning, to learn the relation *"Flag F is a good flag for program P (and therefore F should be enabled)"*, the background knowledge B might contain clauses describing structural aspects of each program and E^+ and E^- could include examples of programs for which flag F has a good or bad influence on performance respectively. The following is an example of how such a problem could be encoded in Prolog rules as input for CProgol4.4:

```
B   =  program(prog1). program(prog2). program(prog3).
       flag(flag1). ft(ft1). avg(ft1,0.6).
       ft(ft1,prog1,0.9). ft(ft1,prog2,0.7). ft(ft1,prog3,0.2).
       small_ft(P,Ft) :- ft(Ft,P,N), avg(Ft,Avg), N<Avg.
       large_ft(P,Ft) :- ft(Ft,P,N), avg(Ft,Avg), N>=Avg.
E+  =  goodFlag(prog1,flag1). goodFlag(prog2,flag1)
E-  =  :- goodFlag(prog3,flag1).
```

In addition to this background theory, mode declarations are required to constrain the search space used for generalisation [52]. There are two types of mode declaration: mode head (`modeh/2`) and mode body (`modeb/2`) which define the format of literals that may appear respectively in the head and body of any learned hypothesis. The first term of each mode declaration specifies the recall, which is the number of alternative instantiations permitted for the predicate described in term two. The second term defines the type of terms that may appear in the predicate and whether they are an input variable (`+`), output variable (`-`) or constant (`#`). For this motivating example, the following mode declarations allow learned rules that consist of `goodFlag/2` in the head and `ft/3`, `small_ft/2` and `large_ft/2` in the body:

```
:- modeh(*,goodFlag(+program,#flag))?
:- modeb(*,ft(#ft,+program,#any))?
:- modeb(*,small_ft(+program,#ft))?
:- modeb(*,large_ft(+program,#ft))?
```

The asterisks (`*`) in the above declarations specify an arbitrary recall (which is set to 100 in Progol by default). The `modeh` declaration states that the head of a learned rule may contain `goodFlag/2` with an input variable of type `program` as its first term and a constant of type `flag` as its second term. The first `modeb` states that `ft/3` with a constant of type `ft`, an input variable of type `program` and a constant of any type may appear in the body of hypotheses.

The generalisation process in Progol proceeds by selecting one positive example at a time and first constructing the *most specific clause* that explains the example. This is constructed by placing the positive example at the head and adding to the body, all true literals for which the `modeb` declarations are provable. In this case the most specific clause is as follows:

```
goodFlag(P,flag1) :- ft(ft1,P,0.900), large_ft(P,ft1).
```

Now Progol constructs several hypotheses and scores them based on the number of positive and negative examples they cover and the number of literals in the rule (compression). The aim is to cover as many of the positive examples as possible and none of the negatives. The highest scoring hypothesis is chosen, any positive examples it covers are retracted from the background theory and generalisation continues with the next remaining positive example.

In this case, Progol will return the following hypothesis which states that `flag1` is a good flag for program P if feature 1 of P is large:

H = `goodFlag(P,flag1) :- large_ft(P,ft1).`

In Ch. 6, the background knowledge is augmented to capture additional relations between features and to generalise the information contained within the Prolog-encoded IR in order to allow more complex rules to be learnt when applied to real data.

2.3 Hardware platforms

This thesis targets two embedded platforms featuring popular embedded processors. The first is the STM32VLDISCOVERY embedded system development board which features an ARM Cortex-M3 (CM3) 32-bit processor that is a popular choice of processor for Internet of Things (IoT) platforms [7]. The second is the Beaglebone development board which has an ARM Cortex-A8 (CA8) 32-bit processor that has featured in many mobile devices and is much more complex than the CM3, with a longer, superscalar pipepline and NEON vector floating point unit.

The bare-metal CM3 gives very consistent results as there is no cache or Operating System (OS) to contribute to noise. On the other hand the Beaglebone/ArchLinux setup exhibits more variation due to the OS and the presence of cache. Consistent results are important for compiler tuning experiments as too much noise would mask the actual effect of compiler optimisations and longer experiments would be needed to average away the noise. In the training phase, noise can lead to inaccurate models being learned and in the prediction phase it can affect the evaluation of the model as some of the performance differences could be due to noise. To minimise training data variation each benchmark is looped several times to give a reasonable execution time (~500ms on the CM3 and ~5s on the CA8). As the CM3 produces less variation than the CA8, it is used as the basis for developing the methods in Chs. 5 to 7. Once developed, the methods are then tested on the CA8 to demonstrate the generality of the methods and applicability to other platforms.

Table 2.1: Hardware features of the target platforms

Processor	Platform	Clock speed	Flash	RAM	Pipeline stages	FPU	Superscalar	Cache
Cortex-M3	STM32VLDiscovery	24MHz	128KB	8KB	3	N	N	N
Cortex-A8	Beaglebone	720MHz	8GB	256MB	13	NEON	Y	Y

2.4 Benchmarks

To explore the effects that different flags have on a range of programs, representative benchmarks are required that exercise the hardware in various ways.

This section describes the two benchmark suites used in this study. The first suite, BEEBS, is chosen because it is the largest known collection of open source benchmarks compatible with resource-limited embedded systems. The second suite, cBench, is chosen for comparison with earlier compiler tuning research Ch. 4, but as it requires a file system, it is only compatible with the Beaglebone which runs an OS and not the bare-metal CM3.

2.4.1 Bristol/Embecosm Embedded Benchmark Suite (BEEBS)

This study uses the 84 benchmarks of the Bristol/Embecosm Embedded Benchmark Suite (BEEBS) [58], which to the author's best knowledge is the largest collection of free open source benchmarks available for embedded systems.[7] The benchmarks cover a diverse range of characteristics as demonstrated in Fig. 2.3 and were produced in response to the lack of freely available benchmarks for resource limited bare-metal embedded systems such as the CM3. Other existing benchmark suites have fewer benchmarks and their reliance on an OS and/or file system being present make them incompatible with bare-metal systems. Some of the BEEBS programs were in fact derived and adapted from the MiBench [35], WCET [34] and DSPstone [73] suites.

The BEEBS suite has been used in several recent academic studies [57, 20, 21, 13, 56, 51] and in industry for software performance modeling [72], development of Epiphany GCC for the Parallella project[8] and performance analysis on RISC-V platforms[9].

[7]Six programs are no longer in the master branch as their license could not be confirmed and BEEBS requires all benchmarks to be under GPL.

[8]`http://parallella.org`

[9]`http://orconf.org/#cariscv`

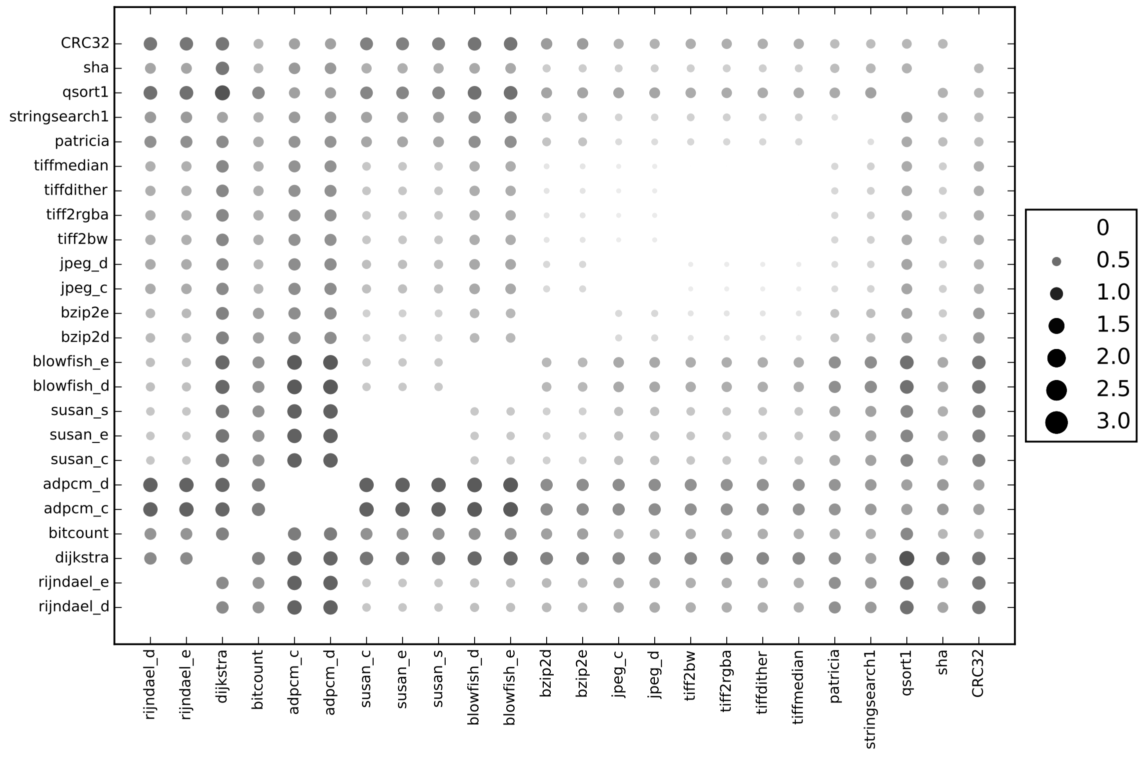


Figure 2.2: Euclidean distance between normalised feature vector of each cBench program. Note: a similar graph appears in Fig. 9 of [27].

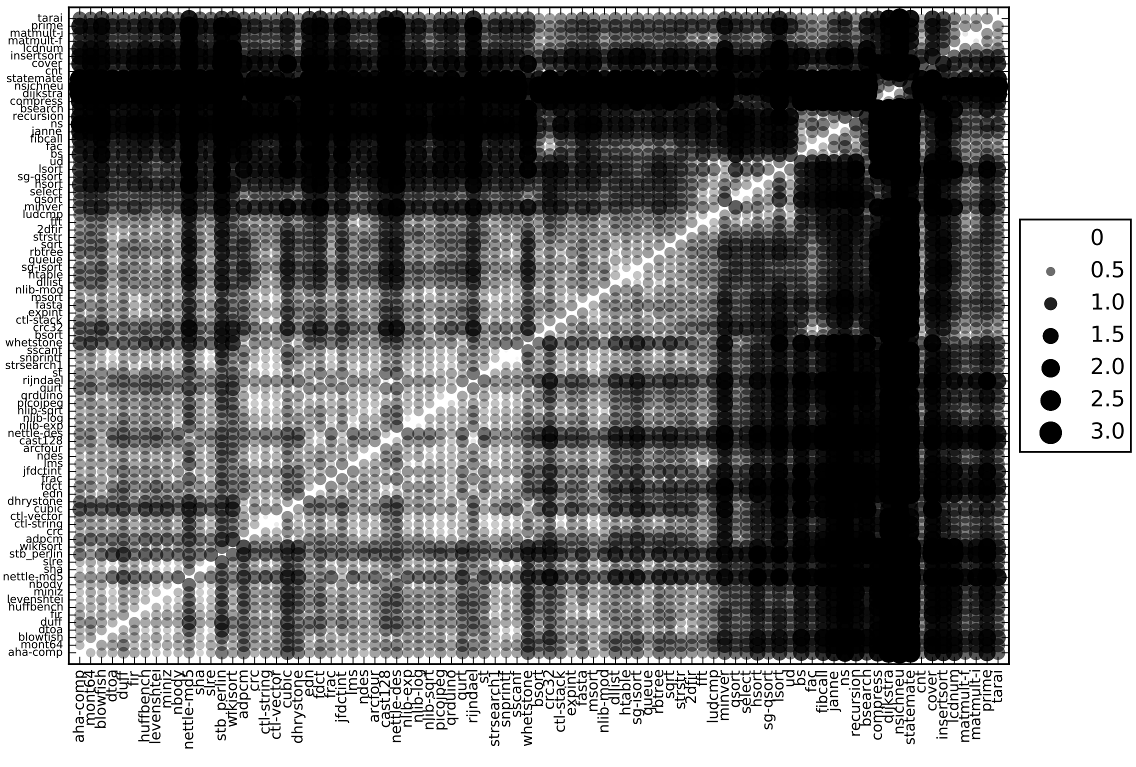


Figure 2.3: Euclidean distance between normalised feature vector of each BEEBS program

Each BEEBS benchmark consists of at least one source file containing the benchmark itself plus another file `main.c` which controls the number of times the benchmark is run according to a repeat factor. The repeat factor is used to produce a runtime long enough to obtain reliable measurements and it also enables BEEBS to target a wide range of platforms which may execute particular benchmarks considerably faster or slower than other systems. Programs that run too fast may need to be looped tens of thousands of times in order to produce a long enough runtime. In these cases, the loop overhead may account for most of the measurement.

Most of the benchmarks require test input data on which to operate. In reality, the input data would not be known at compile-time and would typically be supplied via command-line parameters, data files or an input stream from a device (e.g. sensor). To make a fair comparison between different compilations of the same benchmark, the input data must be fixed. However, BEEBS targets bare-metal embedded systems which have no command line or file handling support for providing input files or parameters, therefore the input is fixed by hard-coding it into the source code (which can be problematic unless some care is taken as explained in Sec. 3.1).

2.4.2 Collective Benchmarks (cBench)

The cBench benchmark suite [1] consists of 32 benchmarks grouped into seven categories (automotive, compression, consumer, network, office, security and telecom). Like BEEBS (Sec. 2.4.1), there is a configurable repeat factor, called a loop wrap, to make the benchmark run for long enough to measure.

This thesis uses 24 benchmarks from cBench, the same set used in COBAYN [10]. The remaining eight benchmarks produced compilation errors which may explain their exclusion from the COBAYN study.

Although cBench is used in several existing compiler tuning studies [27, 10, 68, 9], the small number of benchmarks limits the generality of the evaluation. This thesis uses the BEEBS benchmark suite which contains over three times as many benchmarks and is much more diverse.

An analysis of the feature vector of each program shows that cBench contains clusters of very similar programs (Fig. 2.2), whereas the BEEBS programs are much more distant (Fig. 2.3). On average, the euclidean distance between the feature vector of each program is 0.99 for cBench and 1.50 for BEEBS. The similarity of benchmarks in cBench poses an issue for the cross-validation techniques commonly used to evaluate machine learning based compiler tuning. Cross-validation systematically excludes one or more programs from the training set and uses them to test the model trained by the remaining programs. The two most common approaches are leave-one-out cross-validation (which leaves one program out at a time) and 10-fold cross-validation (which leaves 10% of the programs out at a time). In leave-one-out cross-validation, a 1NN classifier need only find the near-identical remaining program (e.g. to predict for blowfish_e, find blowfish_d) which is likely to benefit from similar optimisations. Even 10-fold cross-validation would only remove two or three programs at a time for cBench. A more suitable approach would be to remove a whole class of benchmarks (this could be called leave-one-class-out cross-validation), as done in MiCOMP [9]. Even so, the sample size is still small and a larger, more diverse benchmark suite is desirable.

Six of the BEEBS benchmarks were derived from cBench (`blowfish`, `crc32`, `cubic`, `dijkstra`, `rijndael`, `sha`), but importantly, where cBench contains encryption and decryption versions of `blowfish` and `rijndael`, only one appears in BEEBS. The effect of benchmark choice on the evaluation of compiler tuning techniques is explored further in Ch. 4.

2.5 Measuring energy and time

This work uses a custom energy monitoring board called the MAGEEC WAND to obtain accurate energy measurements for each execution of the benchmarks. The board was developed by the MAchine Guided Energy Efficient Compilation (MAGEEC) project [2] at the University of Bristol in collaboration with industry compiler experts, Embecosm[10]. The firmware, software and hardware for the energy monitoring board is all open source and publicly available.

The monitoring board is connected to a target board by inserting a shunt resistor inline with the target's power supply. By measuring voltage drop across the shunt resistor, the monitoring board calculates current, power and energy and reports it to a host computer. Energy measurements are triggered automatically from the target board by toggling a GPIO pin at the start and end of the target code to be measured.

An additional benefit of using the MAGEEC WAND is that it also produces very accurate time measurements, with two million samples per second. Timing is very consistent on the CM3 but the CA8 exhibits more variation due to the presence of an OS and cache. To counteract some of the influence from caching, the cache and buffers of the CA8 are cleared before each execution and the CPU frequency is fixed. Each execution is also run with high priority by using a low 'nice' value.

Some previous studies have used energy models [32] but as the model error adds noise to the data, real hardware measurements are preferred.

[10] `www.embecosm.com`

CHAPTER 3

QUANTIFYING THE POTENTIAL BENEFITS OF COMPILER TUNING

This chapter aims to motivate the development of compiler tuning approaches by intensively searching the configuration space to establish a lower bound on the improvements available for the target benchmarks and platforms.

The potential benefits of compiler tuning are investigated on two embedded hardware platforms compatible with bespoke energy monitoring hardware designed in the MAGEEC project [2] by the University of Bristol and industry compiler experts, Embecosm. The first platform is the STM32VLDiscovery which features an ARM Cortex-M3 (CM3) processor and the second is the Beaglebone which features an ARM Cortex-A8 (CA8) processor (Sec. 2.3).

The state-of-the-art open source Bristol/Embecosm Embedded Benchmark Suite (BEEBS) [58] is used to measure the effects of different configurations on each program as it is the the largest and most diverse suite available for targeting resource-limited bare-metal embedded systems such as the STM32VLDiscovery. Many benchmark suites depend on the presence of an OS and/or file system and are therefore not suitable for bare-metal targets. In contrast, the Beaglebone is able to run Linux and can be targeted by both BEEBS and cBench. The use of cBench aids comparison with existing work as the suite appears in many previous compiler tuning studies [27, 10, 68]. As part of this investigatory study, an oversight in the design of BEEBS was identified and fixed to enable a more realistic evaluation of the methods used in this thesis.

The study begins with a brief analysis of standard optimisation levels available in GCC 4.9.3 and their impact on execution time, energy consumption and code size. These optimisation levels provide a quick and convenient way for software engineers to attempt to optimise their code, however, as this chapter shows, they often produce compilations that are far from optimal.

Next, two state-of-the-art iterative compilation techniques, Random Iterative Compilation (RIC) [3] and Combined Elimination (CE) [59], are used to intensively search for more optimal configurations beyond the standard optimisation levels. The biggest improvements were found for time and energy, both of which were strongly correlated across the benchmarks and platforms. There was little room for improvement on CM3 code size while some significant improvements were found for the CA8.

Overall, CE finds better configurations that RIC, and in a shorter amount of time. Interestingly, this contrasts with the findings of a previous paper [18]. Possible explanations are discussed in Sec. 3.5.1, along with the strengths and weaknesses of the two iterative compilation approaches. The original CE algorithm begins with all flags enabled and selectively disables individual flags. Further experiments show that starting CE with a different baseline does not drastically alter the performance of the algorithm.

The data produced in this chapter form the basis for training data in the rest of this thesis as well as giving a measure of the possible improvements available for each benchmark.

3.1 BEEBS data initialisation

As part of this study, an oversight in the design of BEEBS was fixed in order to increase the reliability of the experiments in this thesis and future work. Analysis of preliminary iterative compilation results showed that the compiler was able to over-optimise given knowledge of test input data necessarily hard coded into benchmarks leading to large speed-ups that do not hold when input data is not known in advance. This is a conceptual flaw that might also affect other benchmark suites.

This study modified BEEBS to eliminate cases where it was possible for the compiler to optimise based on input data.[1] This was done by using an `initialise_benchmark` function which initialises any input data required by the benchmark. The `initialise_benchmark` function is defined outside of `main.c`, but is called from within `main.c`. As long as link-time optimisation is disabled, the knowledge that `initialise_benchmark` is called in `main.c` cannot be used in the optimisation of the other source files. Benchmarks for which input data was already given by global variables or arrays did not need adjusting because the compiler cannot assume that the globals are not changed elsewhere in the program.

Over-optimisation led to 1% of overall gains seen in preliminary experiments. Ten of the benchmarks gained over 5% advantage from having input data exposed to the compiler.

For example, the compiler was able to over-optimise `expint` (which calculates exponential integrals) based on constant input data to a key function in the benchmark. With inputs as constants, one configuration reduced the execution of `expint` by 18% compared to `-O3`, but without these inputs available for optimisation the reduction was a smaller 8%.

Except where noted, the rest of this thesis proceeds using the improved version of BEEBS.

[1]The changes are now in the master branch of BEEBS (http://beebs.eu)

3.2 Evaluating configurations

Each configuration x is evaluated against a baseline configuration b to enable direct comparisons across programs. The highest optimisation level, `-O3` is used as the baseline, as is common in compiler tuning work. For example, the execution time for configuration x relative to `-O3` is calculated as follows:

$$r = \frac{\textit{execution time of configuration } x}{\textit{execution time of } \texttt{-O3}} \tag{3.1}$$

The result r can be interpreted as follows: if $r < 1$ then x ran faster than `-O3`. Conversely, if $r > 1$ then x took longer than `-O3`.

3.3 Standard optimisation levels

The standard optimisation levels offer a convenient set of configurations for software engineers to apply to their code (Sec. 2.1.2). They are intended to optimise for execution time and/or code size, but their impact on energy consumption is less well explored.

This section shows how the optimisation levels affect the execution time, energy consumption and code size of each benchmark on the CM3 (Figs. 3.1 to 3.3) and CA8 (Figs. 3.4 to 3.9).

Across each platform `-O3` gave the best overall execution time and energy consumption followed by `-O2` (Figs. 3.1, 3.2, 3.4, 3.5, 3.7 and 3.8). Unsurprisingly, `-O0` performed worst overall as it applies the fewest optimisations. On average `-Os` and `-O1` fall between `-O0` and `-O2`. In very few cases `-O2` outperforms `-O3`, but `-O2` often performs significantly worse and up to four times slower. One further observation is that the averages for `-O1`, `-O2` and `-O3` are much closer together on cBench than BEEBS.

On the CM3, code size was very similar for each optimisation level except on the `compress` benchmark where `-O1`, `-O2` and `-Os` produced compilations that were almost half the size of `-O3` (Fig. 3.3). There was more variation on the CA8 where `-O1`, `-O2`, and `-Os` gave the smallest average code size, while `-O3` produced larger sizes closer to `-O0`.

In summary, these results show that the progressive optimisation levels do indeed increase average speed-ups as intended and they also increase energy efficiency. Conversely, `-Os`, which is supposed to optimise for code size, does give the smallest code size on cBench, but is beaten by `-O1` and `-O2` on BEEBS.

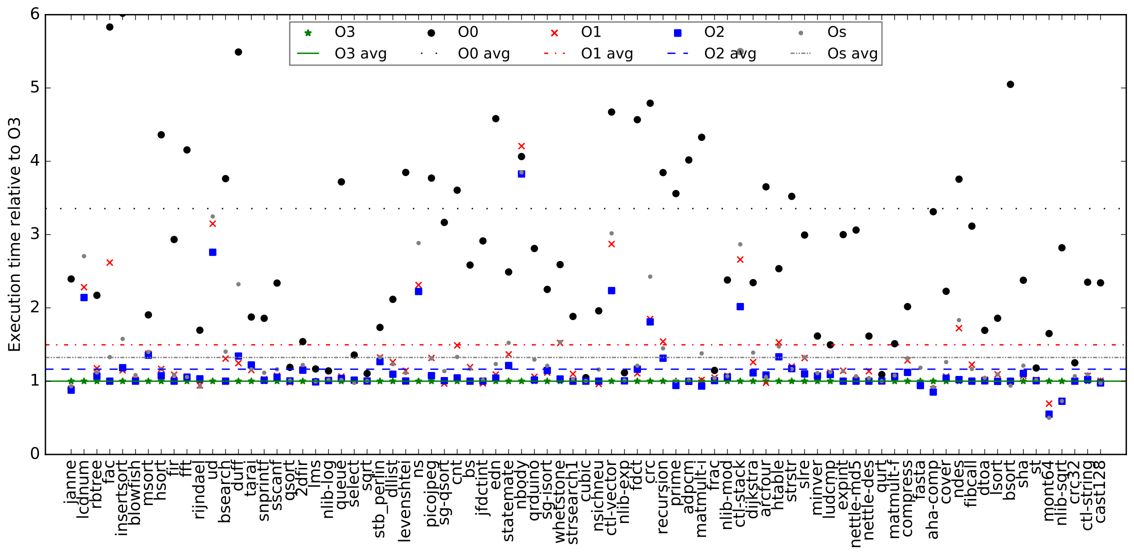


Figure 3.1: Execution time of standard optimisation levels (CM3 BEEBS)

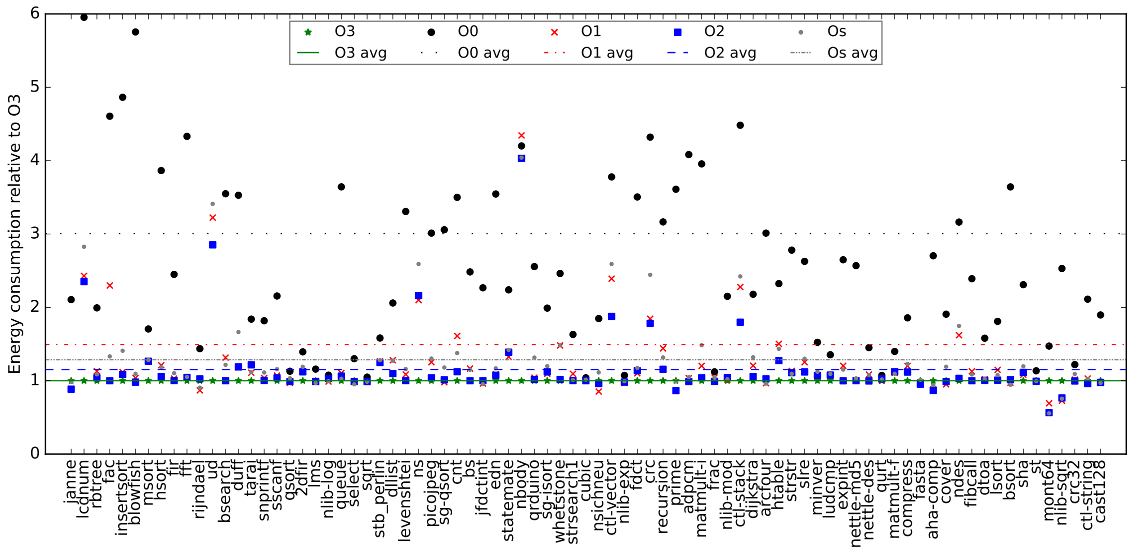


Figure 3.2: Energy consumption of standard optimisation levels (CM3 BEEBS)

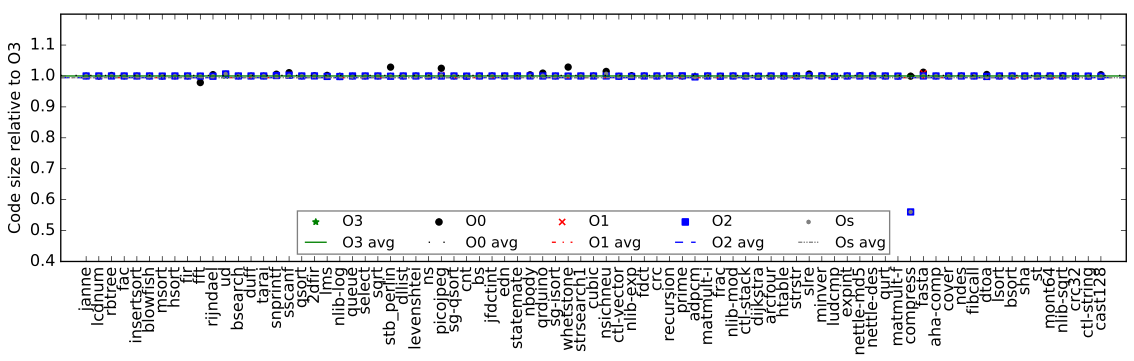


Figure 3.3: Code size of standard optimisation levels (CM3 BEEBS)

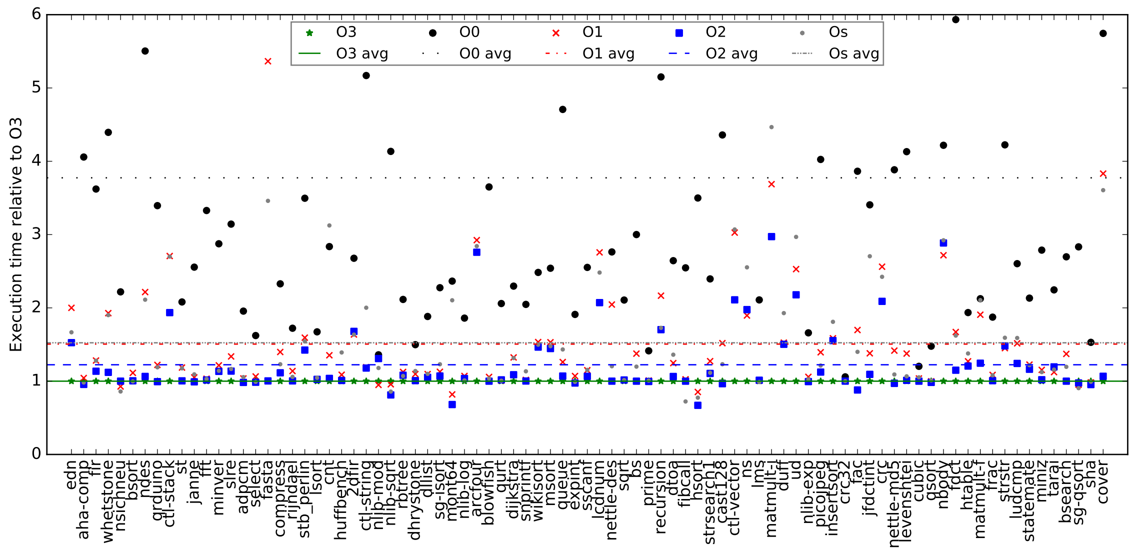


Figure 3.4: Execution time of standard optimisation levels (CA8 BEEBS)

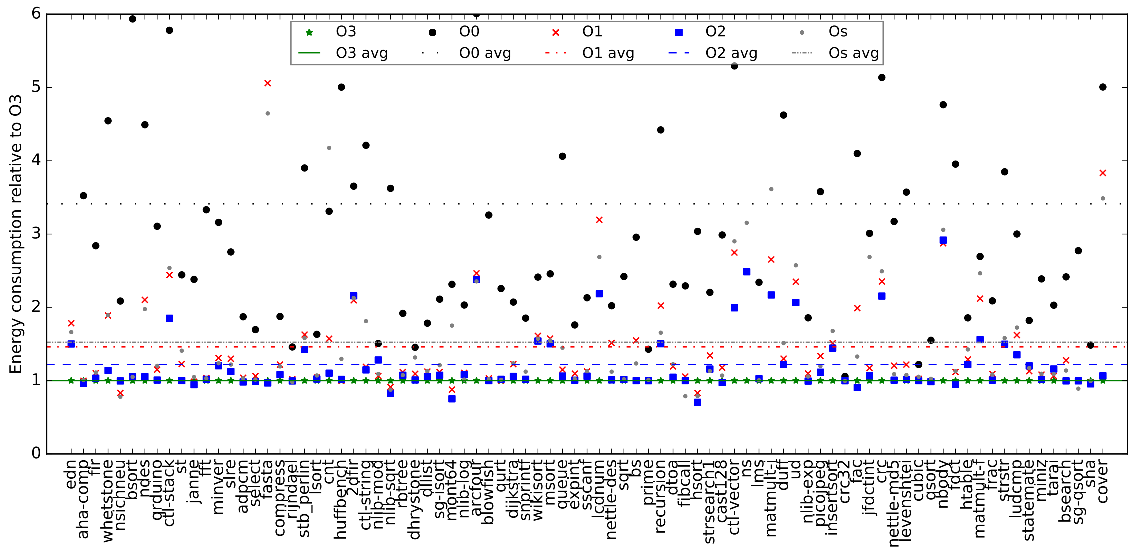


Figure 3.5: Energy consumption of standard optimisation levels (CA8 BEEBS)

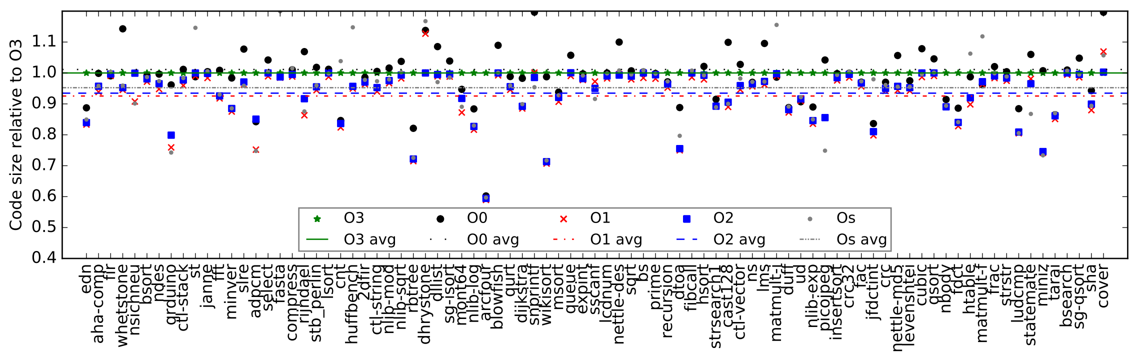


Figure 3.6: Code size of standard optimisation levels (CA8 BEEBS)

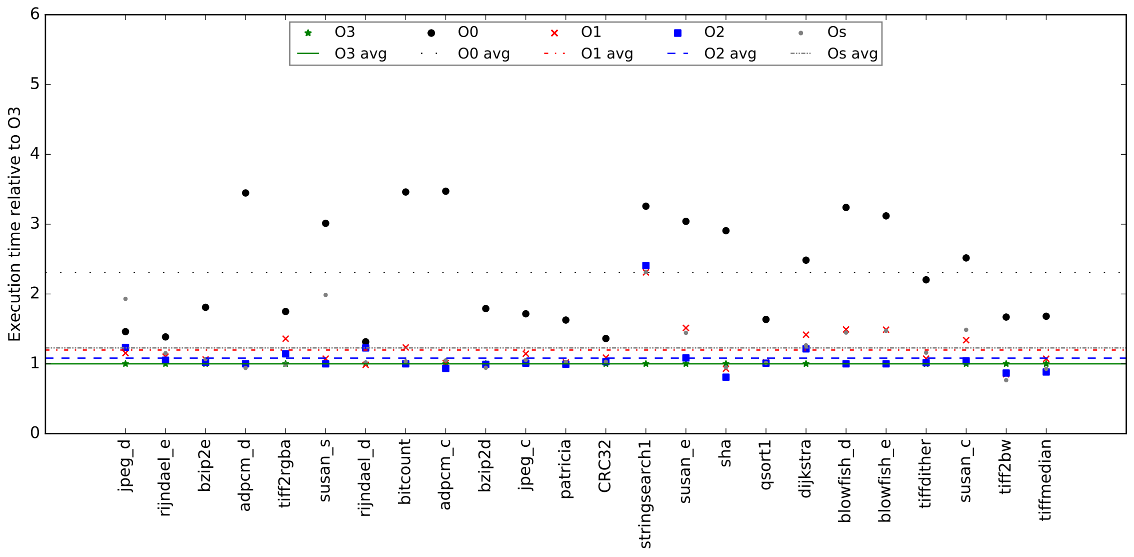


Figure 3.7: Execution time of standard optimisation levels (CA8 cBench)

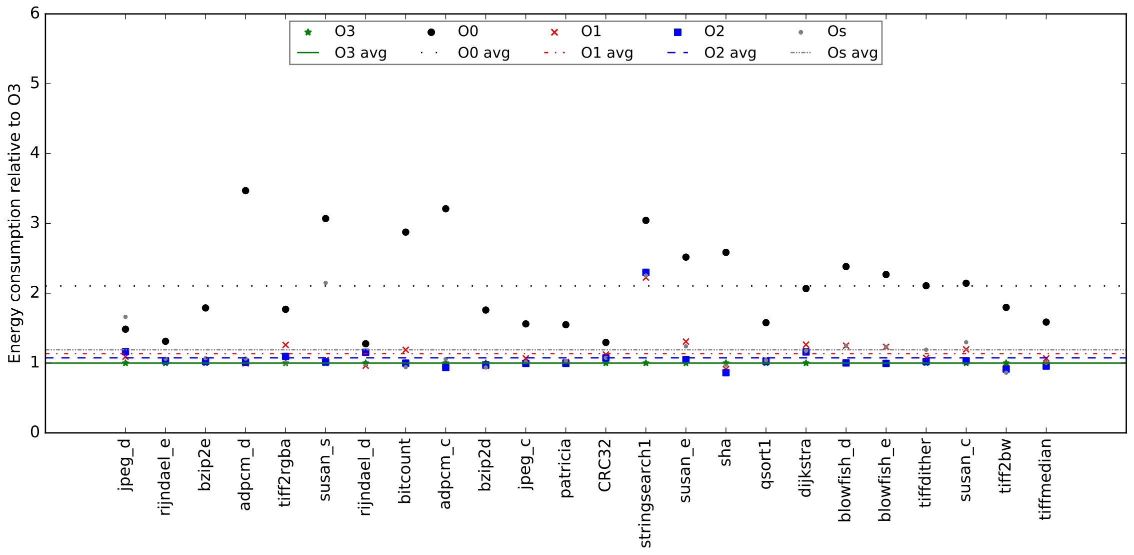


Figure 3.8: Energy consumption of standard optimisation levels (CA8 cBench)

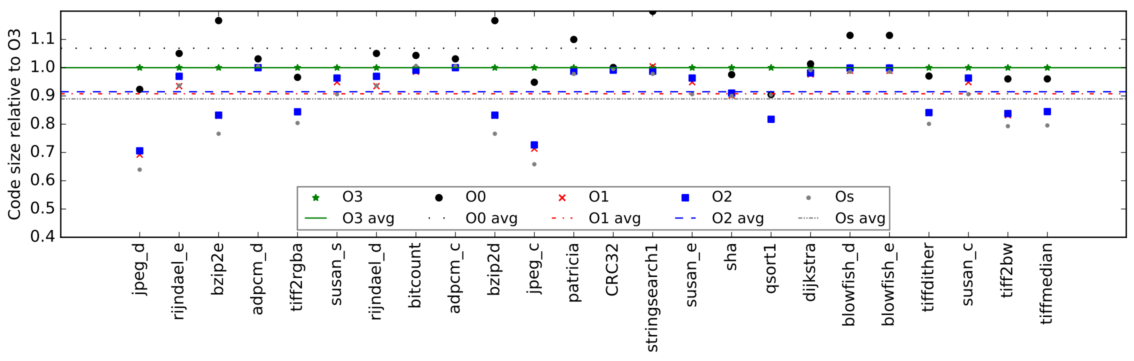


Figure 3.9: Code size of standard optimisation levels (CA8 cBench)

3.4 Analysing the configuration space with Random Iterative Compilation

To search for potential gains on the target platforms, this study focuses on 133 flags available in GCC 4.9.3. This includes 26 flags not enabled by -O3 and excludes flags that do not follow standards, reduce precision, require additional profiling information or are purely intended for C++ or debugging. The set of flags is fixed for all experiments in this thesis to allow for consistent comparisons. Unlike some previous works [10, 38], this set includes flags that are enabled at `-O0` (thus not assuming that such flags are always good) and other flags that are not enabled even at `-O3` (thus avoiding missing out on potential improvements from non-O3 flags).

For RIC, a random sample of 1000 configurations was generated by selecting `-O1`, `-O2` or `-O3` with probability $p(\frac{1}{3})$ and enabling each flag with probability $p(\frac{1}{2})$. A base optimisation level (`-O1`, `-O2`, `-O3`) is selected for two reasons: firstly, many flags are ignored by GCC unless `-O1` or higher is selected and secondly, some optimisations cannot be controlled by flags and this aims to capture the effects of such optimisations in the data set.

To improve the efficiency of the searches, the md5 hash of each compiled binary is stored along with its performance measurement. If any future compilation has the same hash, then the previously cached performance is used rather than re-executing the binary. In general, this has more benefit on CE where there is a higher similarity between configurations since each newly tested configuration differs to the current baseline by exactly one flag.

3.4.1 Distribution of results

Figures 3.10 to 3.12 show the distribution of results for each configuration on each metric for the two platforms. Raw measurements are on the left-hand side and results relative to `-O3` are on the right-hand side.

On many of the benchmarks it was possible to significantly improve upon `-O3` for time and energy by up to 80%. The average reduction in time and energy ranges between 13-15% across the platforms and benchmarks as shown in Table. 3.1. Code size remained largely unchanged on the CM3 except for `compress` where it could be greatly reduced (as already seen when using `-O2` instead of `-O3` (Sec. 3.3)) and `fir`, `stringsearch1` and `dtoa` for which code size was greatly *increased* by some configurations. The CA8 showed more variation with some code sizes being reduced by up to 50% and on average 9% for BEEBS and 12% for cBench.

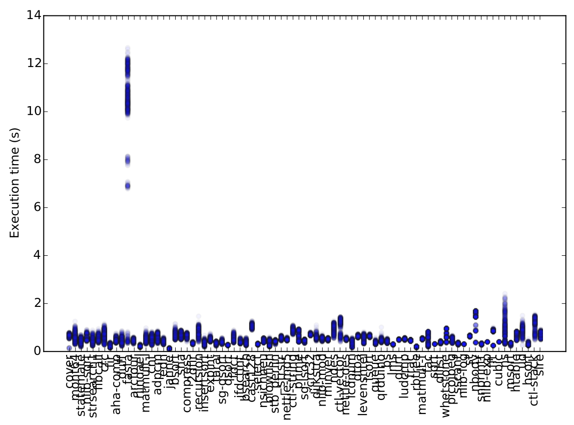
On several benchmarks for which the execution time or energy consumption of `-O3` could be greatly improved upon, such as `statemate` and `crc`, there were very few configurations close to the best found configuration. This suggests it could be harder to find high performing configurations for benchmarks that have greater potential reductions.

Four out of the 1000 configurations revealed a possible bug in GCC where compiling five of the benchmarks causes GCC to fill up the RAM entirely during compilation. More investigation

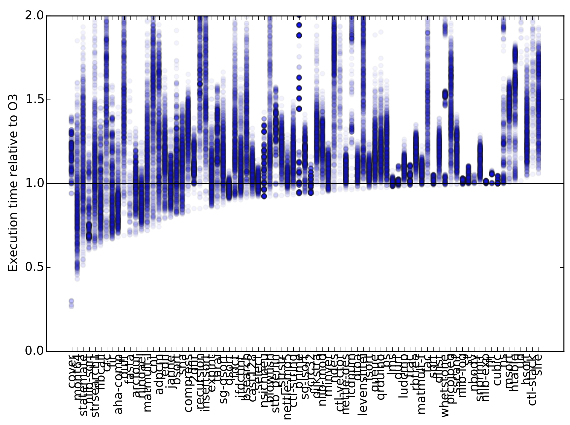
Table 3.1: Average improvement found by RIC for execution time, energy consumption and code size

Platform	Benchmarks	Time	Energy	Code size
CM3	BEEBS	13%	14%	1%
CA8	BEEBS	13%	14%	9%
CA8	cBench	15%	14%	12%

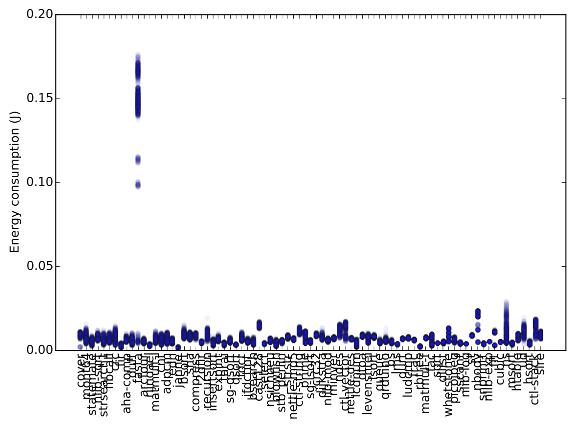
is required as to the cause of this bug. The CE experiments (Sec. 3.5) did not produce such configurations.



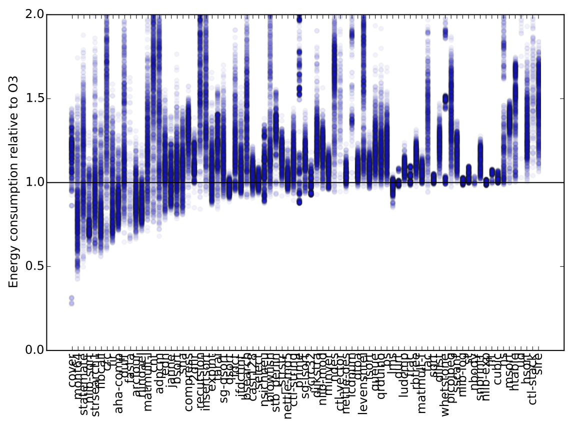
((a)) Execution time (s)



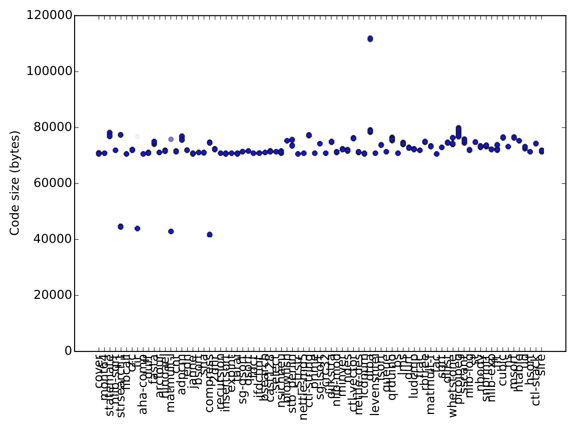
((b)) Execution time relative to -O3



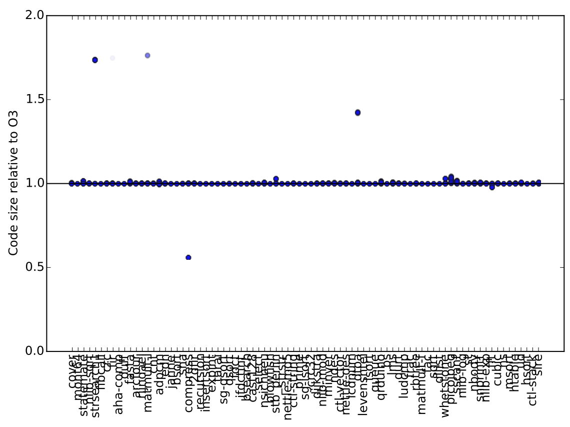
((c)) Energy consumption (J)



((d)) Energy consumption relative to -O3

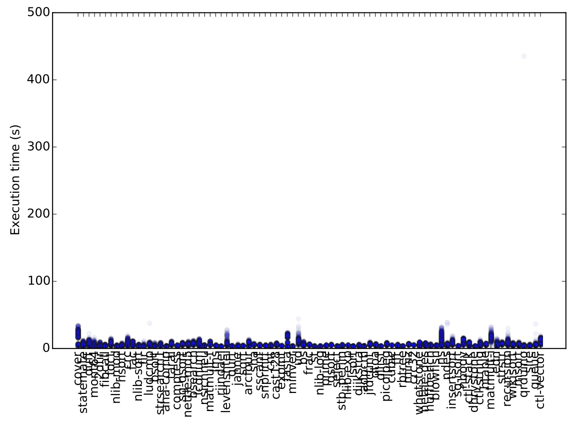


((e)) Code size (bytes)

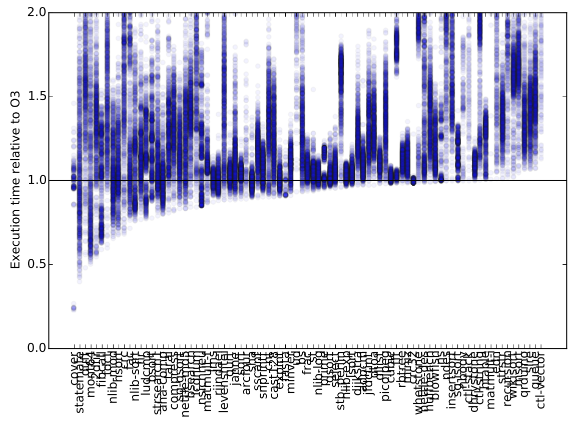


((f)) Code size relative to -O3

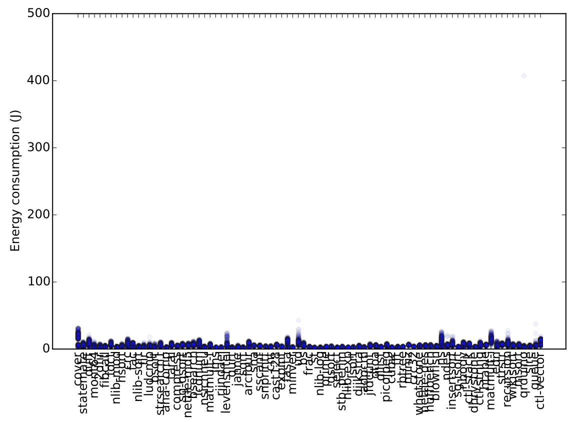
Figure 3.10: Execution time, energy consumption and code size of 1000 random configurations applied to each benchmark (CM3)



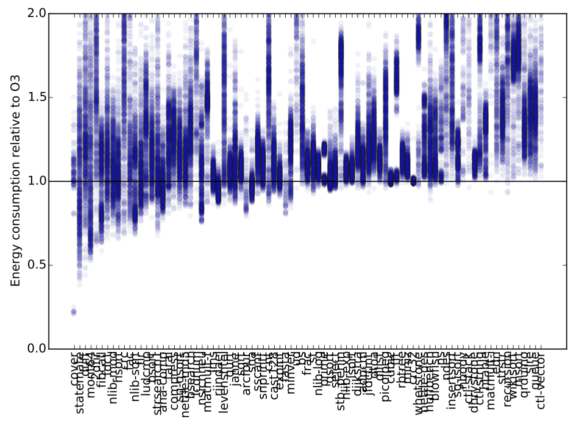
((a)) Execution time (s)



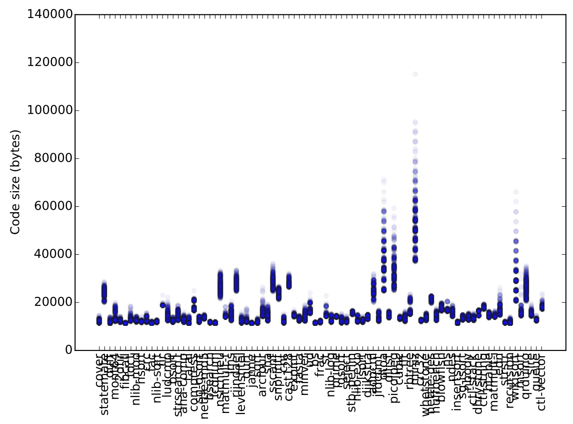
((b)) Execution time relative to -O3



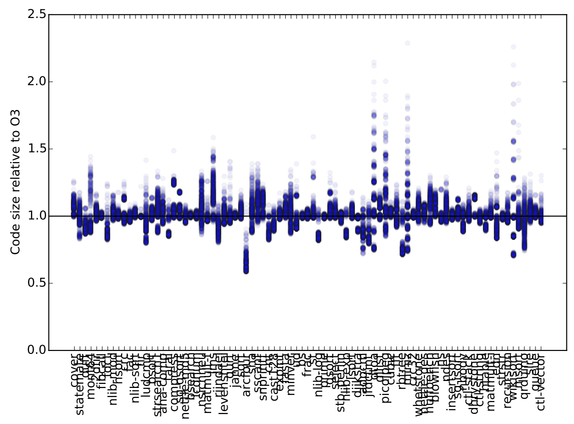
((c)) Energy consumption (J)



((d)) Energy consumption relative to -O3

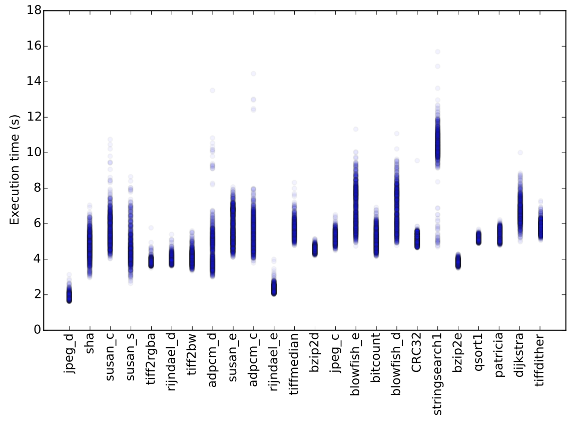


((e)) Code size (bytes)

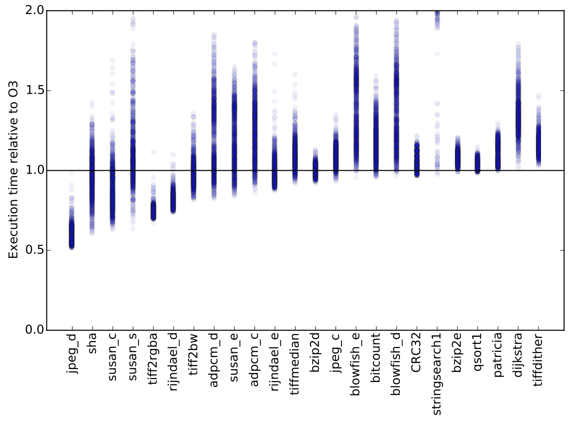


((f)) Code size relative to -O3

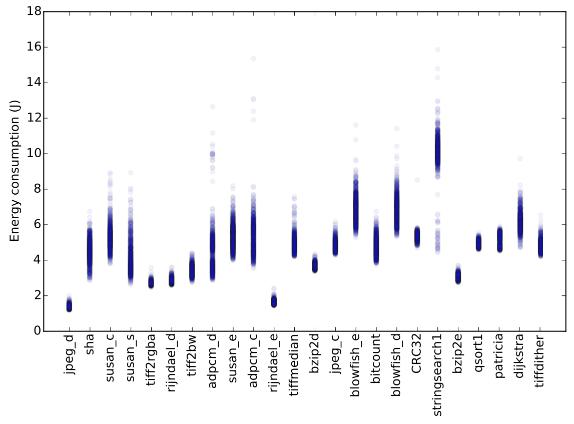
Figure 3.11: Execution time, energy consumption and code size of 1000 configurations applied to each benchmark (CA8 BEEBS)



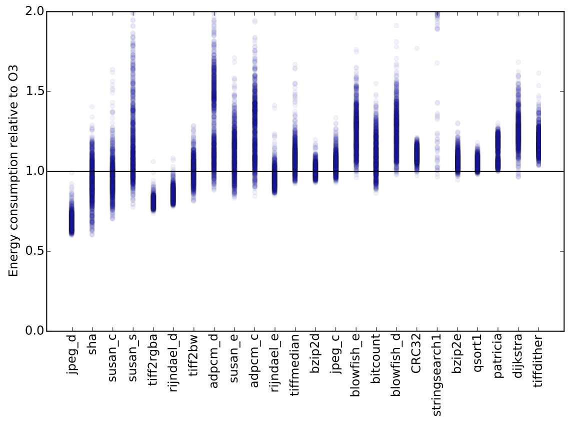
((a)) Execution time (s)



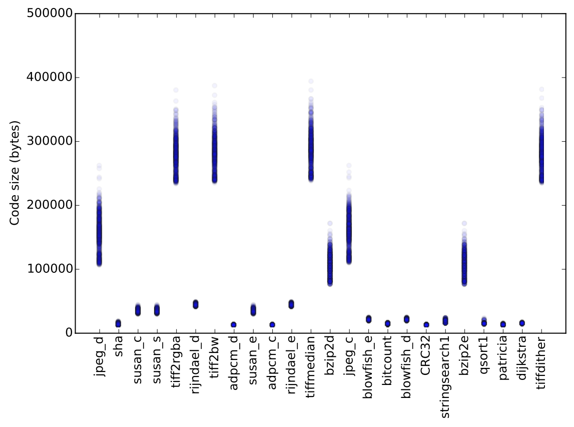
((b)) Execution time relative to -O3



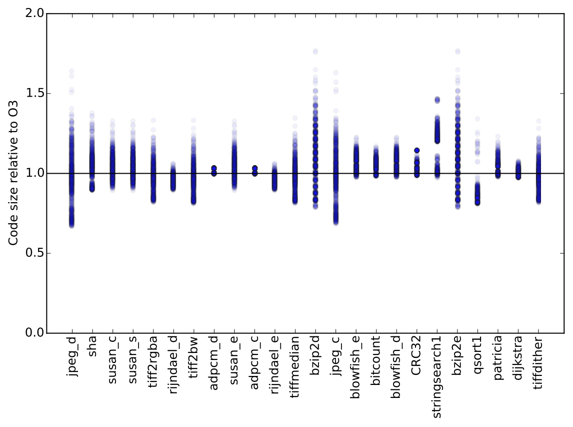
((c)) Energy consumption (J)



((d)) Energy consumption relative to -O3



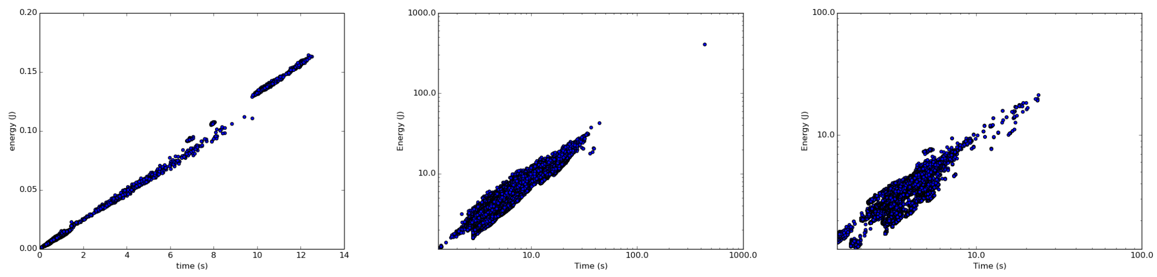
((e)) Code size (bytes)



((f)) Code size relative to -O3

Figure 3.12: Execution time, energy consumption and code size of 1000 configurations applied to each benchmark (CA8 cBench)

3.4.2 Energy-time correlation



((a)) BEEBS (CM3) ((b)) BEEBS (CA8) ((c)) cBench (CA8)

Figure 3.13: Correlation between time and energy for 1000 random configurations across all benchmarks

Energy and time were strongly correlated across all of the benchmarks (Fig. 3.13). A strong correlation was also observed in [57] which compared the relationship between time and energy on ten BEEBS programs run on five architectures including the CM3 and CA8. The study also discovered a more complex relationship on the CA8, which features a superscalar processor that allows it to execute multiple instructions at once, thus doing the same amount of work but in a shorter amount of time.

Given that energy and time were closely related on the target platforms and benchmarks, the rest of the methods explored will concentrate on execution time.

3.5 Finding further improvements with Combined Elimination[2]

This section uses the alternative approach of CE to search for further improvements that were not captured by RIC. Random Iterative Compilation is a naive approach that does not take the performance of previous configurations into account. Other methods such as CE [59] and GA [3] were proposed to build on knowledge from previous iterations, but researchers found that RIC outperformed them [3, 18] (Sec. 2.1.3.1). The results in Sec. 3.5.1 show that the relationship between RIC and CE is more complex, and on this particular experimental setup, CE outperforms RIC overall. There are some notable exceptions where RIC finds significantly better configurations for a few benchmarks, which are discussed further.

The CE data set was generated using the original CE algorithm [59] as described in Sec. 2.1.3.1. In preliminary experiments, the possibility of starting CE with different baseline configurations was also explored. The results presented in Sec. 3.5.2 suggest there is no clear benefit to starting CE from a particularly good configuration.

[2]Work in this section also appears in [15]

3.5.1 Comparison of RIC and CE

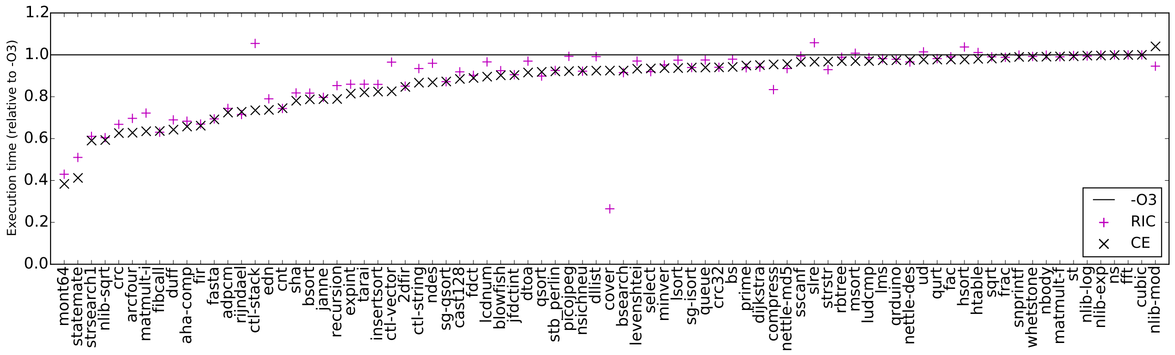


Figure 3.14: Best execution time achieved by RIC (1000 configurations) and CE (CM3)

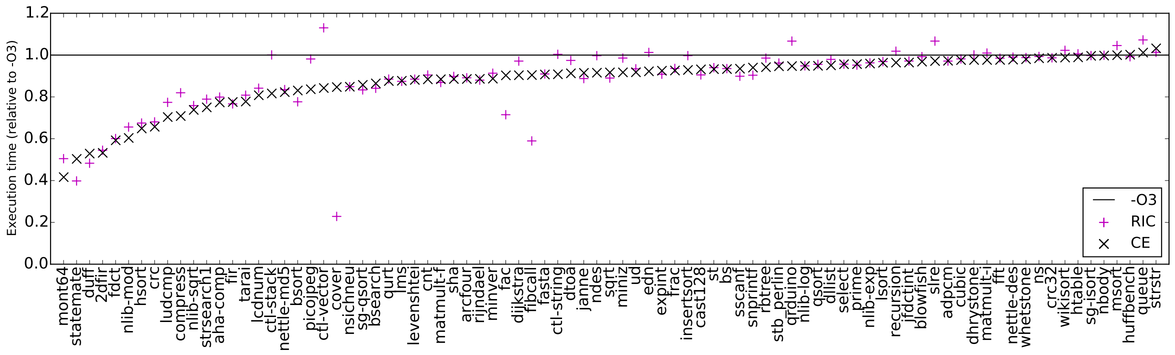


Figure 3.15: Best execution time achieved by RIC (1000 configurations) and CE (CA8 BEEBS)

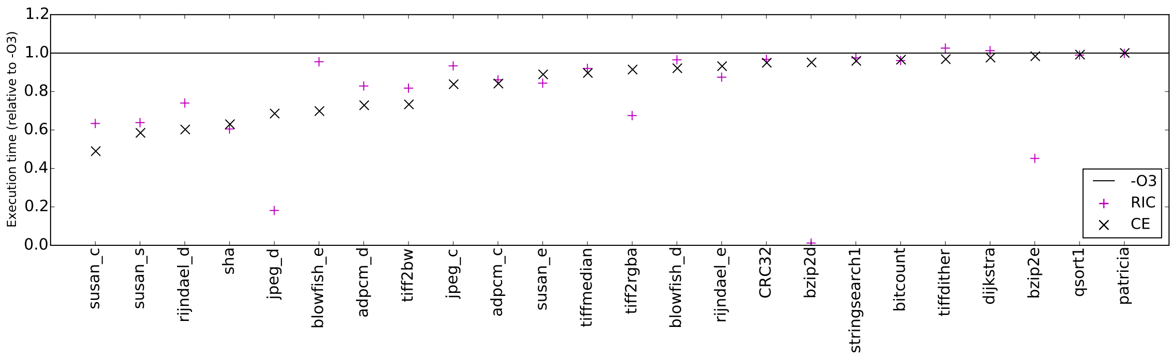


Figure 3.16: Best execution time achieved by RIC (1000 configurations) and CE (CA8 cBench)

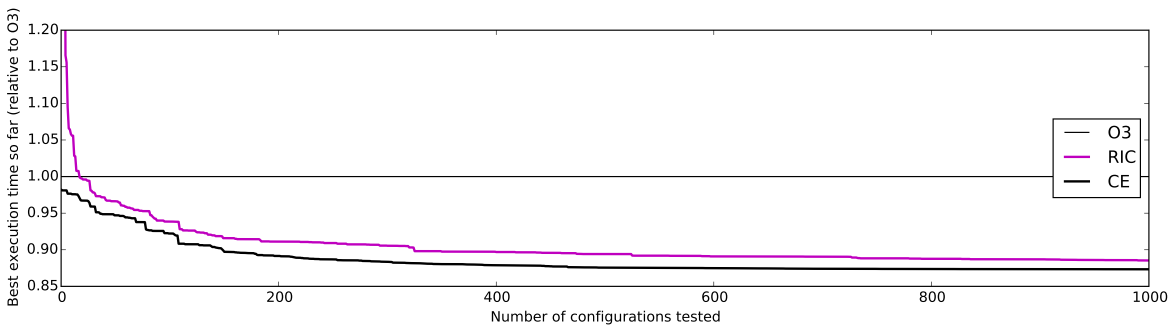


Figure 3.17: Average of best execution time for each benchmark after each configuration tested by RIC (1000 configurations) and CE (CM3)

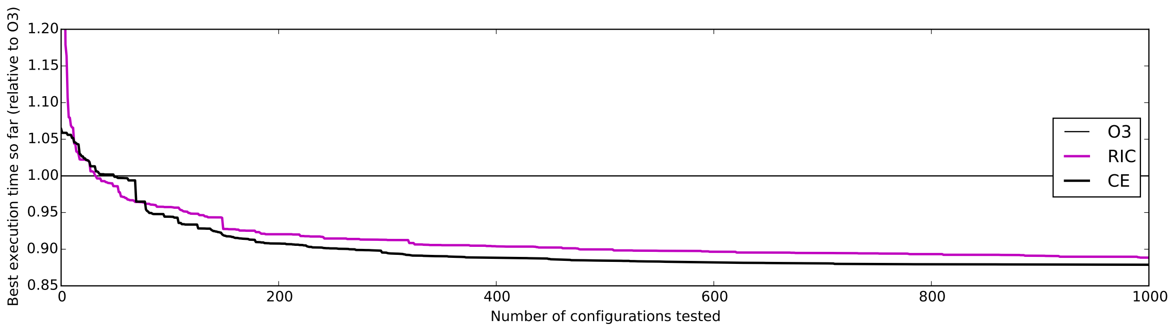


Figure 3.18: Average of best execution time for each benchmark after each configuration tested by RIC (1000 configurations) and CE (CA8 BEEBS)

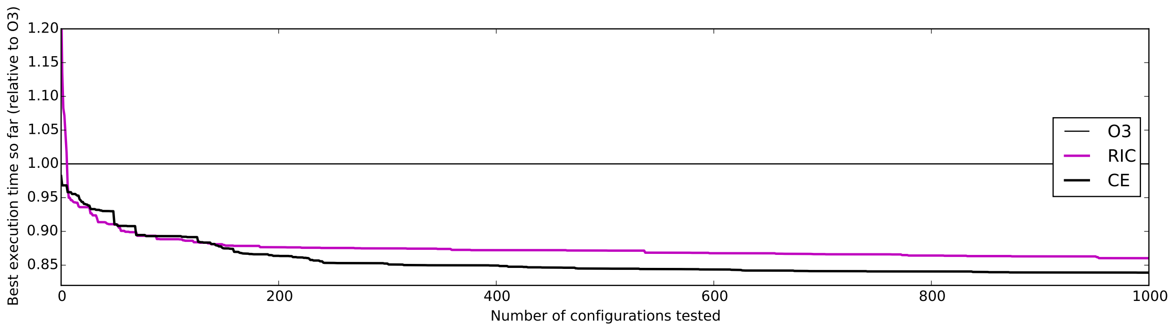


Figure 3.19: Average of best execution time for each benchmark after each configuration tested by RIC (1000 configurations) and CE (CA8 cBench)

This section compares the performance of RIC and CE in terms of best configuration found per benchmark and time taken to find good configurations. Overall, CE outperformed RIC but further analysis gives some insight as to why RIC occasionally finds better configurations.

The best execution times achieved by RIC and CE are shown for the CM3 (Fig. 3.14), CA8

BEEBS (Fig. 3.15) and CA8 cBench (Fig. 3.16). On the majority of benchmarks, CE outperformed RIC, including a few cases were RIC was unable to improve on `-O3` despite CE finding better configurations. For example, for RIC performed over 10% worse than `-O3` on `ctl-vector` on the CA8 whereas CE made a 15% improvement.

Conversely, RIC performed significantly better than CE in eight cases: `cover` and `compress` on the CM3; `cover`, `fac` and `fibcall` on CA8 BEEBS; and `bzip2e`, `jpeg_d` and `tiff2rgba` on CA8 cBench. Analysis of the `cover` RIC results for both the CM3 and CA8 showed that two flags, `-fivopts` and `-ftree-ch`, were always disabled in the best configurations. Further experiments showed that exclusively disabling one of these flags degraded performance and it was in fact the combination of both flags being disabled that led to improved performance. This is a dependency between the two flags which the CE algorithm is unable to capture due to the way it considers a single flag at a time.

While CE does not completely disregard dependencies (each decision to enable or disable a flag is dependent on the current baseline configuration) it only considers the effect of a single flag at a time, rather than toggling multiple flags at once. Allowing all single and pairs of flags to be toggled increases the search space exponentially but as a compromise, the CE algorithm could be modified to consider groups of flags with known dependencies (although finding these dependencies is non-trivial [19, 57]). Further work is required to determine whether the remaining six benchmarks for which RIC significantly outperformed CE also exhibit dependencies between flags that CE was unable to capture.

The real value of CE becomes apparent when analysing the amount of time each method takes to find good configurations. This is shown by plotting the average of the current best performance achieved on each benchmark after each configuration is tested (Figs. 3.17 to 3.19).

On the CM3, CE remains ahead of RIC for the full set of 1000 configurations tested. On the CA8, the two methods perform comparatively until CE overtakes RIC after 68 configurations on BEEBS and 141 configurations on cBench, and stays in the lead for the remaining iterations. Note that CE takes 134 configurations to test the initial baseline and each of the 133 flags. At configuration 68 on CA8 BEEBS, a single flag, `-fsched-interblock`, is disabled, which has a strong impact on performance. Conversely, on CA8 cBench it is only after multiple flags are disabled that CE remains ahead of RIC.

The RIC experiments were terminated after 1000 configurations due to time constraints, but the trajectory suggests it would take much longer for RIC to match the performance achieved by CE. Overall, RIC iterative compilation took 35.5 days to run (7.5 days CM3, 22 days CA8 BEEBS and 6 days CA8 cBench) and CE took 9.5 days (2.5 days CM3, 5 days CA8 BEEBS and 2 days CA8 cBench).

Cavazos et al. also compared RIC and CE but in contrast to the work in this thesis, they found that RIC outperformed CE. A direct comparison between the two studies is not possible as they are based on different platforms, benchmarks, optimisations and compilers. However, the following brief insight explains how the impact of flag dependencies presents a plausible reason why CE performed better in the present work.

Combined Elimination takes $N+1$ configurations to test the performance of the initial baseline and disabling each of the N flags individually. The results in the present work show significant gains even in this initial stage of removing single flags. Conversely, in [18] the majority of gains only occurred once the algorithm had begun disabling multiple flags. Therefore, flag dependencies may have a greater impact on their setup.

The following are two ways in which the experimental setup might influence the amount of flag dependencies. Firstly, the flags of the PathScale EKOPath compiler used by [18] may have more interdependencies than in GCC and secondly, their platform and/or benchmarks may be more sensitive to these flag dependencies.

3.5.2 Impact of different baselines

The standard CE algorithm [59] described in Sec. 2.1.3.1 starts with a baseline configuration of all flags enabled. In preliminary experiments on 60 BEEBS programs, two alternative baselines were tested: CE-O3 (which starts with `-O3`) and CE-random (which starts with the best configuration found by RIC). To work with these new baselines, the algorithm was modified to invert rather than disable flags. Consequently, any flag that is enabled in the initial configuration can be disabled by the modified algorithm and any flag that starts disabled can be enabled.

Note that the opposite of CE which could be called *combined addition*, in which the baseline has all flags disabled, was also briefly explored. Such an approach is extremely inefficient because many of the early iterations produce largely unoptimised code that produce slow execution times (for example, see the performance of `-O0` in Sec. 3.3). Intuitively, it is better to enable all flags and produce code with some level of optimisation rather than disable all flags and produce completely unoptimised code.

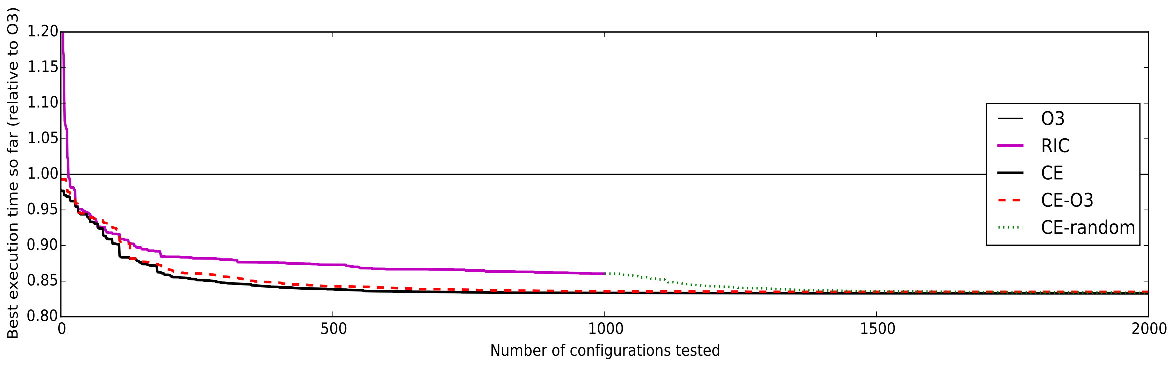


Figure 3.20: Best execution time achieved after each configuration tested for 1000 random configurations, CE-all, CE-O3 and CE-random (relative to O3)

Table 3.2: Average improvement made by RIC and CE combined for execution time

Platform	Benchmarks	RIC	CE	Best of RIC/CE
CM3	BEEBS	13%	14%	16%
CA8	BEEBS	13%	13%	16%
CA8	cBench	15%	18%	20%

The best overall execution time achieved after each iteration is shown in (Fig. 3.20) for CE-all, CE-O3 and CE-random. Surprisingly, CE-random shows no clear benefit in starting CE with a particularly good configuration. This could indicate that such a good configuration is close to a local optimum. CE-random also took slightly longer to terminate than CE.

Each of the three baselines tested produced comparable performance increases. Starting with all flags enabled is preferable because it slightly outperformed the other baselines and took the fewest iterations. No clear advantage was identified by starting the algorithm with a particularly good configuration found by RIC.

3.6 Summary of potential gains

The potential gains are quantified by taking the best known configuration found by either RIC or CE for each benchmark (Table. 3.2). This gives an overall improvement of 16% on both CM3 and CA8 BEEBS, and 20% on CA8 cBench compared to `-O3`. The best known configuration for each program provides a target with which to compare the compiler tuning methods tested in this thesis.

CHAPTER 4

RE-EVALUATION OF PREDICTIVE COMPILER TUNING APPROACHES

In this chapter, two state-of-the-art predictive compiler tuning approaches, Milepost [27] and COBAYN [10], are re-evaluated on the experimental setup of this thesis. Milepost is chosen because the ILP method developed later in Ch. 6 builds on its relational encoding of GCC's IR and COBAYN is chosen as one of the most recent and important works on the optimisation selection problem with a working implementation publicly available.

First there is a introduction to evaluating machine learning models (Sec. 4.1), followed by the re-evaluation of Milepost (Sec. 4.2) and COBAYN (Sec. 4.3).

4.1 Evaluating machine learning models

Machine learning models are evaluated by testing their performance on previously unseen examples that were not part of the training set. This is achieved by either by using a test set that is independent of the training set or applying cross validation.

Cross validation iterates over a number of 'folds' in which a proportion of the training data is excluded from training and presented as a test set. The two most common approaches are leave-one-out cross validation and 10-fold cross validation. In leave-one-out cross validation, a single program is excluded at a time from training and presented as a test. In 10-fold cross validation, a tenth of the programs are excluded from training in each fold.

Cross validation is useful when there are relatively few data points available, such as in compiler tuning where the number of benchmarks is typically in the tens. This is contrast to big data problems with tens of thousands of data points where it is feasible to partition into a single training and test set.

Cross validation gives a measure of how well the algorithm generalises the problem and can detect overfitting. It is not expected to perform as well as using the whole training set as the model is trained on less data. This is particularly true on compiler tuning problems in which the benchmark space is sparse.

4.2 Milepost[1]

The Milepost [27] project put a lot of effort into empirically selecting suitable features for machine learning and the approach targeted a similar number of flags to the present work. They tested 1NN and decision tree approaches on 22 cBench programs and concluded that 1NN gave the best results (Sec. 2.1.4). This section re-evaluates Milepost 1NN on nearly four times as many benchmarks by using BEEBS to target the CM3 and CA8.

The Milepost project is of particular interest due to the Datalog encoded relational IR that its Milepost GCC produces. In Milepost, this IR is only used to calculate simple statistical aggregates that describe features of the code. Later in Ch. 6, ILP will be used to learn directly from the IR which retains potentially important structural information that is otherwise lost by flattening into a predefined feature vector. The re-evaluation in this section provides comparative data for evaluating the methods developed in this thesis.

Training data was produced for 1NN by extracting the feature vector for the most time consuming function of each program (using Milepost GCC) and combining this with the RIC data from the investigatory study in Ch. 3. A new implementation of the 1NN algorithm was created as Milepost had not been trained for the CM3 and it was not possible to supply new data to their system.

4.2.1 Results

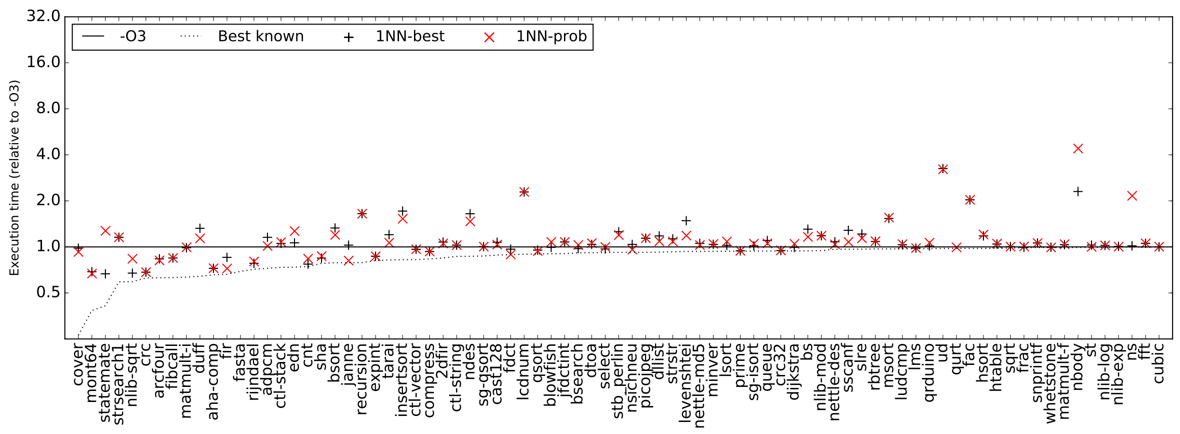


Figure 4.1: Performance of 1NN-best and 1NN-prob (CM3)

[1]This section builds on work in [13].

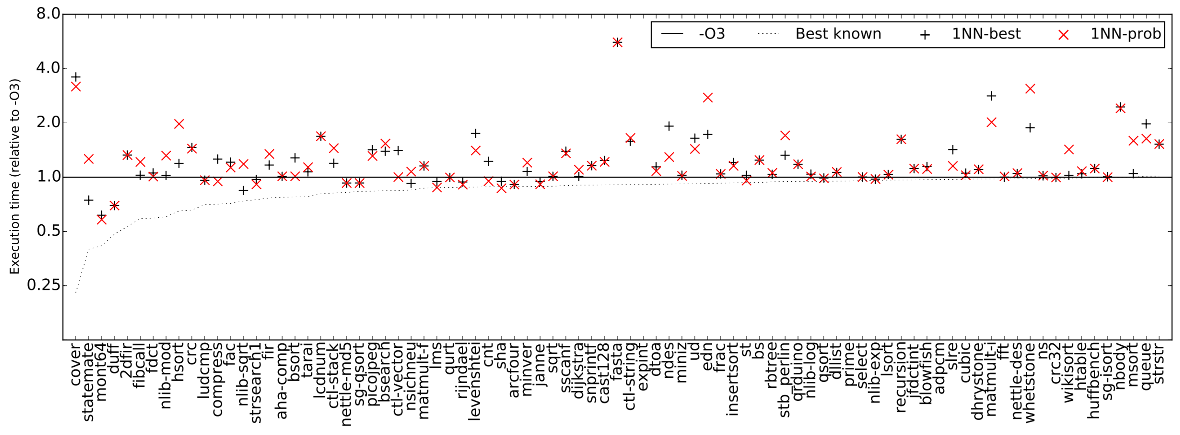


Figure 4.2: Performance of 1NN-best and 1NN-prob (CA8 BEEBS)

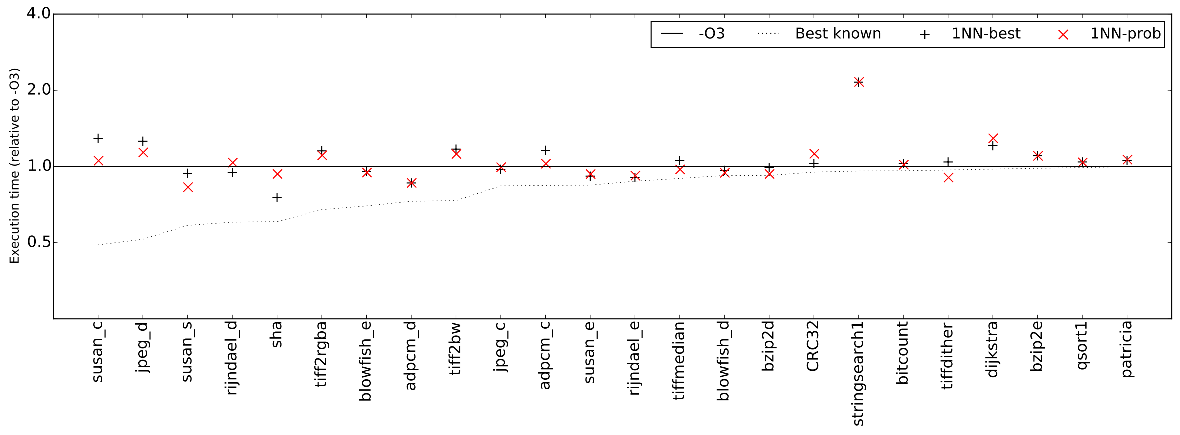


Figure 4.3: Performance of 1NN-best and 1NN-prob (CA8 cBench)

As in the original study, leave-one-out cross validation (Sec. 4.1) was used to test the 1NN-best and 1NN-prob approaches.

Both the 1NN approaches actually performed worse than `-O3` overall (Figs. 4.1 to 4.3). On BEEBS, 1NN-prob increased overall execution time by 13% (CM3) and 22% (CA8). Similarly, 1NN-best increased execution time by 12% (CM3) and 20% (CA8). For cBench on the CA8, 1NN-prob was 4% slower than `-O3` and 1NN-best was 6% slower. In summary, the two techniques performed comparably, with 1NN-best slightly better overall on BEEBS and 1NN-prob slightly better on cBench. Milepost [27] also found that 1NN-prob outperformed 1NN-best on their data set targeting cBench, but in contrast they achieved an average speed-up of 10% using 1NN-prob on an embedded ARC platform.

The fact that 1NN performed worse on BEEBS than cBench could be because the BEEBS training programs are more diverse than those in cBench. Figures 2.2 and 2.3 show the diversity of the two suites by comparing the Euclidean distance between the normalised feature vector of each program.

The set of cBench programs used by Milepost contains clusters of very similar programs (the lighter the dot, the closer the two programs). In fact, the `blowfish` encryption and decryption programs have identical source code but are included as two different benchmarks `blowfish_e` and `blowfish_d` (as is the case for `rijndael`). If the initial set of 22 benchmarks is reduced to allow only one variation of each program, then this gives a total of 13 programs. In contrast, the BEEBS programs were designed specifically to cover a wide range of features and consequently the programs are much more distant (Fig. 2.3). Some of the BEEBS programs were in fact derived from cBench including `blowfish` and `rijndael`, however, these only appear once in the set and overall BEEBS contains only three pairs of very similar programs (`ctl-stack`, `ctl-string`), (`matmult-int`, `matmult-float`) and (`trio-sscanf`, `trio-sprintf`).

One way to improve the 1NN results would be to add extra training programs to cover a wider range of optimisation scenarios, however, this may require a huge number of programs which would increase training times dramatically. Alternatively, one could search for patterns within the program structure that may help to predict effective configurations, as in the ILP methodology proposed in Ch. 6.

4.3 COBAYN

The recent COBAYN study [10] sought to capture dependencies between flags by using a Bayesian Network algorithm. While the study targeted the Pandaboard (with Cortex-A9 processor), the experiments focused on seven flags identified by earlier work as having the most significant impact on the Intel Xeon processor which is a different processor [19]. First the impact of limiting the flag set is investigated and then the performance of COBAYN in 1-shot and 8-shot mode is explored.

It was originally intended to re-evaluate COBAYN on the Pandaboard running ArchLinux, as used in the original study, however, the board proved to be unstable for large scale iterative compilation experiments and would often overheat despite cooling efforts. Enabling CPU frequency scaling allowed the experiments to run for longer, but it is desirable to keep the CPU frequency constant for consistent training data so that the impact of configurations are measured, not the impact of frequency scaling. There is no mention of hardware issues in the COBAYN study and the CPU frequency settings are not discussed.

This section tests the COBAYN approach on the CM3 and CA8 using a publicly available implementation available on GitHub: `https://github.com/amirjamez/COBAYN`.

4.3.1 Choice of flags

To demonstrate the impact of limiting the set of flags on which compiler tuning is focused, this section compares the seven flags explored by COBAYN with the 133 flags on which this thesis

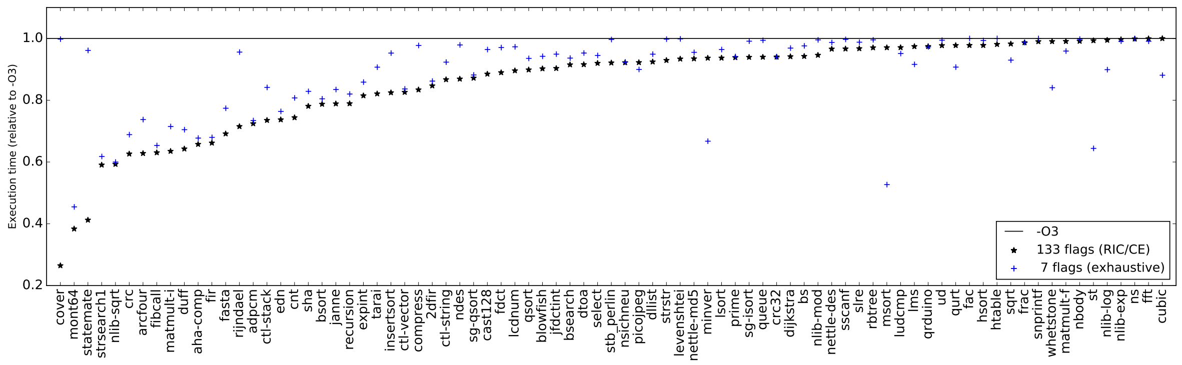


Figure 4.4: Performance of exhaustive search (7 flags) and RIC/CE search (133 flags) (CM3)

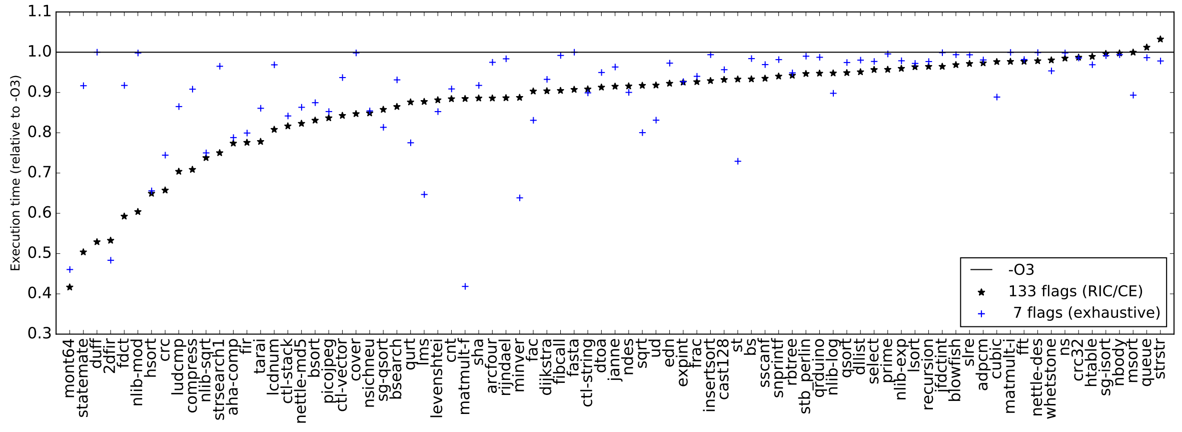


Figure 4.5: Performance of exhaustive search (7 flags) and RIC/CE search (133 flags) (CA8 BEEBS)

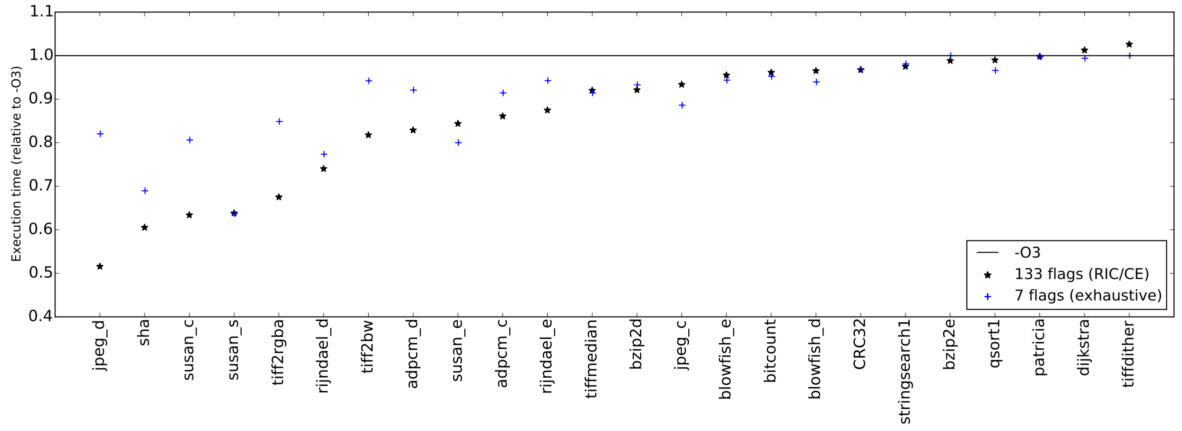


Figure 4.6: Performance of exhaustive search (7 flags) and RIC/CE search (133 flags) (CA8 cBench)

focuses. Exhaustive data for the 128 configurations yielded by the seven flags are compared to RIC and CE data for the 133 flags for the CM3 and CA8.

On the CM3, the seven flags achieved 12% speed-up while the 133 flags achieved 16% speed-up (Fig. 4.4). Similarly, on the Cortex-A8, the seven flags achieved 11% speed-up on both BEEBS and cBench, while the 133 flags achieved 16% on BEEBS and 20% on cBench (Figs. 4.5 and 4.6). Therefore, it is clearly advantageous to develop methods that can handle large sets of flags in order to access these additional speed-ups and avoid the extra effort of reducing the flag set before experimentation can begin on each platform.

There were some cases where the seven flags found much larger improvements, this is due to the inclusion of `-funsafe-math-optimizations` in the seven flags; this flag was excluded from the 133 flags explored in this thesis because it does not conform to IEEE floating point standards and may produce incorrect/inaccurate output. Should the user wish to benefit from this flag while using a method trained with the 133 flags, they can simply try adding it to whatever configuration has been generated. Additionally, the automaticity of these approaches make it convenient to target whichever candidate flags are desired.

Not only does choosing a small set of flags limit the improvements that can be made on some programs, this selection may need to be done for each new platform. Therefore, it is advantageous to let the compiler tuning approach decide which flags are most important by embedding this decision into the methodology.

4.3.2 Performance of COBAYN models

Both the 1-shot and 8-shot modes were tested on static features. 1-shot often produced configurations worse than `-O3` (Figs. 4.7 and 4.8). Conversely, 8-shot came close to the best possible configuration of the seven flags in most cases. Given there are only seven flags and some or all effects may be independent, it is not unlikely that a near-optimal configuration is found within eight attempts. The poor performance of 1-shot is not surprising as the original study shows that it takes 2-3 configurations to start outperforming `-O3`. The fact that 1-shot often performs significantly worse than `-O3` suggests a weakness in the model and demonstrates that predicting the configuration in 1-shot is a much harder problem. While 8-shot gets close to the best seven flag configuration, it requires multiple evaluations to achieve this, which becomes infeasible for larger programs with long compilation times.

It was not possible to test COBAYN on cBench due to a bug in the framework. Although it would be interesting to pursue fixing the bug, this falls outside the scope of this thesis. COBAYN also does not appear to scale well to a large set of flags. When trying to train a model for the 133 flags, the framework crashed. This limitation means that users must invest extra effort to select a reduced flag set before gathering training for COBAYN and this process needs repeating for each new platform. The approaches developed in this thesis allow large sets of flags to be used and do not assume prior knowledge of which flags are important. In fact, the sensitivity analysis within the ILP method (Ch. 7) automatically filters important flags for a given configuration.

4.4 Summary of results

This chapter re-evaluated Milepost and COBAYN and found that overall, the 1-shot approaches performed worse than `-O3` (Table. 4.1). The only exception was COBAYN 1-shot which achieved a 3% improvement on CM3 BEEBS. In contrast, the COBAYN 8-shot methods reached higher gains of 11% on CM3 BEEBS and 9% on CA8 BEEBS, but this is at the expense of running each program eight times.

Table 4.1: Average improvement made by Milepost and COBAYN

Platform	Benchmarks	1NN-prob	1NN-best	COBAYN 1-shot	COBAYN 8-shot
CM3	BEEBS	-13%	-12%	3%	11%
CA8	BEEBS	-22%	-20%	-3%	9%
CA8	cBench	-4%	-6%	-	-

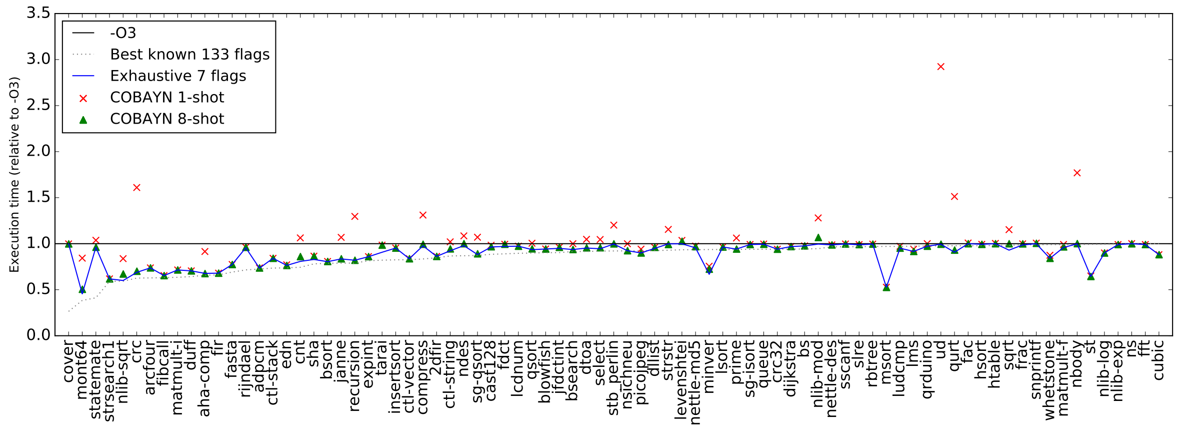


Figure 4.7: Performance of COBAYN in 1-shot and 8-shot mode (CM3)

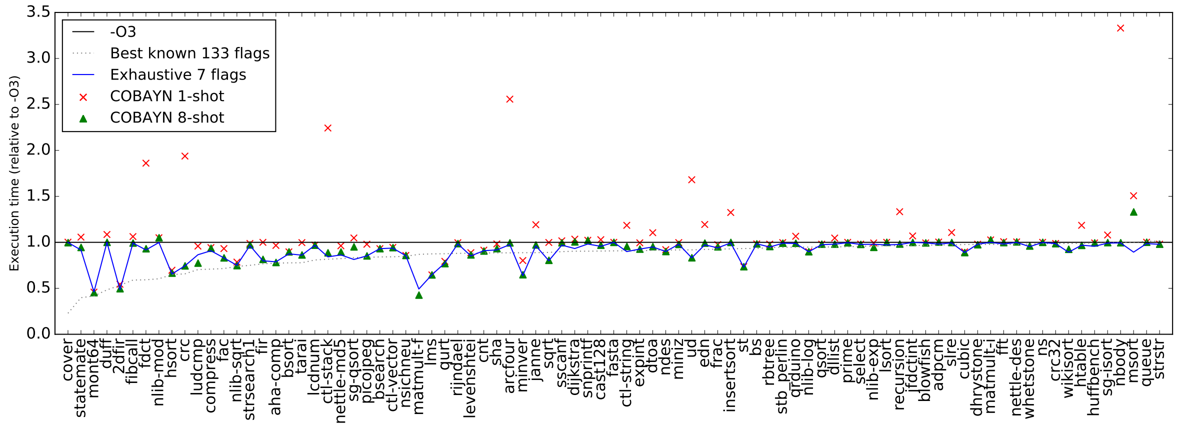


Figure 4.8: Performance of COBAYN in 1-shot and 8-shot mode (CA8 BEEBS)

CHAPTER 5

CONSTRUCTING NEW PROGRAM-AGNOSTIC OPTIMISATION LEVELS

This chapter explores the benefits of tuning the compiler to a whole platform by identifying and exploiting flags that consistently improve or degrade performance. In contrast to iterative compilation techniques that perform a new search for each program (Ch. 3), this approach seeks to automatically find a single configuration that performs well across a whole benchmark suite. Once constructed, this program-agnostic configuration can be applied to new programs with no further effort and none of the complications of a machine learning based approach.

The approach taken is similar to COLE [39] which changed the goal of iterative compilation from optimising the performance of a single target program to optimising the performance of a whole benchmark suite. In COLE, most of the analysis related to average performance and the impact on individual benchmarks was not clear (Sec. 2.1.3.2), for example, some programs may have performed considerably worse than `-O3`. While they used a genetic algorithm for their study, the present work adapts CE and introduces a threshold to prevent any one program from performing significantly worse than `-O3`. Using CE with this threshold allows evaluations of some configurations to be aborted early as soon as a program performs worse than the threshold.

This analysis uses over two times as many flags as [39], which did not consider flags not enabled at `-O3` and used an older version of GCC (4.1.2) with fewer flags available.

A key contribution of this chapter is the comparison of program-agnostic configurations with state-of-the-art Milepost and COBAYN predictive approaches. Previous machine learning approaches do not compare to program-agnostic configurations constructed for the target platform, but it is important to test the added benefit of such program-specific approaches to check whether a machine learning approach is actually necessary.

Section 5.1 investigates the hypothesis that constructing program-agnostic configurations is competitive with more complex state-of-the-art predictive approaches. In order to test this hypothesis, this study adapts the CE algorithm to construct a single configuration, called `-Ocm3` that outperforms `-O3` on the BEEBS benchmark suite running on the STM32VLDiscovery. Cross-validation experiments are used to compare the method to more the complex COBAYN and Milepost machine learning approaches. The approach is then evaluated on an additional platform, the Beaglebone, for which `-Oca8` is created respectively. Each program-agnostic configuration provides a significant speed-up over `-O3`. Two flags that were disabled in `-Ocm3` are analysed in detail to explain why they degrade performance on the STM32VLDiscovery and therefore why disabling them has a positive impact on performance.

Although the newly constructed optimisation levels outperform `-O3` on their target platforms, further gains are still possible by targeting individual programs directly. Section 5.2 explores a compromise between program-agnostic and program-specific optimisation by constructing a small set of configurations such that for each program there exists at least one configuration that provides good performance. This provides a small platform-specific set of configurations on which to perform iterative compilation on new programs. The approach builds on a method proposed in [63] for targeting the phase-ordering problem in LLVM.

Section 5.2 proposes an Integer Linear Programming that, in contrast to [63], is guaranteed to find the optimal subset of the configurations sampled by iterative compilation. Furthermore, the approach targets the goal of minimising overall execution time rather than maximising program coverage, so that obtaining larger improvements is prioritised over covering programs with only small available gains.

Running iterative compilation over the reduced set provides near-optimal results across the benchmarks, but it would still be preferable to predict a configuration for a given program without having to compile and run multiple times. Therefore, the next chapter (Ch. 6) will apply ILP to the problem of one-shot compiler configuration prediction.

5.1 Constructing program-agnostic optimisation levels[1]

Iterative compilation techniques are typically applied to search for a single configuration to improve the performance of a particular program. By changing the goal from enhancing a target program's performance to maximising the overall performance of a benchmark suite, iterative compilation methods can be adapted to find a single configuration tailored to the target platform. This section aims to find a single configuration that improves overall performance without having a significant negative impact on any one program.

5.1.1 Adapting combined elimination

The aim is to build the creation of optimisation levels into the CE algorithm such that the final configuration is the new optimisation level itself. Intuitively, the method starts with `-O3` as its

[1]Work in this section also appears in [15].

baseline configuration and continually enables or disables the next flag that gives the biggest improvement across *all* benchmarks while not causing any one benchmark to perform worse than a threshold *t%* of `-O3`. The result is a configuration that performs at least within *t%* of `-O3` or better for each benchmark. Threshold *t%* controls the trade-off between performance gains and losses which will be explored further in Sec. 5.1.2.

The approach is tested on CE by making the following changes to the original algorithm [59] that was described in Sec. 2.1.3:

- Instead of targeting the performance of a single program, the target is the overall performance of the benchmark suite running on a given platform.
- Rather than starting with a baseline configuration of all flags enabled and then selectively disabling flags, the baseline configuration is `-O3` and the algorithm can either disable flags that are in `-O3` or enable flags that are not in `-O3`. Any configuration upon which the user wishes to improve can be chosen as the baseline.
- As soon as a configuration causes a program to perform *t%* worse than `-O3`, the remaining tests for that configuration are skipped as it will not satisfy the requirement that performance must be at least within *t%* of `-O3` or better. This increases the efficiency of the search by avoiding unnecessary evaluations.
- As in Sec. 3.4, to further aid efficiency of the search, the md5 hash of each compiled binary is stored along with its performance measurement. If a subsequent binary has a matching hash then the cached performance is used rather than rerunning the program.

The next section (Sec. 5.1.2) analyses the results of applying this method to find a single configuration that outperforms `-O3` on the CM3 and it also highlights how the modifications to the algorithm improve the efficiency of the search.

5.1.2 Threshold trade-off in constructing `-Ocm3`

The proposed method for constructing `-Ocm3` (Sec. 5.1.1) was tested with several thresholds from $t = 0\%$ to $t = 6\%$ using fixed increments of 1%. The configuration generated by $t = 5\%$ gave the best average performance which was 10% better than `-O3`. Under this configuration, many of the benchmarks perform close to their best known configuration and only a few perform worse than `-O3` (Fig. 5.1). The worst performing program ran 4% slower than `-O3`. Several benchmarks performed as well as the best known configuration.

A more conservative threshold $t = 2\%$ performs 4% better than `-O3` overall and still manages several improvements with fewer programs performing worse than `-O3`. There is, then, a trade-off between optimising some benchmarks in exchange for losses on others.

With $t = 5\%$ the method identifies 20 flags that should be disabled from `-O3` and three additional flags that should be enabled. Two of these flags will be discussed in detail later (Sec. 5.1.3).

```
-O3 -fno-tree-loop-if-convert -fno-common -fipa-pta -fno-sched-interblock
    -fno-tree-copyrename -fno-peephole2 -fno-expensive-optimizations
    -fno-ipa-sra -fgcse-las -fno-schedule-insns
    -fno-tree-loop-distribute-patterns -fno-caller-saves -fno-optimize-strlen
    -fno-inline-functions-called-once -fno-tree-slsr -fno-tree-scev-cprop
    -funroll-all-loops -fno-sched-dep-count-heuristic -fno-tree-ccp
    -fno-predictive-commoning -fno-ipa-pure-const -fno-merge-constants
    -fno-tree-pta
```

Listing 5.1: The -Ocm3 configuration

These results demonstrate that the gains of -Ocm3 created by $t = 5\%$ outweigh the losses and support its use on the CM3 instead of -O3. The complete configuration is given in line 2 (with the flags shown in the order that they were disabled or enabled by the method). The method took 19 hours to run with $t = 5$, which is over twice as fast as CE and seven times faster than RIC used in (Ch. 3).

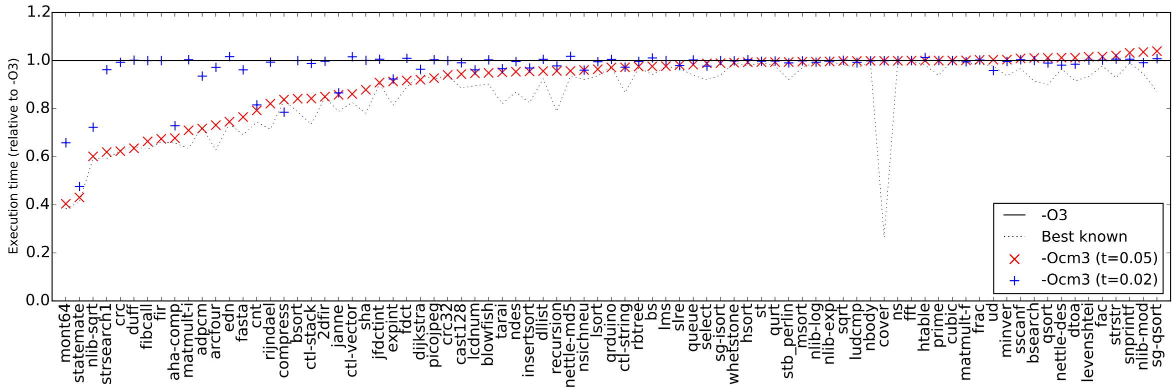


Figure 5.1: Trade-off between thresholds for constructing -Ocm3

5.1.3 Analysis of two excluded flags

To explain why some of the flags included in -O3 appear to actually reduce performance on the CM3 architecture, this section analyses two such flags (-fcommon and -ftree-loop-if-convert) which the adapted CE method indicates should always be disabled. Although both of these flags are in fact enabled at all optimisation levels from -O0 upwards, disabling them actually reduces the overall average execution time of BEEBS by 3% and significantly improves the performance of 13 benchmarks while leaving all the others virtually unaffected.

5.1.3.1 -fcommon

The -fcommon flag controls the placement of uninitialised global variables within object code. As stated in the GCC manual, the flag is provided for compatibility but may lead to a speed or code size penalty on some platforms [69]. Disabling the flag on the Cortex-M3 improves overall execution time by 1% and has a significant impact on `statemate` and `compress` which are improved by 43% and 16% respectively.

The use of `-fcommon` prevents the compiler from knowing that two global variables will share contiguous memory. Such information could be used on the CM3 to exploit Load Multiple Increment After (LDMIA) or Store Multiple Increment After (STMIA) instructions which allow two variables to be loaded or stored in a single instruction.

In more detail, `-fcommon` allows duplicate definitions of uninitialised global variables across different source files. Each definition of a global variable (including duplicates) appears in the common section of the object code and the linker then chooses which of these definitions to use. Unfortunately, this prevents the compiler from knowing the relative location of global variables, and it cannot optimise based on the assumption that they will occupy contiguous memory.

In contrast, when `-fcommon` is disabled, each global variable can only be defined once and any other declarations must be qualified with the `extern` keyword. Each global variable is defined once in the data section of the object code and its location relative to other variables is preserved.

The effect of `-fcommon` can be illustrated by discussing the following example code:

```
int x,y,z;
void g() {
    z = x - y;
    x = z * y;
    y = z * x;
}
```

Compiling with `-fcommon` produces twice as many instructions than when it is disabled. Only the disabled version manages to reduce the number of memory instructions by taking advantage of LDMIA and STMIA. In addition, the enabled version uses more than 4 registers which causes a further inefficiency on the CM3 as additional stack operations are required to ensure the extra registers are restored to their original values before the function returns [6].

5.1.3.2 -ftree-loop-if-convert

This flag converts conditional jumps in innermost loops to branchless equivalents in order to improve later vectorization optimisations switched on at `-O3` [69]. There is no indication in the GCC manual, however, that this flag might degrade performance on a processor such as the CM3 that does not support vectorization. This section investigates the impact of this flag further and explains why it does indeed increase runtime on the CM3.

Disabling `-ftree-loop-if-convert` improves overall execution time by 2% and significantly improves `aha-mont` by 50%, `newlib-sqrt` and `aha-compress` by 25%, while not degrading the performance of any remaining benchmark.

Intuitively, this flag removes an if-statement and replaces it with code that always executes both the if-true and if-false body and then uses predicated instructions to determine which result(s) should be used. Consider the following if-statement found in `newlib-sqrt`:

```
if(t<=ix) {
    s    = t+r;
    ix  -= t;
    q   += r;
}
```

When `-ftree-loop-if-convert` is enabled, the code is converted to the following:

```
s2    = t+r;
ix2  -= t;
q2   += r;
ix    = (t<=ix) ? ix2 : ix;
s     = (t<=ix) ?  s2 : s;
q     = (t<=ix) ?  q2 : q;
```

The code produced by `-ftree-loop-if-convert` always executes the if-true body, but then must execute three more statements to decide which value to use for each variable. In contrast, the original version of the code only executes the if-true body when the condition is true. It is likely that the disabled version would perform increasingly well as the proportion of times the if-condition evaluates to false increases. Under the default input data for `newlib-sqrt` the true:false ratio is 1:2.

5.1.3.3 Lessons Learned

This section analysed two flags in detail to determine why disabling them is beneficial to performance on the CM3. While such manual analysis provides interesting insights it is a time-consuming task that is infeasible to repeat for the very many flags and platforms available. The adapted CE based approach enables the automatic discovery of such important flags for new architectures without the need for in-depth manual analysis.

5.1.4 Cross-validation of `-Ocm3`

In Sec. 5.1.2 the whole of BEEBS was used to construct a single configuration, `-Ocm3`, that performed well across the benchmark suite. To verify that the method does not simply over-fit the benchmark suite, this section uses the standard 10-fold cross-validation technique to test the method on unseen programs.

5.1.4.1 Method

In 10-fold cross-validation, the programs are partitioned into ten training and test folds. In each fold, 90% of the programs form the training set and the remaining 10% form the test set. Each program appears in the test set of exactly one fold and in the training set of the other nine folds. The folds for this analysis were generated using uniform random sampling.

In each fold x, the configuration `-Ocm3-fold-x` is constructed based on the training set and its performance is evaluated on the test set.

5.1.4.2 Cross-validation Results

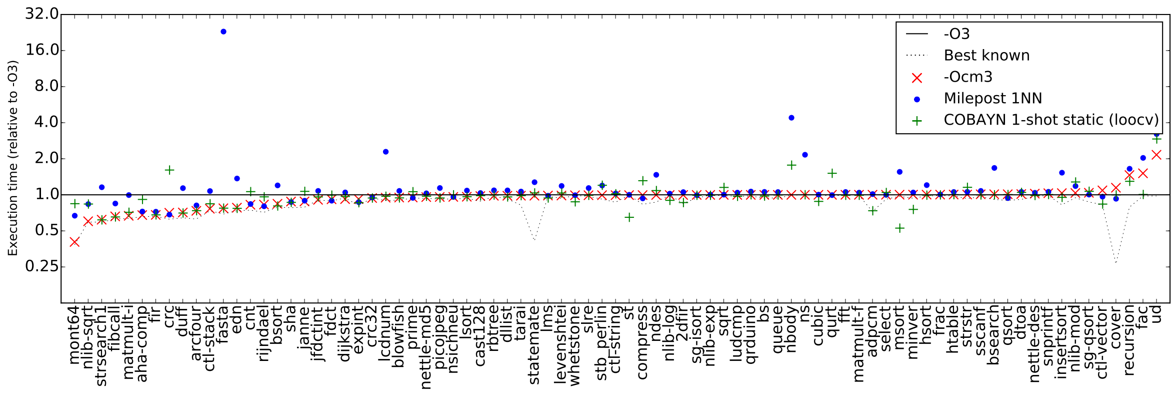


Figure 5.2: 10-fold cross-validation of `-Ocm3`, Milepost 1NN and COBAYN 1-shot (logarithmic scale)

In cross-validation `-Ocm3` performed 6% better than `-O3` overall and fifteen programs reached speed-ups of over 20% (Fig. 5.2). In many cases, performance was close to the maximum known potential gain. Figure 5.2 also compares performance to Milepost discussed later in Sec. 5.1.4.3.

Three programs (`recursion`, `fac` and `ud`) ran over 20% slower than `-O3`. This is actually an artefact of using cross-validation as each of the three programs have unique optimisation requirements that are not captured by any other program in the training set. Therefore excluding these programs from the training set prevents their requirements being included in the configuration. The first two programs also feature recursive calls, which would not normally be used on embedded systems due to memory constraints.

Both `recursion` and `ud` appeared in the same cross-validation test fold. The configuration generated for this fold disables two flags (`-ftree-reassoc` and `-fipa-cp-clone`) that significantly optimise `recursion` and `ud` respectively. This is the only fold that disables these flags therefore `recursion` and `ud` are unique in their dependence on these flags and none of the remaining training programs could prevent them from being disabled.

A similar story holds for `fac` in another fold. This program gains significant benefit from enabling `-foptimize-sibling-calls` and disabling `fmodulo-sched` but the configuration constructed in this fold does the opposite by disabling the former and enabling the latter. As in the previous scenario, this is the only fold that features these particular settings.

BEEBS was deliberately designed to include a diverse range of benchmarks with little redundancy between them, therefore optimal performance cannot be expected when training on a subset of the benchmarks. Nevertheless, these results do show that the method performs well in general and not due purely to chance.

In conclusion, the majority of programs performed as well as or better than `-O3`. In practice, should a program perform worse than `-O3`, the user can simply choose `-O3` instead. This is a much less time-intensive task than choosing from hundreds of configurations.

5.1.4.3 Comparison with machine learning approaches

The `-Ocm3` cross-validation results were compared with the state-of-the-art 1NN probabilistic machine learning approach from Milepost [27] and COBAYN [10] using the same cross-validation folds. Milepost 1NN performed 14% slower than `-O3` with the majority of programs performing worse than both `-O3` and `-Ocm3` (Fig. 5.2). The COBAYN 1-shot approach improved performance by 3% in 1-shot mode and by 11% in 8-shot mode (at the expense of seven extra evaluations compared to the 1-shot methods). The `-Ocm3` level performed best out of `-O3` and the 1-shot predictive approaches but without the overheads and complexities of a machine learning based approach.

5.1.5 Testing on other platforms

In order to demonstrate that the adapted CE method can also optimise GCC for other embedded platforms, this section constructs and tests two new optimisation levels for the CA8 processor. `-Oca8-beebs` (Listing 5.2) is constructed using BEEBS and `-Oca8-cbench` using cBench (Listing 5.3, The threshold $t = 5\%$ was used to produce these configurations but it is possible that other thresholds may improve the results further. The time-intensive RIC and CE experiments of Ch. 3 were used to quantify the potential gains.

Although two configurations are constructed for the CA8, the intention is still that a single configuration would be constructed based on a diverse benchmark suite such as BEEBS. The additional `-Oca8-cbench` configuration in this section is constructed purely to show the impact that the benchmark suite has on the resulting configuration.

Applying the `-Oca8-beebs` configuration to BEEBS and cBench improves performance by up to 40% and gives an overall improvement of 2% compared to `-O3` on each benchmark suite (Fig. 5.3), thus suggesting that the configuration is generally good for the platform. Conversely, `-Oca8-cbench` only offered a few modest improvements amd gave the same overall

```
-O3 -fno-tree-loop-if-convert -fno-tree-loop-distribute-patterns
    -fno-sched-spec -fno-tree-tail-merge -fno-inline-functions-called-once
    -fno-predictive-commoning -fno-tree-ccp -fno-tree-cselim
    -fno-tree-slp-vectorize -fno-tree-scev-cprop -fno-tree-slsr
```

Listing 5.2: The `-Oca8-beebs` configuration

```
-O3 -fno-tree-scev-cprop -fno-tree-bit-ccp -fno-tree-phiprop
    -fno-ira-hoist-pressure
```

Listing 5.3: The `-Oca8-cbench` configuration

performance as `-O3` on both BEEBS and cBench (Fig. 5.4). Therefore, constructing a program-agnostic configuration based on cBench did not improve upon `-O3` overall.

Interestingly, the two configurations only disable flags from `-O3`, they do not enable any additional flags. Eleven flags are disabled by `-Oca8-beebs` and four by `-Oca8-cbench` and they have one flag in common `-fno-tree-scev-cprop`. The `-Oca8-cbench` configuration disables fewer flags than `-Oca8-beebs`; this could be because the less diverse cBench does not exercise as many of the optimisations as BEEBS.

Several flags are common to both `-Ocm3` and `-Oca8-beebs` and therefore particular flags may have similar effects across the Cortex-M and Cortex-A embedded processor families (this will be tested further in Ch. 8).

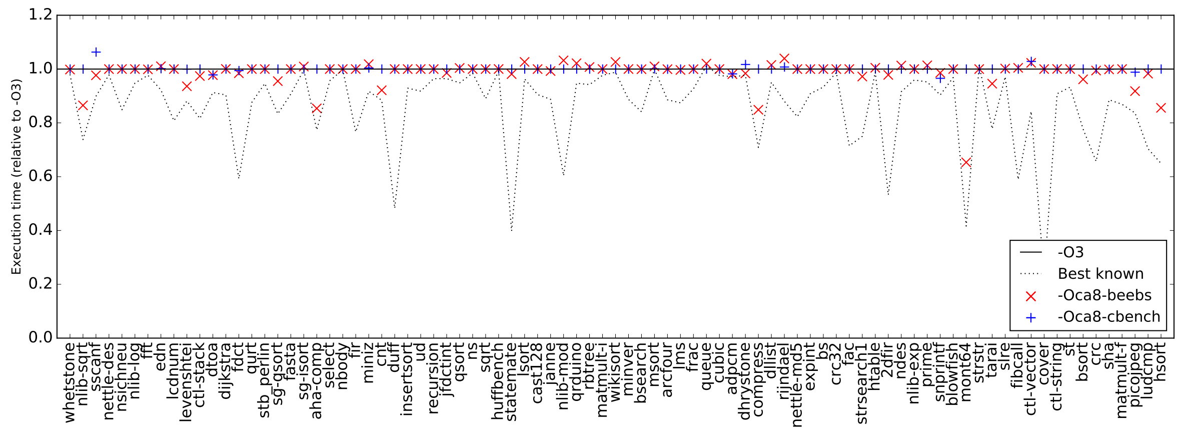


Figure 5.3: Performance of `-Oca8-beebs` and `-Oca8-cbench` on CA8 BEEBS

5.2 Finding a small set of good configurations

This section shows how iterative compilation data can be analysed in order to find a small set of configurations which contains at least one good configuration for each benchmark. The reduced

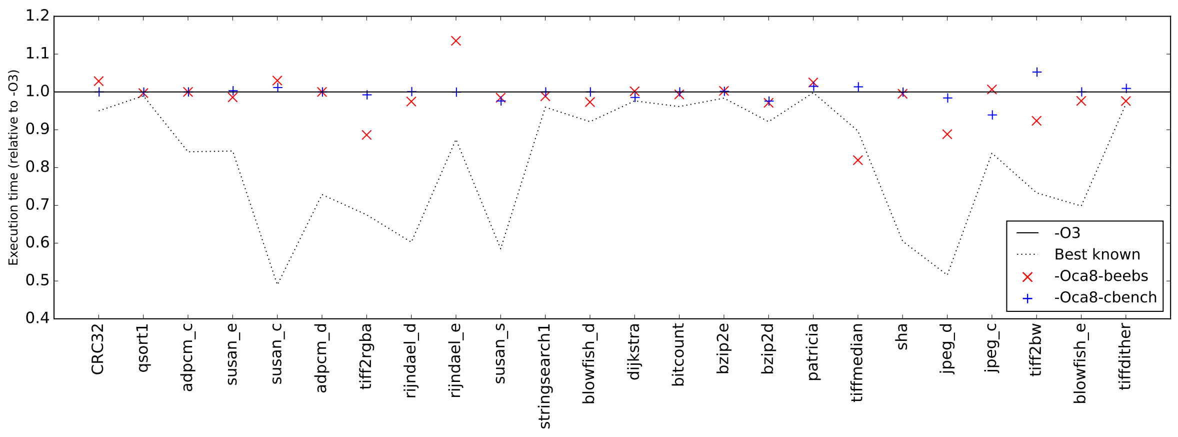


Figure 5.4: Performance of `-Oca8-beebs` and `-Oca8-cbench` on CA8 cBench

set offers a small selection of high quality configurations that can tested on new test programs in less time than other iterative compilation methods.

Finding reduced sets was originally investigated by Purini et al. [63], who aimed to identify a small set of configurations such that there exists at least one configuration that performs greater than LLVM's `O2` for each program. This problem is closely related to the problem of finding the hitting set [42], described as follows: given a set of sets U, find the smallest set H containing at least one element from each of the sets in U. In the context of compiler tuning, U contains a set of good configurations for each benchmark and H is the smallest set that contains at least one good configuration for each benchmark.

Purini et al. developed three methods for generating reduced sets: the first two aimed to cover the most benchmarks and the final method sought to cluster similar configurations and then choose the configuration from each cluster that covers the most programs. One of these approaches is tested in Method 2 (Sec. 5.2.1.2) but later is shown that the task is more suited to Integer Linear Programming which is guaranteed to find the global optimum for a given set size if it exists (Sec. 5.2.1.4).

5.2.1 Method

Methods 1 and 2 aim to find the hitting set (Sec. 5.2), whereas Methods 3 and 4 focus on finding a set that minimises the overall execution time of the benchmarks, thus removing the need to artificially classify each configuration's performance.

5.2.1.1 Method 1: cover least covered benchmark first

A benchmark is covered if the reduced set contains at least one configuration that is good for that benchmark. At each iteration this method selects the uncovered benchmark with the smallest number of good configurations and adds a new configuration to the reduced set in order to cover the benchmark. In the case of a tie (where more than one configuration could be used to cover the benchmark), the configuration that gives the lowest average execution time across all benchmarks is chosen. The algorithm stops when all benchmarks are covered.

5.2.1.2 Method 2: highest covering configuration first

The second method prioritises the inclusion of configurations that cover the most benchmarks. It is adapted from algorithm 2 of [63], but instead of seeking to find configurations that are better than `-O2`, the target is configurations that perform within 5% of the best configuration in the original set.

The algorithm proceeds as follows: find the configuration that covers (is good for) the most benchmarks and add it to the reduced set. In the case of a tie (where two configurations cover the same number of benchmarks), the configuration with lowest average execution time across all benchmarks is chosen.

5.2.1.3 Method 3: reduce overall execution time

The performance of a reduced set can be quantified by summing the best possible execution time yielded by the reduced set for each benchmark. Let $t_{b,c}$ be the execution time relative to `-O3` for benchmark b compiled with configuration c. Then, the performance for reduced set R is given by summing the best possible execution time it gives for each benchmark as follows:

$$\sum_{b=1}^{m} min(t_{b,c}) \qquad c \in R \tag{5.1}$$

At each iteration, Method 3 adds the configuration c' whose addition to the reduced set R gives the biggest reduction in execution time across all benchmarks, which is formalised as follows:

$$\underset{c'}{\arg\min} \left(\sum_{b=1}^{m} min(t_{b,c}) \right) \qquad c, c' \in C, c \in \{c'\} \cup R, c' \notin R \tag{5.2}$$

The advantage of targeting performance directly is that it eliminates the need for an artificial threshold that defines a configuration as good or bad for a particular program.

5.2.1.4 Method 4: Integer Linear Programming

The final method uses Integer Linear Programming to find the optimal reduced set of size k. Integer Linear Programming [24] aims to minimise or maximise a goal whilst satisfying a series of constraints represented as linear inequalities.

In the context of this work, the goal is to minimise total execution time of all the benchmarks such that one configuration is selected for each benchmark and only k configurations are selected overall. The approach was implemented using the GNU Linear Programming Kit (GLPK) [50].

As in the previous method, let $t_{b,c}$ be the execution time relative to `-O3` for benchmark b compiled with configuration c. Let $x_{b,c} = 1$ if c is the chosen configuration for b, otherwise $x_{b,c} = 0$. The goal is to minimise the total execution time for each benchmark when executed with its chosen configuration, as shown by Eq. (5.3).

$$\sum_{b=1}^{m}\sum_{c=1}^{n} t_{b,c} \cdot x_{b,c} \tag{5.3}$$

Constraints were written to ensure that only one configuration is chosen per benchmark (Eq. (5.4)), exactly k configurations are chosen (Eq. (5.5)), each configuration in the reduced set is used at least once (Eq. (5.6)) and no configuration can be used that is not in the reduced set (Eq. (5.7))

$$\sum_{c=1}^{n} y_c = k \tag{5.4}$$

$$\sum_{c=1}^{n} x_{b,c} = 1 \qquad \forall b \in \{1,2,\dots,m\} \tag{5.5}$$

$$y_c - \sum_{b=1}^{m} x_{b,c} <= 0 \qquad \forall c \in \{1,2,\dots,n\} \tag{5.6}$$

$$y_c - x_{b,c} >= 0 \qquad \forall b \in \{1,2,\dots,m\}, \forall c \in \{1,2,\dots,n\} \tag{5.7}$$

Since k must be manually specified, the method was tested for $k = 1$ to $k = n$ (where n is the number of benchmarks) in Sec. 5.2.2.

5.2.2 Results

This section compares the performance of each method when attempting to produce a reduced set of configurations from RIC data (Sec. 5.2.2.1). This is followed by a brief insight into why the task is not well suited to CE data (Sec. 5.2.2.2).

5.2.2.1 Random Iterative Compilation

In Figs. 5.5 to 5.7, Methods 1 to 3 show the average execution time achieved after each configuration is added to the reduced set. For method 4, the average execution time is shown for the sets produced for $k = 1$ to $k = n$.

Methods 1 and 2 terminate once they reach their goal of covering each benchmark at least once. For example, on CM3 BEEBS, Method 1 terminated after 22 configurations and Method 2 after 16 configurations. In contrast, Method 3 continues adding configurations to the reduced set as long as doing so reduces the overall execution time of the benchmark suite. As Method 4 guarantees the best reduction for a given size k, it is consistently the best performer for each k.

Methods 1 and 2 focus predominantly on benchmark coverage rather than overall execution time and consequently they performed the worst overall. Conversely, the performance driven Method 3 is consistently close to the optimal subset shown by Method 4.

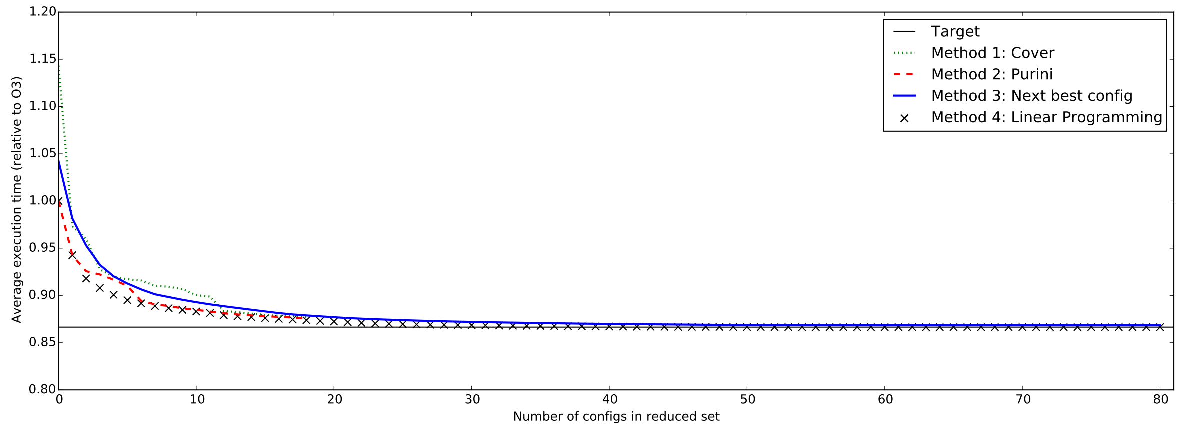


Figure 5.5: Performance of each method for producing a reduced set (CM3)

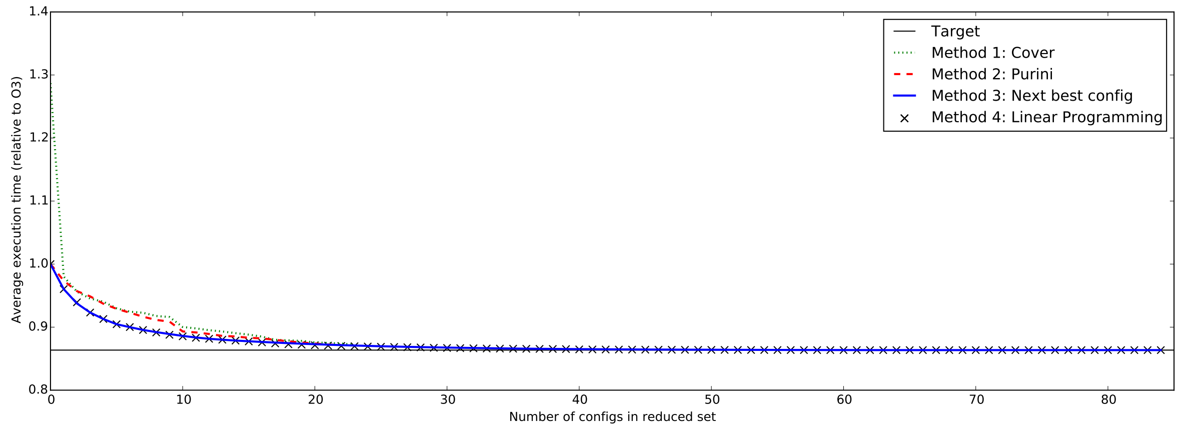


Figure 5.6: Performance of each method for producing a reduced set (CA8 BEEBS)

5.2.2.2 Combined Elimination

The above methods were also tested on CE data but with little success. The difficulty with CE is that it evaluates a different set of configurations for each program. When considering X flags, the first $X+1$ configurations will be common to all programs, but after this, each program may follow a different set of configurations depending on how the algorithm progresses. In the CE data for CM3, there were over 36,000 configurations tested. In contrast, the random iterative compilation experiments provide measurements for all configurations in the random sample for all benchmarks.

5.3 Conclusion

This chapter demonstrated an automatic method for tuning the GCC compiler to a given target architecture. Using the approach, two new optimisation levels were generated, `-Ocm3` and

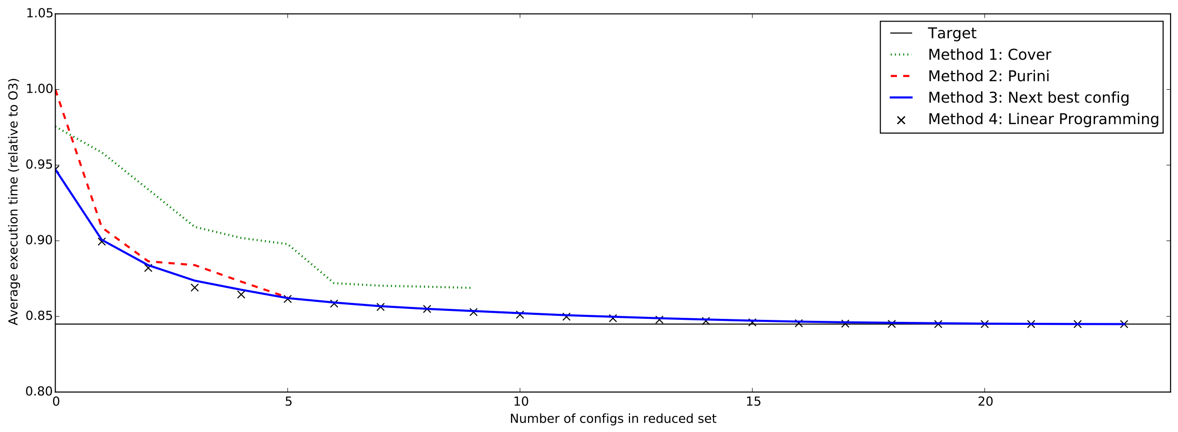


Figure 5.7: Performance of each method for producing a reduced set (CA8 cBench)

`-Oca8-beebs`, that outperform GCC's highest safe optimisation level `-O3` on the CM3 and CA8 by 10% and 2% respectively (Table. 5.1).

These new optimisation levels are offered as platform-specific alternatives to `-O3`. In situations where they might be found to reduce performance the user can simply opt for `-O3`. Choosing between two configurations is much less arduous than the hundreds considered by iterative compilation searches for each new program. While the new optimisation levels offer significant improvements on many benchmarks they do not guarantee the full potential gains of a time-intensive iterative compilation search tailored to a given program. Therefore, the user must decide whether it is worthwhile and feasible to invest considerable extra time in running iterative compilation to optimise their program of choice or simply use a program-agnostic configuration. In any case, it is feasible to try these configurations on any program developed for the CM3 or CA8.

The approach was demonstrated on CE, but in principle, it can be applied to any iterative compilation method by changing the goal from optimising the performance of a single program to optimising the performance of a representative benchmark suite. It is possible that some iterative compilation approaches may be more suited to particular benchmarks, compilers, platforms and optimisations.

In theory, compiler designers can adjust optimisation levels for each architecture. The analyses of the two flags in Sec. 5.1.3 shows it is possible to reason by hand about which flags need to be removed. In practice this happens to some extent, but the fact that these flags were not removed

Table 5.1: Average improvement made by program-agnostic configurations

Platform	Benchmarks	Config	Result
CM3	BEEBS	`-Ocm3`	10%
CA8	BEEBS	`-Oca8-beebs`	2%
CA8	cBench	`-Oca8-cbench`	0%

from `-O3` for these architectures shows there is a need for automated analyses like those explored in this chapter.

This chapter also proposed a new method for finding a small set of configurations that cover a wide range of programs. Running iterative compilation on this small set of configurations produces near-optimal results, but it would still be preferable to be predict a single good configuration for a given program. The next chapter develops an ILP approach for the task of one-shot configuration prediction.

Configurations generated by the methods in this chapter could be useful for robust compiler tuning by testing each of the selected configurations against the GCC test suite to gain confidence that they produce correct behaviour before deploying them to an end-user.

CHAPTER 6

APPLYING ILP TO COMPILER TUNING[1]

This chapter explores the feasibility of applying ILP to the task of compiler tuning in order to automatically discover relevant features directly from a structural relational representation of programs.

Despite the significant gains demonstrated by constructing program-agnostic configurations in Ch. 5, further improvements are available by investing effort into targeting programs individually. It is, however, infeasible to apply time-intensive iterative compilation techniques to each new program and platform pair, therefore recent works have investigated applying machine learning based methods to automatically predict effective configurations (Sec. 2.1).

Most predictive compiler tuning approaches rely on feature vectors of simple program properties that summarise characteristics of the source code. These predefined features are manually selected and may not necessarily be the most relevant for the task.

This study builds on the work of Milepost [27], which tested 1-Nearest-Neighbour (1NN) and decision tree methods for reducing the execution time of a target program given a feature vector describing its characteristics (Sec. 2.1.4.1). One of the key contributions from the Milepost project is the Milepost GCC compiler that produces a Datalog-encoded structural description of C programs based on the Intermediate Representation (IR) used internally by GCC. While Milepost simply used the IR to generate a predefined feature vector of 56 features for machine learning[2], this chapter seeks to apply ILP directly to the IR in order to automatically discover relevant features for the learning problem. A major advantage of targeting the IR directly is that it retains potentially valuable structural information for predicting compiler flags which is otherwise lost when flattening into a feature vector. Furthermore, there is an underlying assumption in existing

[1]Work in this chapter also appears in [13].

[2]The Milepost paper [27] states there are 56 features but the Milepost GCC release actually extracts 55 features – ft56 (number of unconditional branches) is not extracted.

approaches using predefined feature vectors [10, 68, 27] – that the features are relevant across different platforms. The ILP approach builds feature construction into the training phase thus allowing the most relevant features to be chosen for the target platform.

This chapter tests the hypothesis that a relational first-order logic approach which uses IR can exploit structured relations in IR to predict better configurations than a feature vector approach. In order to test this hypothesis, two ILP methodologies are proposed for relating program features to good flags (those that improve performance) and bad flags (those that degrade performance). The first approach (ILP+FV) uses the Milepost feature vector supplemented with extra relations generated using Prolog queries. The second approach (ILP+IR) seeks to discover new features by analysing Milepost's Datalog-encoded IR. In both cases, the CProgol4.4 ILP system [52] is used to learn associations between good and bad flags and program features.

The methods are tested on the CM3 using the BEEBS benchmarks and the results show that ILP+IR outperforms both ILP+FV and the Milepost 1NN feature vector approach. Rules learned by ILP are discussed to show the intuition they provide and how they can also help to tune the method further. Finally, a hybrid approach is suggested that combines the strengths of each method to improve performance further.

The experiments in this chapter are based on 60 benchmarks from an older version of BEEBS before the fixes to input data initialisation were made (Sec. 3.1). They provide an evaluation of a prototype ILP approach from which valuable findings enabled the method to be improved further in Ch. 7.

The rest of this chapter is organised as follows. First the methodology for ILP+IR and ILP+FV is developed (Sec. 6.1), then the methods are tested and compared to Milepost (Sec. 6.2). After analysing the rules learned by the approach, the main conclusions of the chapter are summarised (Sec. 6.3).

6.1 Method

This section develops the methodology for applying ILP to the task of selecting effective compiler flags on embedded software. First a data set is generated based on the BEEBS benchmark suite (Sec. 6.1.1) and then two new ILP approaches are proposed: ILP+FV and ILP+IR. ILP+FV replicates the Milepost approach within an ILP setting using rule learning instead of 1NN (Sec. 6.1.2), while ILP+IR introduces IR which provides a lossless representation of the code and contains potentially useful learning information that is not present in the feature vector but which ILP can target directly (Sec. 6.1.3).

6.1.1 Identifying examples of significant flags

Distinguishing which individual flags have a significantly good or bad influence on a program's performance is a hard task due to the complex and often not well understood interactions that can occur between them [19, 57]. Since it is infeasible to perform an exhaustive search to obtain an exact model of flag effects, this study uses iterative compilation to gather data on a subset of the configuration space that is then analysed to identify good and bad flags.

To make a fair comparison with Milepost, the same iterative compilation technique, RIC, was used to study the effects of 133 flags that are available when compiling for the CM3 with GCC 4.9.3 (Sec. 3.4). As in Ch. 3, 1000 random configurations were generated by selecting `O1`, `O2` or `O3` with probability $p(\frac{1}{3})$ and then explicitly enabling or disabling each of the 133 flags with $p(\frac{1}{2})$. Each benchmark program was compiled with each configuration and the resulting execution times were recorded.

The rest of this section proposes and evaluates three methods for finding good and bad flags from iterative compilation data.

6.1.1.1 Option 1: Average performance of configurations in which the flag is enabled/disabled

For each program and flag, divide the configurations into two sets: C_1 in which the flag is present and C_0 in which the flag is not present. If the average performance of C_1 is better than C_0 then the flag is defined as good for performance, otherwise it is defined as bad for performance.

6.1.1.2 Option 2: Appearances in configurations above median performance vs appearances in configurations below median performance

For each program, order the configurations by performance (best first), then find the top half of configurations C_{good} and bottom half of configurations C_{bad}. If a flag appears more often in C_{good} than C_{bad} then it is classified as good for performance, otherwise it is classified as bad for performance.

6.1.1.3 Option 3: Appearances in good configurations

This option classifies the flags based on the IID probability density function in Eq. (2.1). For each program, let C_{good} be the set of configurations that performed within $\theta\%$ of the best known configuration. The frequency that a flag appears in C_{good} is used to estimate the likelihood that the flag leads to good performance.

For example, classify a flag as good for performance if it appears more than $x\%$ in C_{good} and bad if it appears less than $y\%$. Note that flags which have no effect on performance may appear frequently (or infrequently) in C_{good} by chance. To reduce such noise, flags falling between thresholds x and y could be classified as neither good nor bad (a machine learning system would need to decide how to treat such flags, for example: always switch on, always switch off, or enable with $p(0.5)$).

6.1.1.4 Evaluation

This evaluation quantifies how closely each flag selection method approximates the performance given by the RIC data. The three proposed methods were evaluated by applying the following steps for each program:

1. Use the method to identify good and bad flags from RIC data.
2. Construct a configuration by enabling good flags and disabling bad flags.
3. Compare the program's performance with the constructed configuration compared to the best RIC result.

As Option 3 has tunable parameters, it was tested with a range of values for θ, x and y.

The average performance achieved by each option is shown in Fig. 6.1 relative to best RIC. Option 3 gave the best performance and therefore found the most beneficial classification of good and bad flags out of the three options. Performance was close to best RIC when the thresholds were set to $\theta = 2\%, x = 40\%$ and $y = 40\%$.

Although $\theta = 2\%$ gave the best performance, there are few configurations per program that fall within this threshold. Fewer configurations means less confidence in the selected flags as there is a higher likelihood that a flag appears frequently (or infrequently) by chance. With $\theta = 2\%$ there are 31 programs with fewer than five good configurations. Relaxing the threshold to 5% gives 16 programs with less than five good configurations but still gets close to best RIC. The value of θ controls a trade-off between the number of configurations from which to select good and bad flags and the quality of those configurations and the flags selected.

The poorer performance of Options 1 and 2 suggests that analysing bad configurations does not necessarily aid the identification of good and bad flags. A good flag may appear frequently (and a bad flag may appear infrequently) in both good and bad configurations. For example, further analysis shows that on the `recursion` benchmark, the flag `-foptimize-sibling-calls` is disabled in the top 15 configurations but also 43 out of the 50 worst configurations.

The results also suggest it is better to assume a flag is good rather than bad, for example, setting a very high threshold for a good flag (e.g. $x = y = 80\%$) gave much worse performance than setting a very low threshold (e.g. $x = y = 20\%$). Therefore, the ILP prototype will be developed with the goal of determining which flags are bad for a given program and should therefore be disabled (note this is also the goal of CE). This will be captured by the predicate `badFlag/2` rather than `goodFlag/2` which appears in the earlier motivating example (Sec. 2.2.1).

6.1.1.5 Summary of selected approach

The most significant flags for each program were identified by analysing good configurations (defined as those with an execution time within 5% of the best configuration). A flag was identified as good for performance if it appeared in at least 75% of good configurations or bad if it appeared in

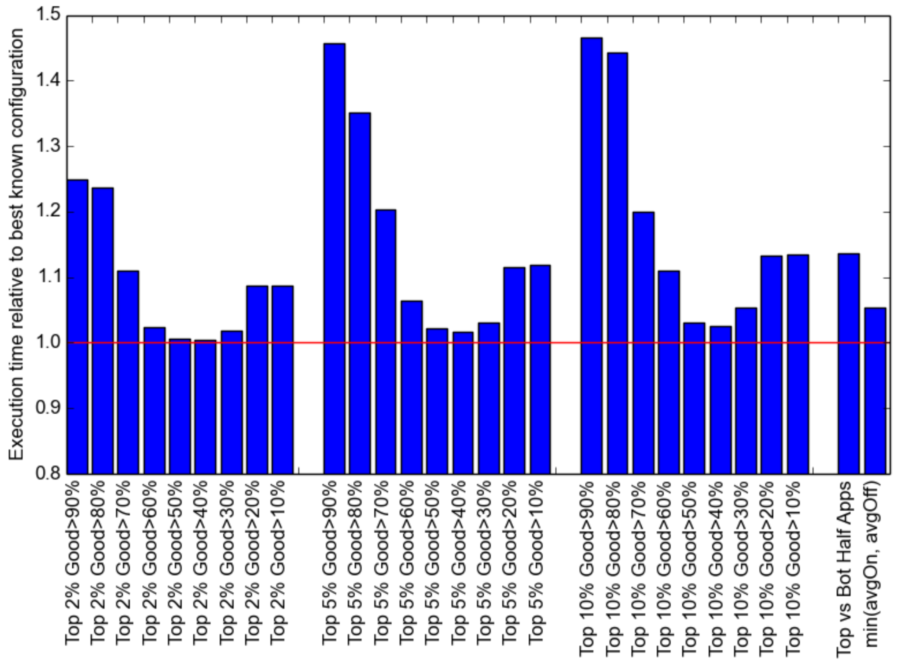


Figure 6.1: Comparison of flag selection methods

no more than 25% of good configurations.[3] These parameters, determined by further preliminary testing, are intended to filter out insignificant flags and prevent them from adding noise to the training set.

Good and bad flags were turned into negative and positive examples of the predicate `badFlag/2` (as the results in Sec. 6.1.1.4 suggest it is beneficial to selectively disable flags). For example, if flag x were good and flag y were bad for program a then this would be represented as:

```
:- badFlag(a,x).              % good flag
badFlag(a,y).                 % bad flag
```

The approach taken assumes that the flag effects are independent and does not try to capture dependencies between flags. As with any approach using the IID model [3, 27] there may be false interactions that reduce performance, for example, two flags that have a positive effect when applied separately, could have a negative when applied together. However, independent approaches are still beneficial as long as the gains of predicting individual flags outweigh any losses from negative interactions.

It was also considered to train and predict at configuration level, rather than flag level, by supplying examples of good and bad configurations instead of flags. In theory, this would help

[3]Sixteen programs with fewer than five good configurations were excluded from the training set as there were not enough data to approximate the effects of individual flags. This limitation is overcome in the next chapter (Ch. 7)

capture dependencies between flags (as they are embedded in the configuration itself) but there are several disadvantages. Learning a relationship between program structure and a whole configuration is a more complex task than relating the performance of individual flags to program structure. Each program would need be to trained on a common set of configurations in order to generalise well, therefore iterative compilation methods that test programs on vastly different sets of configurations (e.g. CE, GA) would not be applicable as the training data would be too sparse. Furthermore, for each program there are typically few good configurations and many more poor configurations (Sec. 3.4) thus causing a great imbalance between the number of positive and negative examples. Finally, a configuration level approach treats the configuration as a whole whereas flag level can ignore unimportant flags.

6.1.2 Extracting program features for ILP+FV[4]

In order to make a fair comparison with Milepost and focus on testing the effect of rule learning and IR, the ILP+FV method uses Milepost GCC to extract the feature vector for the most time consuming function of each benchmark, which was found by profiling each program using the `gprof` tool on an x86 processor.[5] Profiling in this way enables learning to be concentrated on the most characteristic function of the program and filters out sections of the code that may have little effect on performance. Each feature vector was normalised by number of instructions (feature 24) and converted into a series of Prolog facts to enable interpretation by Progol.

Further predicates were created to allow Progol to summarise and compare between the features of each program using quartiles and averages. For example, `qt(P,Ft,Q)` gives the quartile Q (1, 2, 3 or 4) that feature Ft is in for program P.

6.1.3 Extracting Intermediate Representation for ILP+IR

Since it is not known whether the feature vector contains the best features for the learning task, the ILP+IR method seeks to automatically discover relevant features during learning rather than predefine them. In Milepost GCC [27], the feature vector is generated by applying Prolog rules to the Milepost IR. This study aims to learn directly from the IR as it provides a lossless representation of the code which may contain potentially relevant structural information that is not present in the feature vector. In general, any IR could be used for the background knowledge (for example, LLVM IR), but this study favours the Milepost IR since it encodes precisely the internal data structures over which the GCC optimisations are applied.

To allow comparisons to be consistent with ILP+FV and Milepost, the Milepost IR was extracted for the most time consuming function of each benchmark and used as the basis for

[4]Early versions of the ILP+FV approach appear in [12] and [11].

[5]As it is very difficult to obtain function-level execution time for the Cortex-M3, the most time consuming function was estimated by profiling an x86. Each function should be called the same number of times on each architecture, but relative function costs may vary.

the background knowledge. The original Milepost IR used a separate file for each function and the resulting Prolog clauses did not define the program or function to which they referred. In this ILP methodology, program and function names are added to each predicate to guarantee uniqueness. Suppose `edge_src(ed0,bb1)` describes an edge in function `f` of program `p`, then this would become `edge_src(p,f,ed0,bb1)`.

Some of the predicates within the Milepost IR were too specific to allow meaningful generalisations, therefore ILP+IR includes more general rules to group such cases. For example, the facts `assign_class(P,F,st1,minus_expr)` and `assign_class(P,F,st2, plus_expr)` denote two statements `st1` and `st2` that contain a subtraction and addition respectively (further predicates determine whether it is an integer or floating point operation). Since addition and subtraction are handled the same in hardware, it is sensible to group the two classes by adding the following rules:

```
assign_addsub(P,F,S) :- assign_subcode(P,F,S,plus_expr).
assign_addsub(P,F,S) :- assign_subcode(P,F,S,minus_expr).
```

The IR also contains an `expr_code(P,F,E,C)` predicate which asserts that expression `E` belongs to expression class `C` (for example, integer constant, variable declaration or array reference). The following rule was added to identify expression classes that appear more than once in a function `F`.

```
expr_code2(P,F,C) :- expr_code(P,F,E1,C), expr_code(P,F,E2,C),
                     E1 @< E2.
```

Other predicates contained numerical data such as probability `Pr` of an edge being taken, denoted by `edge_prob(P,F,E,Pr)`. The following rules were added to allow broader comparisons between edge probabilities as follows:

```
edge_prob_low(P,F,E)  :- edge_prob(P,F,E,N), N < 0.5.
edge_prob_high(P,F,E) :- edge_prob(P,F,E,N), N >= 0.5.
```

6.1.4 Learning rules for bad flags

In both ILP+FV and ILP+IR, Progol learns rules for `badFlag/2` which are used to predict which of the 133 flags should be switched off for good performance on a given program. The predicted configuration is constructed by enabling `O3` then explicitly disabling the flags identified as bad and enabling the remaining flags.

In ILP+FV, Progol learns rules for `badFlag/2` using the following mode declarations (Sec. 2.2) to restrict the search space:

```
:- modeh(*,badFlag(+program,#flag))?
:- modeb(*,large_ft(+program,#ft))?
:- modeb(*,small_ft(+program,#ft))?
:- modeb(*,non_zero(+program,#ft))?
:- modeb(*,ft(#ft,+program,0))?
:- modeb(*,qt(+program,#ft,#any))?
```

These allow learned rules to use the knowledge that a feature for a program is larger or smaller than average, non-zero, zero or within a particular quartile. To avoid over-fitting, the rules are not allowed to contain exact feature values (other than zero).

In ILP+IR, Progol learns from the Milepost IR, rather than the feature vector, using modeb declarations such as the following:

```
:- modeb(*,edge_src(+program,-func,-edge,-bb))?
:- modeb(*,edge_dest(+program,-func,-edge,-bb))?
:- modeb(*,edge_prob_low(+program,-func,-edge))?
:- modeb(*,edge_prob_high(+program,-func,-edge))?
:- modeb(*,expr_code2(+program,-func,#any))?
:- modeb(*,assign_addsub(+program,-func,-stmt))?
```

These declarations allow rules to contain knowledge of edges between source and destination basic blocks, edges with a high or low probability of being taken, classes of expressions that appear more than once and assignments featuring addition or subtraction. The last four declarations feature some of the new predicates that group elements of the IR to aid the learning of general rules (Sec. 6.1.3).

6.2 Results and evaluation

The results of testing the ILP prototype are presented in two parts: first the ILP+FV and ILP+IR methods are compared to Milepost (Sec. 6.2.1) and then a hybrid approach is suggested that builds on the individual strengths of ILP, 1NN-prob and `-O3` to further improve predictive performance (Sec. 6.2.2). Each method was evaluated with leave-one-out cross-validation.

6.2.1 ILP+FV and ILP+IR

The ILP+FV method outperformed the 1NN-prob approach used by Milepost in 32 out of 60 benchmarks. The ILP method was further improved by training on the more expressive IR rather than the feature vector. The ILP+IR method outperformed ILP+FV and 1NN-prob in 39 out of 60 benchmarks each. Furthermore, ILP+FV and 1NN-prob performed worse than `-O3` as they increased average execution time by 6% and 7% respectively, while ILP+IR made a 1% improvement (Table. 6.1).

Figure 6.2 shows that each predictive method performed well on different programs and that no one of these approaches was best across all benchmarks. Therefore a better strategy might be

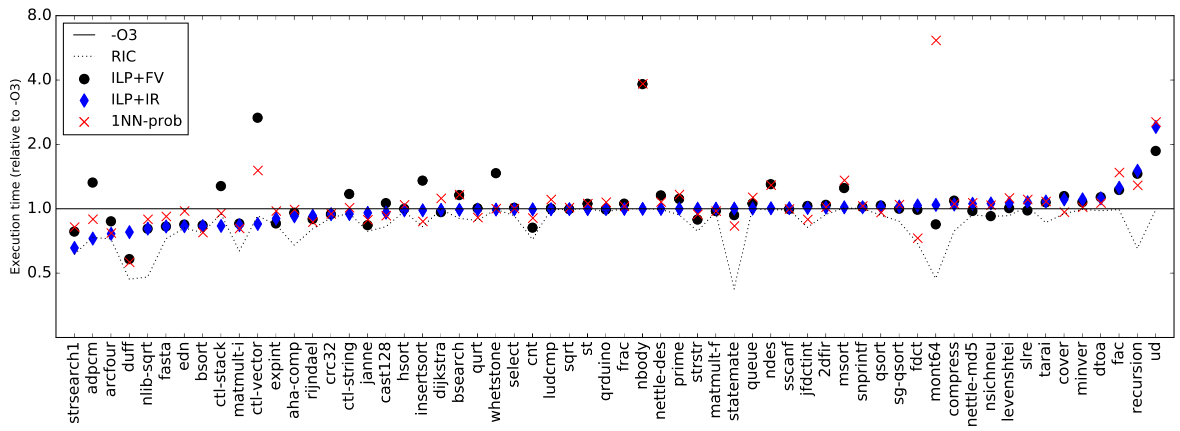


Figure 6.2: ILP+FV, ILP+IR, 1NN-prob and best of 1000 random configurations

to apply a number of predictive approaches to the target program and select the configuration which gives the best result. Section 6.2.2 tests the gains of such a hybrid approach. The 1NN-prob approach is likely to perform well when the training set contains a program that is similar enough to the test program and benefits from similar configurations. In cases where there is no such similar program, then ILP can still pick out important characteristics that are common between programs in order to make good predictions.

The ILP+IR method was the only predictive approach to outperform RIC on any of the benchmarks. The execution times achieved by ILP+IR for `ctl-stack` and `ctl-vector` were over 5% better than the best result found after testing 1000 random configurations (which takes over 2.5 hours per benchmark compared to ILP+IR which takes under 10 seconds to predict and evaluate its selected configuration once trained).

One of the advantages of ILP is that it produces human-readable rules that show why certain flags were predicted.[6] The rules can be easily translated into English sentences and the IR should be familiar to GCC developers. For example, the following rule:

```
badFlag(P,'-fguess-branch-probability') :-
    stmt_code(P,F,S,gimple_cond),expr_code2(P,F,real_type).
```

can be translated as *"the -fguess-branch-probability flag should be disabled if function F of program P contains a conditional statement S and at least two floating point expressions"*. The guess branch probability optimisation targets conditional statements, therefore it is plausible that any program affected by it should contain a conditional statement. Secondly, the Cortex-M3 hardware does not have a floating point unit [5], instead it relies on the compiler to emulate floating point functionality using integer operations, which is slower and therefore floating point

[6]The full set of rules is available at `github.com/craigblackmore/logiflag`

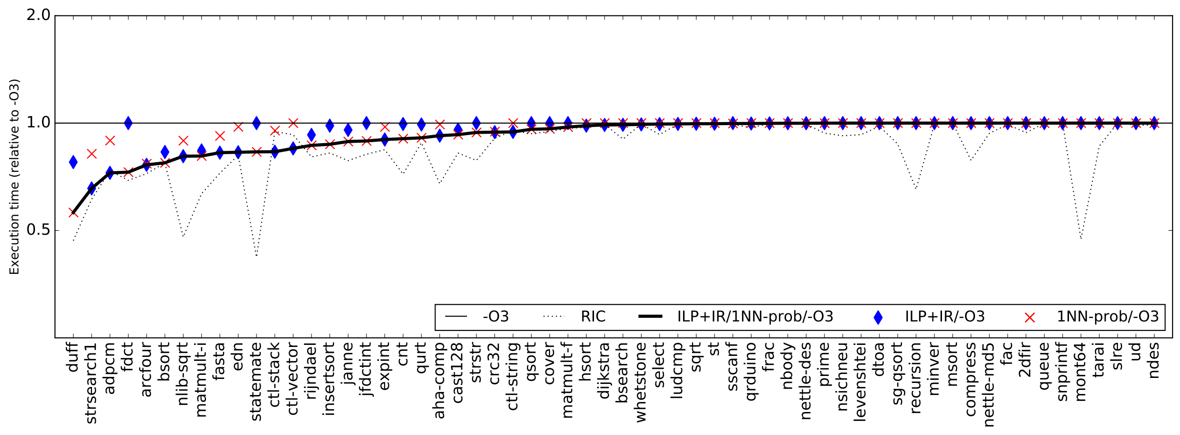


Figure 6.3: ILP+IR/1NN-prob/O3 hybrid and best of 1000 random configurations

operations are more expensive than integer operations and mispredictions would have more of an impact.

Interestingly, the above rule uses the `expr_code2(P,F,C)` predicate introduced in Sec. 6.1.3 to capture expression classes that appear more than once in a given function. Several of the other rules also use this predicate including `badFlag(P,'-fschedule-insns2') :- expr_code2(P,F,array_ref)` which states that *"the -fschedule-insns2 flag should be disabled if function* `F` *of program* `P` *contains at least two array references"*. Here ILP has found a potentially useful feature in the IR which is not present in the flattened feature vector. In fact, none of the attributes in the feature vector refer to array references.

The rules can also identify potential gaps in the training set. For example, further analysis is required to determine if the rule `badFlag(P,'-fschedule-insns') :- expr_int_size(P,F,E,16)` (or *"-fschedule-insns should be disabled if program P of function F contains a 16-bit integer"*) is feasible. This could be achieved by writing a program for which this flag improves performance in order to add a counter-example to the training set.

The ILP method also identified 19 flags that should always be disabled for the target platform. These were learned as facts such as `badFlag(P,'-fsched-spec')` and `badFlag(P,'-funroll-loops')`. Using ILP facts to construct new program-agnostic optimisation levels will be explored later in Sec. 7.3.

6.2.2 Hybrid approach

Given the observation that ILP, 1NN-prob and `O3` performed better on different benchmarks, this section evaluates a hybrid method that tests the configurations given by ILP, 1NN-prob and `O3` and selects the best result. This hybrid approach reduced the average execution time by 8% compared to `O3`.

Over half of the programs gained a positive benefit from the hybrid approach (Fig. 6.3), but there remains a small set of programs for which none of the methods were able to reduce their execution times even though the random approach shows a speed-up is possible. This may be due to lack of knowledge about certain flags in the data set. For example, further analysis of the `fdct` program showed that most of its speed-up can be obtained by disabling the `-ftree-fre` flag, but there exists no other program in the training set for which this was identified as a bad flag. Therefore, when `fdct` is removed during leave-one-out cross validation, none of the remaining examples allow a rule for this flag to be learned. In future, the addition of new training programs would help to increase the number of examples for such flags.

6.3 Conclusion

This chapter introduced and evaluated a novel ILP-based method for predicting effective compiler flags which outperformed the state of the art. Learning from declarative IR rather than a predefined feature vector significantly improved the performance of the method (Table. 6.1). This study also demonstrated the human-readable rules that a ILP based compiler tuning method can produce.

In some cases, 1NN-prob and/or `-O3` performed better than ILP, therefore there is still room for improvement in the design of the ILP approach. The next chapter Ch. 7 will address some limitations in the prototype ILP approach in order to produce a more accurate model, further gains and more meaningful rules.

Table 6.1: Average improvement made by ILP prototype

Platform	Benchmarks	1NN-prob	ILP+FV	ILP+IR	Hybrid
CM3	BEEBS	-7%	-6%	1%	8%

CHAPTER 7

FOCUSING ILP BASED COMPILER TUNING[1]

This chapter builds on the proof-of-principle ILP implementation developed in the previous chapter (Ch. 6) and investigates how ILP can be applied in practice to learn meaningful rules that relate program structure to effective compiler flags. The contribution centres around three key aspects that make the approach work in practice. A sensitivity analysis is used to find robust examples of good and bad flags for each program; the method is trained using CE rather than RIC as Ch. 3 shows the latter finds less optimal configurations on the platforms targeted in this thesis; and the search space of the learner is restricted to allow large functions to be targeted. The new approach achieves an average 7% speed-up over `-O3` and significantly outperforms the proof-of-concept and related feature vector based approaches. Finally, the transferability of the method to other platforms is demonstrated by testing on the ARM Cortex-A8 processor.

The rest of this chapter is structured as follows. First limitations and lessons learned from the ILP prototype are analysed (Sec. 7.1) and then an improved design is developed and evaluated that produces more accurate models and also generates competitive program-agnostic configurations in the process (Sec. 7.2).

7.1 Limitations of initial approach

Following the implementation and testing of ILP+IR, some limitations with this approach are identified: using RIC misses some of the gains available from more intelligent iterative compilation methods such as CE; the method for identifying good and bad flag examples is sensitive to noise in random data and low confidence when there is only a small number of good configurations; the time taken to construct the bottom set for some examples is prohibitively

[1]Work in this chapter also appears in [14].

large; some rules get caught in infinite loops when deployed on the IR for some target programs; Progol's cost function prevents more complex but potentially useful structures being captured; and the search of the rule space is inefficient.

7.2 Improved Inductive Logic Programming based compiler tuning

This section improves on the original ILP approach by overcoming practical limitations of the method. Firstly a sensitivity analysis is used to leverage robust examples of good and bad flags from CE data rather than RIC. Then it is explained that the original approach is unable to target large functions but restricting the search space of the learner can overcome this. Next the ILP system is changed in order to improve the rule search by changing the fitness function used to evaluate each candidate rule. Finally, the results of 10-fold cross-validation are presented and insights gained from the method's declarative, human-readable rules are discussed.

7.2.1 Identifying significantly good or bad flags

The investigatory study in (Ch. 3) showed that CE found better configurations than RIC and in a shorter amount of time. Therefore, CE is chosen as the method for generating training data rather than RIC used by the original ILP approach Ch. 6.

The original ILP method used a probabilistic approach to identify good or bad flags (Sec. 6.1.1) by classifying good flags as appearing frequently in good configurations and bad flags as appearing infrequently. However, since the configurations were produced using random sampling this leads to noisy examples because flags may appear in good configurations by chance (this is increasingly likely when the set of good configurations is small). In fact, the original method excluded 16 programs from training because they had fewer than five good configurations.

Furthermore, the probabilistic approach is not suited to CE data because while it is likely that disabled flags are indeed bad flags, the remaining enabled flags could either be good or have a negligible effect on performance. The probabilistic approach cannot discriminate between good and negligible flags and produces too many examples, many of which are unimportant and this not only confuses the learner but increases the complexity of the problem as there are more examples to consider.

Evidence of noisy examples can be seen in the training data for the original method where `-fvect-cost-model` was classified as good or bad for three programs. This flag chooses the cost model that will inform vectorization decisions [69] but as the CM3 has no vector unit, the flag should not have an impact and should not appear in any examples.

This chapter proposes a sensitivity analysis to find flags that exhibit a significant impact on the target program without leaving the classification to chance. For each program, first the best configuration found by CE is selected. Then each enabled flag is disabled one-by-one – if disabling the flag makes performance over $\theta\%$ worse, then it is classified as a good flag. Similarly, each disabled flag is enabled one-by-one and, if enabling the flag degrades performance by more than $\theta\%$, it is classified as a bad flag. Based on preliminary experiments this study uses $\theta = 5\%$.

This sensitivity analysis gives more reliable examples and it allows all programs to be included in training, even if only one good configuration has been found. With the previous approach, it was impossible to determine whether the flags had appeared in a single good configuration by chance. In addition, this sensitivity analysis can be used with any iterative compilation approach chosen by the user.

7.2.2 Restricting the search space

Re-evaluating the original ILP approach uncovered a practical limitation that prevented learned rules from being applied to larger functions. Each rule disables a particular flag if the target program contains specific structures in its IR. In order to assert that a rule does not hold for a given program (and therefore the flag should be enabled), the whole of the IR must be searched to confirm the absence of the specified structures. When applying the learned rules to predict for a large program such as `picojpeg` the query failed to terminate after running for one day.

To enable large functions to be targeted, the new method restricts the search space to only construct rules at CFG and statement level based on statement type (e.g. assignment, condition), the order in which statements appear and the types of edges connecting basic blocks in the CFG. This excludes expression level information where the size of the IR explodes e.g. for `picojpeg` there are 31,623 Datalog facts that describe expression level information compared to 4,487 at the statement level.

7.2.3 ILP search algorithm and cost function

The ILP method constructs rules by using elements of the IR to try and distinguish between positive and negative examples. Since the IR for each program is so large, it is infeasible to test every possible rule that could be constructed from the IR. Therefore, ILP uses a search algorithm to attempt to find good rules efficiently. Crucial to an efficient search is the cost function used to evaluate the fitness of each rule.

Closer examination of the training phase shows that the ILP search misses large portions of the IR and can become stuck in local optima trying similar rules that do not improve much rather than exploring other parts of the IR. In addition, further analysis shows that the CProgol4.4 ILP system [52] used by the original ILP method only produces rules containing three or fewer literals in the body, even when the maximum allowed body length is increased. This is due to CProgol4.4's use of a 'compression' cost function which penalises longer rules. For the purpose of this compiler tuning task, short rules are unable to capture deeper structures within the code and are limited to picking out a few elements that may or may not be structurally connected.

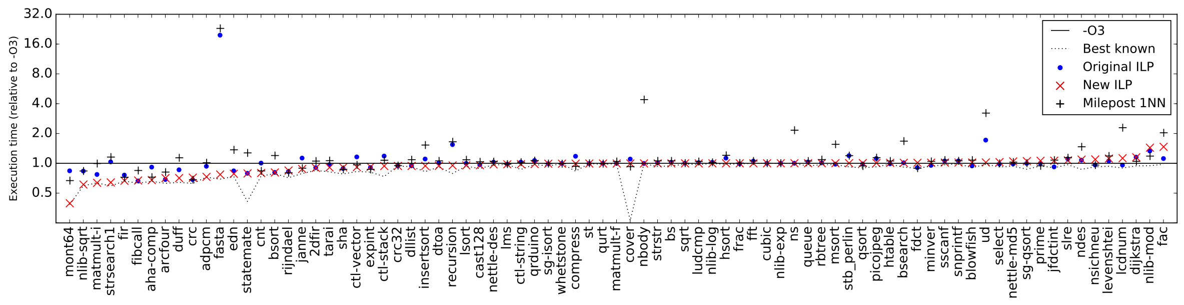


Figure 7.1: Performance of new ILP compared to original ILP approach and Milepost 1NN (logarithmic scale)

The new approach was implemented using an alternative ILP system called ALEPH [66] that allows a custom choice of search algorithm and cost function. The selected approach uses a best-first heuristic search similar to that used by CProgol4.4 but instead of the ‘compression’ cost function it uses ‘coverage’ which only penalises coverage of negative examples (and not rule length). This allows the learner to find longer rules and potentially more useful structural features.

7.2.4 Cross-validation of new ILP approach

10-fold cross-validation experiments were conducted to evaluate the new ILP method and it achieves an overall 7% improvement compared to `-O3`, a 9% improvement over the re-evaluated original ILP method and 18 % over the Milepost 1NN feature vector based approach. Out of the 81 programs, the new approach outperforms `-O3` on 50 programs, the original ILP on 51 programs and Milepost on 64 programs (Fig. 7.1).

The original ILP study (Ch. 6) used a ‘hybrid’ approach which tried `-O3`, ILP and Milepost and for each program chose the configuration that gave the best performance out the three. The new method now performs well as a standalone approach, but in cases where it might perform worse than `-O3` the user can still simply opt for `-O3`.

7.2.5 Insights from human-readable rules

This section gives an example of one of the declarative human-readable rules learned by the new approach.[2] The rule is as follows:

```
badFlag(Prog,'-fguess-branch-probability') :-
  bb_stmt_n(Prog,Func,BB,St1,St2),
  bb_stmt_n(Prog,Func,BB,St2,St3),
```

[2]The full set of rules is available at `github.com/craigblackmore/ilp-focused`

```
    bb_stmt_n(Prog,Func,BB,St3,St4),
    stmt_code(Prog,Func,St1,gimple_assign),
    stmt_code(Prog,Func,St2,gimple_call),
    stmt_code(Prog,Func,St4,gimple_call),
    edge_flag(Prog,Func,Edge,false).
```

This can be read as '-fguess-branch-probability is a bad flag for program $Prog$ if function $Func$ contains an assignment statement followed by two function calls and $Func$ also contains an else clause'. The rule covers four programs (`picojpeg`, `nsichneu`, `tarai` and `recursion`).

The `-fguess-branch-probability` flag enables the compiler's branch prediction heuristics but the rule suggests that these heuristics perform poorly on the identified programs. Further analysis of two of the programs `tarai` and `recursion` shows that the enabled flag causes a sub-optimal ordering of basic blocks leading to more branches being taken than when the flag is disabled. Although both `tarai` and `recursion` are recursive programs which would not typically be used on embedded platforms, additional experiments show that this is an important insight as the same effects were observed on an Intel i5 processor.

7.3 Constructing a new program-agnostic optimisation level

This section shows how the new ILP method led to the construction of a new standard optimisation level, `-Ocm3-ilp`, for the CM3. The name `-Ocm3-ilp` distinguishes from the `-Ocm3` level constructed in Ch. 5.

7.3.1 Performance of `-Ocm3-ilp`

The rules learned by the new ILP approach include seven facts that suggest seven flags should always be disabled on the CM3. Based on these facts, a new configuration `-Ocm3-ilp` was constructed by enabling all 133 flags except the seven identified as always bad. The new configuration outperforms `-O3` by 6% which is close to the 7% average achieved by the new ILP method (Fig. 7.2). On a program-by-program basis it is clear that the ILP rules had a positive influence on some programs therefore some useful rules have been learned. On some programs, `-Ocm3-ilp`, gives a better speed-up which suggests that the ILP approach may have learned some counter-productive rules. The white-box declarative nature of ILP enables future work to investigate these cases further and attempt further refinements to the approach.

Both the `-Ocm3` and `-Ocm3-ilp` configurations perform comparably, therefore analysing the flags with a sensitivity analysis as in `-Ocm3-ilp` is competitive with using iterative compilation to search directly for the `-Ocm3` program-agnostic configuration.

Compared to `-O3`, the `-Ocm3-ilp` configuration disables seven flags and enables 21 flags while `-Ocm3` enables three and disables 20. Together `-Ocm3-ilp` and `-Ocm3` have four disabled and

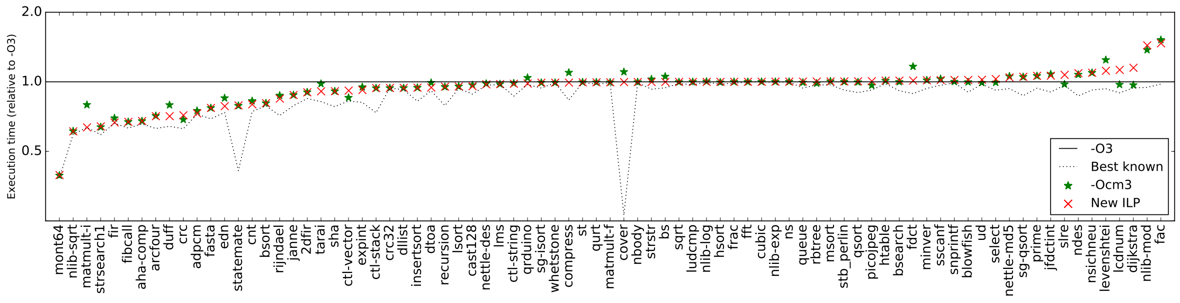


Figure 7.2: Performance of `-Ocm3-ilp` on the Cortex-M3 (logarithmic scale)

three enabled flags in common. Interestingly, the `-Ocm3-ilp` configuration disables `-fcommon` and `-ftree-loop-if-convert` which `-Ocm3` also determined should be disabled for the CM3 Sec. 5.1.3.

7.4 Testing on the Cortex-A8 processor

To demonstrate that the new ILP method can also optimise for other embedded platforms, this section trains and tests models for the Cortex-A8 processor. The CE data generated in Ch. 3 is used to create two rule-based models: one trained on BEEBS and one trained on cBench. The models are tested using 10-fold cross-validation.

The BEEBS model yields some significant gains, but the overall improvement is a modest 1% (Fig. 7.3). The learned rules for CA8 include facts that suggest seven flags should always disabled. These flags were used to construct a new standard optimisation level for the CA8, `-Oca8-ilp-beebs`, but unlike its CM3 counterpart `-Ocm3-ilp`, this configuration performed worse than `-O3` by 3% on average. The several cases where ILP outperforms `-Oca8-ilp-beebs` suggest that ILP has learned important rules for optimising those particular programs that the facts alone do not cover.

The cBench model follows a similar trend (Fig. 7.4). The ILP model performs slightly worse than `-O3` by 1% but the constructed `-Oca8-ilp-cbench` performs in line with `-O3`. In contrast to the BEEBS model, `-Oca8-ilp-cbench` sometimes outperforms ILP by a large margin, suggesting that the ILP model disables one or more flags that would have been beneficial to performance.

Overall, ILP provided less benefit on the CA8 than the CM3, which follows the same trend as the other predictive approaches. The method of selectively disabling flags using the `badFlag` predicate does not translate well to the CA8 because `-O3` is a better starting point than enabling all 133 flags on this processor. To test this hypothesis, an additional model was constructed by learning `goodFlag` and `badFlag` predicates to add and remove flags from `-O3`, rather than purely disabling from the set of all flags. This gave a higher improvement of 3% overall.

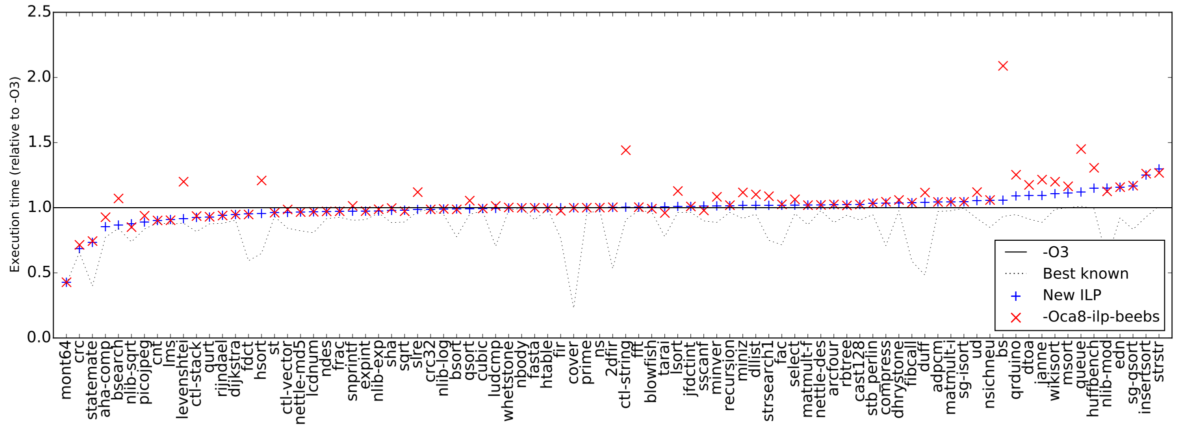


Figure 7.3: Performance of new ILP method and `-Oca8-ilp` on CA8 BEEBS

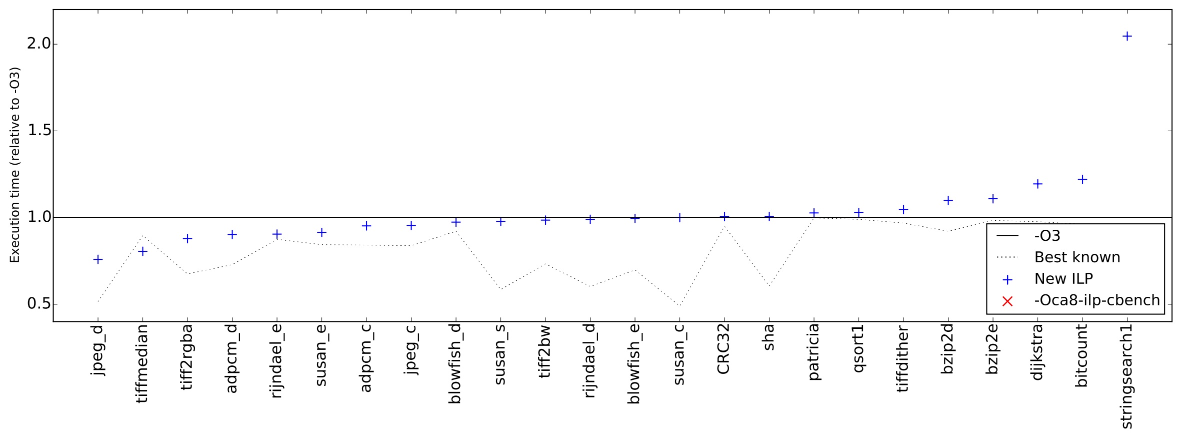


Figure 7.4: Performance of new ILP method and `-Oca8-ilp` on CA8 cBench

7.5 Conclusion

This chapter presented the following three key techniques to make ILP based compiler tuning work in practice. The sensitivity analysis produces robust examples of significant flags for each program and enables any iterative compilation approach to be used in training; the method is trained with CE rather than RIC (as CE finds more optimal configurations in a shorter amount of time) and restricting the search space allows large functions to be targeted.

This chapter developed a viable ILP based compiler tuning method that is competitive with other predictive approaches (Table. 7.1). The new method also facilitated the construction of new optimisation levels, `-Ocm3-ilp`, `-Oca8-ilp-beebs` and `-Oca8-ilp-cbench`, that are tailored specifically to the CM3 and CA8 processors. While the CM3 configuration outperforms `-O3` overall, the CA8 configurations performed in line or slightly worse than `-O3`. The smaller improvement given by program-agnostic configurations on the CA8 could be due to `-O3` being better optimised for the general case on the CA8 than on the CM3.

In future work, ILP could be used to predict flag parameters as well as learn rules that capture dependencies between groups of flags. Weights could also be applied to each flag example to express its significance.

In addition, different intermediate representations could potentially yield more accurate models. There are some limitations with the Milepost IR. Firstly, it is a large representation (for example, the `2dfir` source code contains approximately 60 lines of code and the corresponding Milepost IR consists of over 3000 Datalog facts). Secondly, Milepost GCC (which is used to generate the IR) is no longer maintained and is dependent on an old version of GCC (4.4).

In future, the following representations may also be suitable for the task of compiler tuning. Blindell et al. [36] combined the LLVM control-flow and data-flow representations into a universal representation which allows instruction selection to be performed across basic blocks. Gange et al. [30] proposed representing imperative programs as logic programs during compilation and are currently developing an implementation of this representation (LPVM). Finally, Kafle et al. [41] proposed the use of constrained horn clauses to perform resource analysis (such as the energy consumption of a program).

Table 7.1: Average improvement made by ILP

Platform	Benchmarks	ILP	`-Ocm3-ilp/` `-Oca8-ilp`	1NN-prob	COBAYN 1-shot	COBAYN 8-shot
CM3	BEEBS	7%	6%	-13%	3%	11%
CA8	BEEBS	1%	-3%	-22%	-3%	10%
CA8	cBench	-1%	0%	-4%	-	-

CHAPTER 8

Transferability of compiler tuning methods and models across platforms and to a real world application

This chapter explores the transferability of compiler tuning techniques and models. A technique is transferable if it can be applied to a range of systems comprising different hardware, compilers and applications; for example, the ILP compiler tuning technique can be applied to different platforms, but a new model may need to be trained for each of those platforms. A model is transferable if it improves performance on a different system to which it was trained, without the need for retraining.

Three aspects of compiler tuning transferability are explored. Firstly, can iterative and predictive approaches be applied to different platforms (Sec. 8.1)? Secondly, can the knowledge gained for one platform help to improve performance on another platform (Sec. 8.2)? Finally, do the improvements seen on the BEEBS and cBench benchmarks also translate to improvements on a real application independent of the benchmark suites (Sec. 8.3)?

8.1 How well do the compiler techniques perform on different platforms?

The compiler tuning methods explored in this thesis have been applied to both the CM3 bare-metal and the CA8 running Linux with two benchmark suites. It is clear that the methods can be applied on different experimental setups as long as there is some measurable performance metric to be tuned and a compiler with user-configurable flags that can influence that metric.

Overall, CE performed better the RIC on each platform and set of benchmarks. As discussed in Sec. 3.5.1, the converse may be true for other experimental setups. The main weakness of CE is that it only considers changing one flag at time and cases where multiple flags must be disabled at once may be missed. Therefore, in situations where the impact of dependencies is high, RIC may outperform CE. While determining which method to use for a new platform, program and/or compiler could require some initial experimentation, an alternative approach is to alternate between the two methods.

Each machine learning method tested was able to learn models for the CM3 and CA8. On each platform and benchmark suite, the ILP method performed the best overall. It was also possible to construct new optimisation levels on each platform to outperform `-O3`.

When tested on BEEBS, each of the predictive and program-agnostic approaches performed better on the CM3 than the CA8, despite the RIC and CE searches finding the same overall 16% potential speed-up on the two platforms.

8.2 Can knowledge gained on one platform help to optimise another platform?

It is generally accepted that compiler tuning models must be retrained for each new target platform. This section investigates whether knowledge gained within the model or best configurations found for one platform is also beneficial to another platform.

8.2.1 Transferability of best configurations found by iterative compilation

Supposing a set of good configurations has been identified for a set of programs on a target platform and now the user wishes to compile these programs for a new platform (perhaps due to a hardware upgrade). If these configurations also improve performance on the new platform, it will save investing extra effort in a new search on the second platform.

Figures 8.1 and 8.2 demonstrate that some of the best configurations found by iterative compilation on the CM3 are also beneficial on the CA8 and vice versa. The majority of benchmarks benefited from the best RIC configuration found on the other platform, but in each case, around a quarter of the programs performed worse than `-O3`. Therefore, while there is scope for applying the model for one platform directly to the other, additional tuning of the model is likely required to increase accuracy.

8.2.2 Transferability of program-agnostic optimisation levels

Although, the aim of the program-agnostic configurations developed in Ch. 5 is to optimise for a single target platform, it is possible that the configuration constructed for the CM3 (or CA8) could also improve performance on the CA8 (or CM3) by finding settings that are generally good for embedded ARM architectures. This section shows the effect of applying `-Ocm3` to the CA8, and similarly, `-Oca8-beebs` and `-Oca8-cbench` to the CM3.

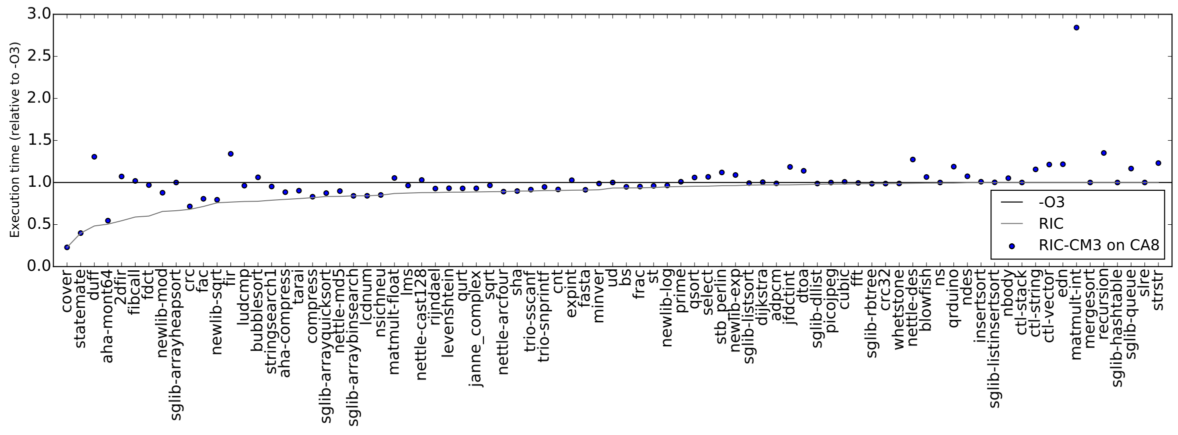


Figure 8.1: Best RIC configuration for CM3 tested on the CA8

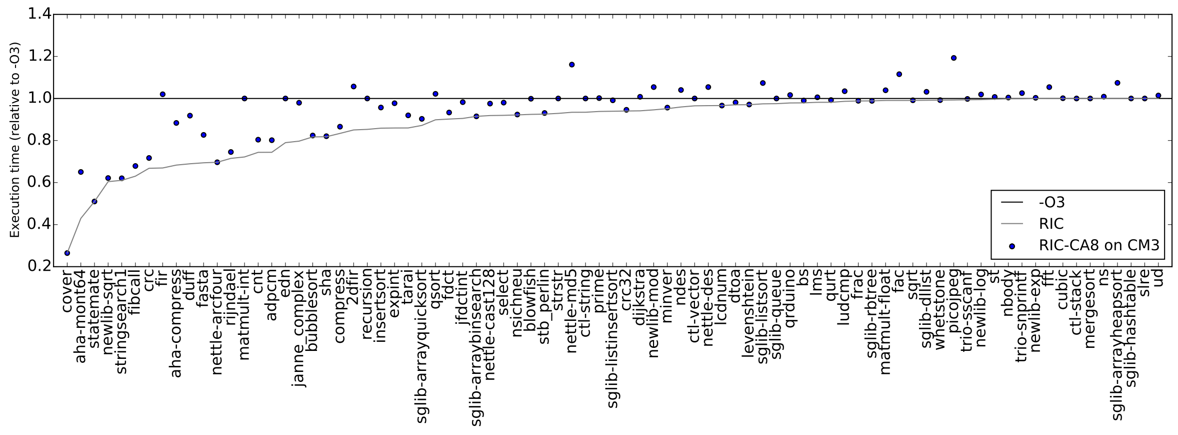


Figure 8.2: Best RIC configuration for CA8 tested on the CM3

On BEEBS, the CM3 and CA8 configurations find significant improvements on several benchmarks (Figs. 8.3 to 8.5). In these cases, the construction of these configurations has made optimisation choices that generalise well across the two platforms. Interestingly, applying the CM3 configurations to the CA8 leads to turbulent results, with some programs performing much better and others performing much worse than `-O3`. A possible explanation is that tuning to the CM3 hardware requires disabling optimisations that are not well suited to this simpler hardware, but which actually have a positive impact when applied to the CA8. In contrast, the CA8 configurations on the CM3 are mostly in line with `-O3` or better.

The CA8 configurations constructed on cBench found very little improvement over `-O3` on the CM3. Conversely, the CM3 configurations did find some speed-ups when used to compile cBench for the CA8.

Note that the original intention of the program-agnostic configuration was to tune the compiler

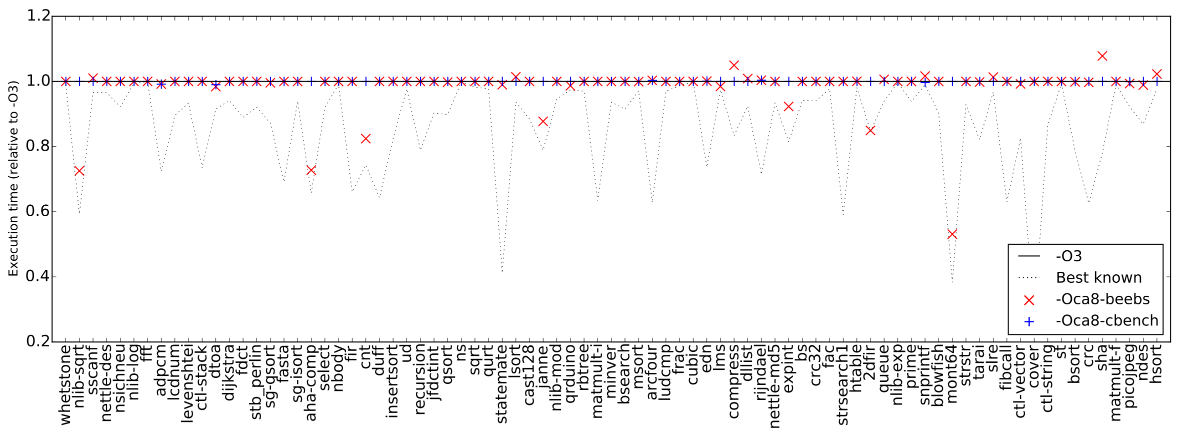


Figure 8.3: -Oca8 on CM3

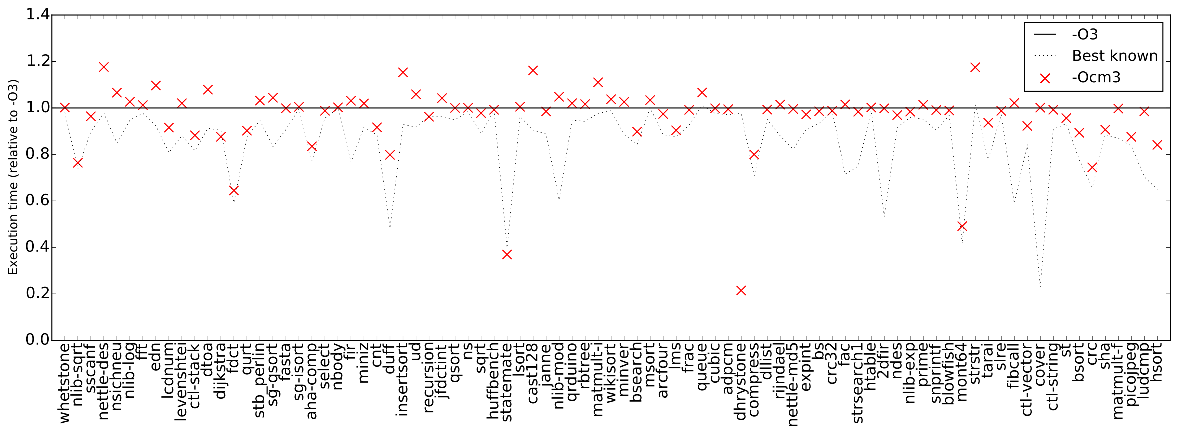


Figure 8.4: -Ocm3 on CA8 BEEBS

to a particular platform. These experiments show that in some cases the configuration can be re-purposed for another platform by taking advantage of settings that are generally good for the two platforms.

8.2.3 Transferability of machine learning models

Traditionally, predictive machine learning approaches learn a new model for each target platform. Supposing a model has been learned for one platform and the user would like to target a second platform that currently has no model, then it would be beneficial to reuse the existing model rather than retrain. This section tests whether the ILP model for the CM3 also gives an improvement on the CA8 and vice versa.

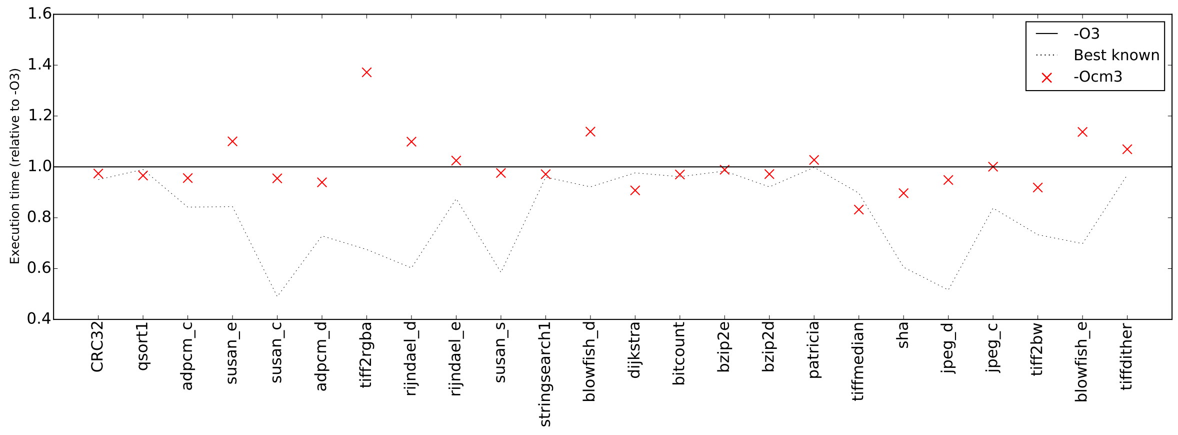


Figure 8.5: `-Ocm3` on CA8 cBench

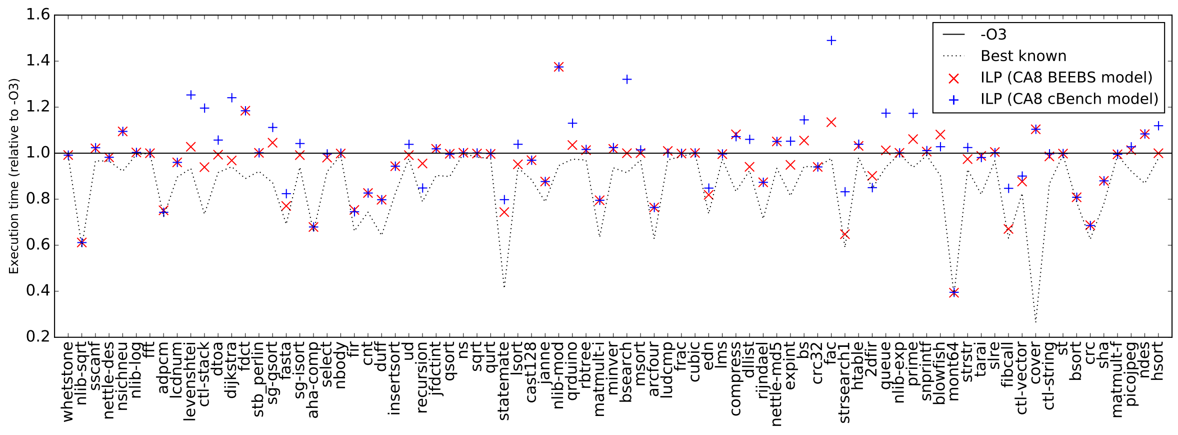


Figure 8.6: CA8 ILP BEEBS and cBench models tested on CM3

8.2.3.1 CA8 model on CM3

In Sec. 7.4, two ILP models were created for the CA8, one trained on BEEBS and one trained on cBench. Applying these models to BEEBS on the CM3 finds near optimal improvements (Fig. 8.6). In particular, the CA8 BEEBS model performed best with a 6% improvement on `-O3`, which is almost as high as the CM3 ILP cross-validation result of 7%. Many of the program-specific decisions made for the CA8 appear to be applicable to both platforms, as also seen in Fig. 8.2.

8.2.3.2 CM3 model on CA8

The CM3 ILP model finds some significant speed-ups on the CA8 for BEEBS and cBench but there are relatively fewer improvements than those found by applying CA8 models to the CM3 (Figs. 8.7 and 8.8). As suggested when discussing the transferability of program-agnostic

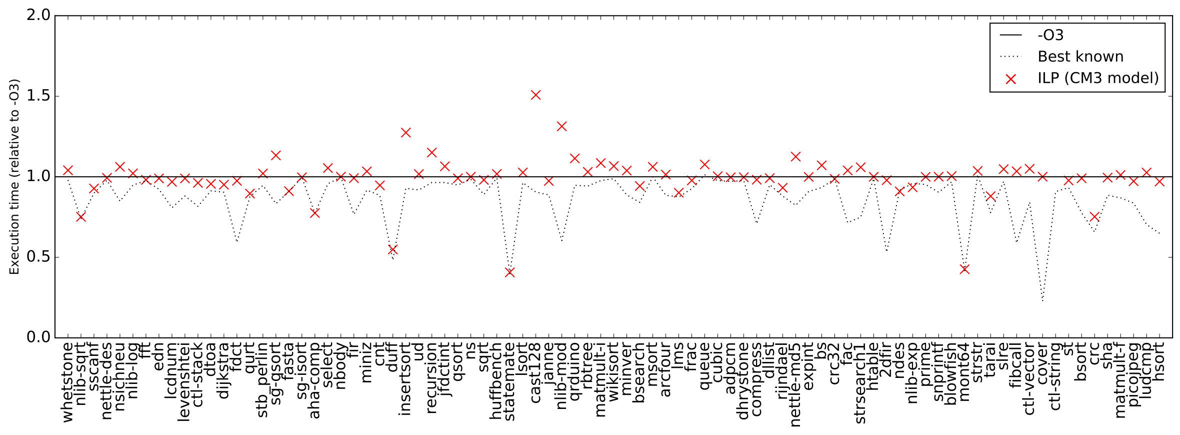


Figure 8.7: CM3 ILP model tested on CA8 (BEEBS)

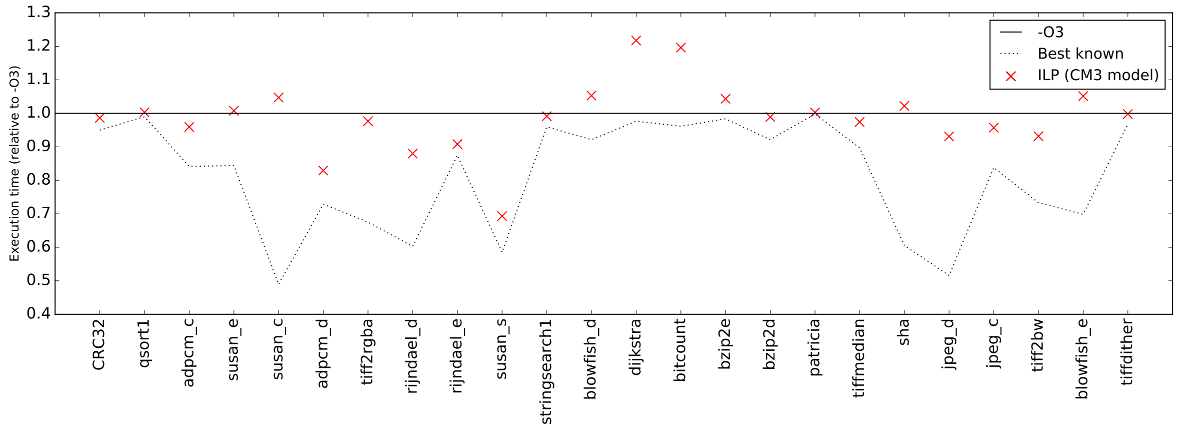


Figure 8.8: CM3 ILP model tested on CA8 (cBench)

configurations (Sec. 8.1), a possible explanation is that the CM3 model disables some flags that are beneficial to the CA8 but degrade performance on the CM3 due to hardware limitations.

8.2.4 Summary

This section demonstrated that knowledge gained for one platform can be used to optimise another platform. In general, the knowledge from the CA8 was more beneficial to the CM3 than vice versa (Tables. 8.1 to 8.3). It is clear, however, that a direct transfer is not always possible. In future, the machine learning model could take into account hardware features as well as software features to produce a multi-platform model.

These experiments targeted two embedded ARM processors, but further work is needed to test the extent to which compiler tuning knowledge is transferable between more distantly related

Table 8.1: Average improvement made by transferring the best RIC configuration for each BEEBS program on the CM3 to the CA8 and vice versa

Platform	Benchmarks	CM3 RIC best config	CA8 RIC best config
CM3	BEEBS	16%	7%
CA8	BEEBS	3%	16%

Table 8.2: Average improvement made by transferring program-agnostic configurations between CM3 and CA8

Platform	Benchmarks	`-Ocm3`	`-Oca8-beebs`	`-Oca8-cbench`
CM3	BEEBS	10%	2%	0%
CA8	BEEBS	6%	2%	0%
CA8	cBench	0%	2%	0%

Table 8.3: Average improvement made by transferring ILP models between CM3 and CA8

Platform	Benchmarks	ILP CM3 BEEBS model	ILP CA8 BEEBS model	ILP CA8 cBench model
CM3	BEEBS	7%*	6%	3%
CA8	BEEBS	2%	1%*	-5%
CA8	cBench	2%	8%	-1%*

*cross-validation result

platforms.

8.3 Do the gains transfer to a real world application?

The aim of this section is to test compiler tuning on an application completely independent to the benchmarks used in evaluations in earlier chapters. This work targets deep learning, a hot topic in IoT Edge computing.

Recently, ARM created the Cortex Microcontroller Software Interface Standard - Neural Network (CMSIS-NN) library to provide optimised kernels that improve the runtime and energy consumption of neural network applications on Cortex-M processors [47]. The library includes NNFunctions that implement commonly used neural network layers (e.g. convolution, fully-connected). Each layer may have multiple implementations to support different input widths and offer optimisations that are only applicable with certain restraints on the inputs. The kernels are constructed with the help of utility functions grouped together as NNSupportFunctions.

A key optimisation made in this library is to use fixed point quantisation, so that the neural networks are implemented with 8-bit or 16-bit fixed point data rather than slower floating point data. Several SIMD optimisations were also made, although these are not applicable to the CM3 targeted here as it has no vector unit.

ARM tested the library by classifying images from the CIFAR-10 dataset [44]. The CIFAR-10 dataset contains 60,000 32x32 colour images divided into ten output classes (of animals and vehicles) with 6,000 images per class (divided into 5,000 training images and 1,000 test images). ARM [47] reported a 4.6x speed-up and 4.9x improvement in energy efficiency on the Cortex-M7 by using the CMSIS-NN library compared to a baseline that uses the CMSIS-DSP (Digital Signal Processing) library along with implementations of functions similar to those available in the Caffe [40] deep learning framework.

As the input data and CNN model require ten times the available RAM of the STM32VLDiscovery, the larger NUCLEO-F207ZG Cortex-M3 development board with 128KB RAM and 1MB flash was used for this case study.

Applying `-Om3` to the already hand-optimised CMSIS-NN library achieved a further 8% speed-up and improvement in energy consumption compared to `-O3` (Table. 8.4). Furthermore, `-Om3` actually outperforms the ILP model (which performed the same as `-O3`) and `-Ocm3-ilp` (which performed 12% worse than `-O3`), thus further demonstrating that a program-agnostic approach is competitive with a more complex machine learning approach.

A reference run compiled with `-O3` takes 5.2s to classify a single image and consumes 4.2mJ energy. Supposing the board is powered by a 2000mAh battery at 3.3V and assuming perfect efficiency, this would allow approximately 5.7 million classifications on a single battery charge. The optimised `-Ocm3` version takes 4.8s and uses 3.9mJ of energy, thus allowing a further 500,000 classifications per battery cycle.

Table 8.4: Summary of improvements on neural network case study program

Platform	Benchmarks	`-Ocm3`	`-Ocm3-ilp`	ILP
CM3	BEEBS	8%	-12%	0%

8.4 Critical evaluation

This thesis aimed to produce a 1-shot predictive approach that exploits IR. The resulting ILP approach improved performance by 7% on CM3 BEEBS and 1% on CA8 BEEBS and outperformed the existing Milepost and COBAYN 1-shot feature vector approaches (Table. 7.1). In addition, the learned models also transferred well between platforms and benchmark suites, in particular, the ILP CA8 BEEBS model found an 8% improvement on CA8 cBench, which exceeds the performance achieved by training directly on cBench and suggests that the model generalises well for the platform (Table. 8.3).

On the case study program, the program-agnostic `-Ocm3` outperformed `-O3` and ILP by 8% (Table. 8.4). This supports the finding in Ch. 5 that program-agnostic optimisation is competitive with predictive program-specific approaches and that a machine learning approach may not always be necessary for targeting previously unseen programs.

On the one hand, the case study program may be an outlier with unique properties that the ILP model could not generalise for based on the BEEBS training set. On the other hand, the program-agnostic `-Ocm3` appears less sensitive to outliers that are not represented in the training set and has found a configuration that is general enough to optimise for potentially very different programs on the platform. One of the goals of `-Ocm3` was to avoid making any single benchmark more than a threshold percentage worse than `-O3` and to find a good balance for all benchmarks.

A further step could be taken to predict whether a model will perform well for the test program. Namely, compare the feature vector or IR for the test program against that of each training program. If no training program appears close enough to the test program, then apply a program-agnostic configuration instead. Determining whether a model is well suited to a given test program could itself be posed as an additional machine learning problem. Automatic program generation could also be used to increase the coverage of the training set so that the model is more likely to be able to optimise the test program.

The ILP and program-agnostic methods are compatible with any iterative compilation approach, but as shown in Ch. 3, the best iterative compilation technique is dependent on the platform, compiler, benchmarks and candidate flags. It is, however, not necessary to choose a single search method. Instead, an ensemble of iterative compilation techniques could be used. For example, one could start with a uniform probability of sampling from each technique, and as the search progresses, update the probabilities to favour the techniques that are currently proving most effective. In this way, the choice of technique can be progressively adapted for each target program. Also, an additional improvement could be made to RIC by increasing the probability with which each flag is enabled; this is based on the observation in Sec. 6.1.1.4 that suggests it is better to assume a flag is good rather than bad.

The results in this thesis are based on real hardware measurements and as such there may be variation in the measurements. This variability was quantified by analysing the RIC data further. The performance of each program and configuration in the RIC data set was measured three times and Table. 8.5 shows the average relative standard deviation (RSD) for each result (as calculated by Eqs. (8.1) and (8.2)).

$$RSD = \frac{1}{\bar{x}}\sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N-1}} \quad \text{where } x_i \text{ is a single measurement} \tag{8.1}$$

$$\text{Average } RSD = \frac{\sum_{b=1}^{m}\sum_{c=1}^{n} RSD_{b,c}}{m \cdot n} \quad \text{where} \quad RSD_{b,c} = RSD \text{ for benchmark } b \text{ with configuration } c \tag{8.2}$$

The measurements on BEEBS were very consistent although the CA8 showed more variation due to the presence of an OS and cache. The higher variability of the cBench benchmarks

Table 8.5: Average relative standard deviation in the RIC dataset

Platform	Benchmarks	Average RSD
CM3	BEEBS	0.01%
CA8	BEEBS	0.38%
CA8	cBench	2.28%

could explain why none of the methods trained on cBench were able to improve upon `-O3`. This could be tackled in future by adaptively repeating each measurement until the variation falls within a required threshold. Moreover, compiler tuning techniques could be trained on consistent benchmarks such as the BEEBS benchmarks and then deployed to improve the performance of more variable programs such as those in cBench. The ability of ILP CA8 BEEBS and `-Oca8-beebs` to improve the performance of CA8 cBench by 8% and 2% respectively demonstrates that this is possible (Table. 8.3). This will be important on more complex processors as the variation is likely to increase.

The techniques were tested on embedded platforms but they are applicable to any platform targeted by a compiler with tunable flags. The flags of interest may differ but some of the tuning decisions may be platform-independent, for example, the rule shown in Sec. 7.2.5 for the CM3 also applied to an Intel i5 processor.

The input data for each benchmark was fixed throughout the experiments. Further tests would be required to determine how well the learned models transfer to different inputs, although choosing realistic input data is non-trivial. In order to train a data-aware model, the IR could be adapted to include data-flow information and also augmented with dynamic features that capture effects of input data during a profile run. Incorporating multiple datasets would, however, increase the time and size complexity of the training task linearly with each new dataset.

While the ILP method achieved significant improvements, the flags were treated as independent and a more accurate model could be learned by considering dependencies between flags. For example, Sec. 3.5.1 showed that on the `cover` benchmark, it was beneficial to disable both `-free-ch` and `-fivopts`, but that disabling just one of these flags increased execution time. If dependencies like these are identified, they could be grouped together for training and prediction.

This thesis focused on deciding whether to enable or disable binary flags, which provides significant improvements over `-O3`. Further gains may be possible by also tuning flag parameters (e.g. loop unroll factor). These parameters could presented as ILP facts, for example, the following facts state that an unroll factor of two or four improves the performance of `program1`: `goodParam(program1, unroll_factor, 2)` and `goodParam(program1, unroll_factor, 4)`. It is possible that multiple unroll factors would be predicted for the same program, in which case, weighting the examples could help to settle ties; this is preferable to averaging the predicted factors as, for example, factors two and four may improve performance but their average may degrade performance.

This thesis aimed to find a general solution to the compiler optimisation selection problem by targeting a diverse benchmark suite. In reality, the most suitable benchmarks are those that reflect the type of application in which the user is interested. The techniques in this thesis could be applied to tune the compiler to particular classes of application rather than for the general case. Just as domain specific processors should be more optimal for a given domain than general purpose processors, the domain specific tuning of a compiler should improve on general purpose compiler tuning.

Chapter 9

Conclusion

This thesis investigated program-specific and program-agnostic solutions to compiler tuning. There are clearly many improvements to be made over the default settings available in the compiler. Accessing these improvements is a matter of time and effort on the part of the software engineer, compiler writer or tuning expert. The software engineer will often not have time to put in this effort and as long as the program runs fast enough, does not consume too much battery or energy budget, they will likely not invest the time in seeking improvement. The software developer begins to care as soon as the performance of the program becomes a barrier preventing productivity, lowering battery or increasing energy budgets. The compiler tuning expert has the power to tune the compiler for the many so that code can be optimised with little effort on the user's part. The initial intensive search pays off by feeding into faster predictive methods that once trained, predict a configuration almost instantaneously.

This thesis showed that tuning the compiler to the platform as a whole captured much of the optimisation potential without the complication of more complex machine learning approaches. Such an observation has not been made before, with most recent research focusing on machine learning approaches with the assumption that they are required.

As with most optimisation problems, there is a trade-off between the time taken to find the solution and the quality of that solution. For a single program that will be deployed widely, the benefits may outweigh the long search time required by iterative compilation. For every day development of large programs, predicting or constructing a configuration that reduces compilation time will speed up the development process. It is down to the user to decide how much time they are willing to let compiler tuning take. One option would be to impose a time limit within the compiler tuning framework so that a k-shot approach could be tried within a restricted period of time.

This work included a thorough evaluation of compiler tuning techniques using the largest and most diverse open source embedded benchmark suite available, BEEBS [58]. Existing state-of-the-art approaches were shown to perform worse on these benchmarks compared to the cBench suite commonly used in compiler tuning research. Creating a realistic and representative benchmark suite is a hard task because of the sheer volume of possible programs that could be written. There needs to be enough diversity to cover different classes of program but enough similarity to allow generalisations to be made. In the case of cBench, some programs were virtually identical which made the compiler tuning task relatively easy. The ILP approach allows relevant structures to be picked out that are important to the selection of optimisations, unlike the feature vector approaches that compare programs or functions as a whole, which in reality, may be too coarse a comparison.

Applications relevant to the platforms in this research include compute intensive tasks carried out by IoT edge devices and the optimisation of mobile devices (e.g. phones, tablets). Other embedded applications are sensitive to timing (for example, they must sleep between reading from a sensor). In such applications there is less room for optimisation from the compiler, firstly, because the device spends a large amount of time idling and secondly, because the compiler could optimise away some of the important timings in the code. In applications that have a mixture of time sensitive and compute intensive functions, compiler tuning could be applied to the compute intensive parts. Additionally, energy can be reduced by using features such as low power mode and dynamic voltage and frequency scaling. Compiler tuning is one piece of the puzzle and the whole system contributes to the energy consumption.

All of the techniques in this thesis are general and can be applied to any platform targeted by a compiler that gives the user control of optimisations on the command line. They can be applied to any target metric that can be measured and which the compiler can influence.

9.1 Future work

The majority of this work applied compiler tuning on each individual platform. Chapter 8 showed that some of the knowledge gained for the CM3 also enabled the CA8 to be optimised and vice versa. This could be because some programs are affected similarly by the same flags regardless of the platform. Also, the two platforms both feature ARM embedded processors and there may be more differences when comparing to another brand of processor e.g. Intel or AMD. In future, the ILP method could be expanded to take into account hardware features such as amount of RAM, number of cores, cache size and pipeline length in order to train on multiple platforms.

Some ILP systems such as XHAIL [64] allow examples to be weighted, therefore, rather than using a binary good/bad classification of flags, the flags could be weighted based on their significance. Applying ILP directly to IR could also be beneficial in other fields such as verification.

The Aleph ILP system supports interactive learning, whereby a domain expert can accept or reject rules with a reason during the training process to improve the machine learner. A related approach might actively test some of the rules during learning and the resulting performance be fed back to decide the quality of the rule.

An additional application for iterative compilation is to identify bugs within the compiler, such as unsound assumptions made by different optimisations. Really, these optimisations should be self contained but in reality, they may make assumptions about the execution and behaviour of previous or future passes.

There was a strong correlation between energy and time on the target platforms and benchmarks, suggesting that most of the energy improvements were due to a reduction in time rather than power and therefore most of the optimisations in GCC optimise for time rather than power (a conclusion also reached in [55]). This is not surprising as energy is much more difficult to measure than time, therefore it is harder for compiler writers to target energy if they cannot quantify the impact of their optimisations. In addition, energy is often a hidden cost; it is obvious if a program takes a long time to run but the energy consumption is often hidden (unless the computer overheats or battery life is poor), so there is often no incentive to focus on energy. Recent projects [25, 2] have aimed to make energy consumption more transparent to increase awareness and enable it to be quantified and optimised.

Options for targeting power directly include developing new compiler optimisations [56], applying dynamic voltage and frequency scaling, and deciding how to schedule work across multiple nodes and cores. Interestingly, in High Performance Computing (HPC), the goal is not typically to save energy, but to achieve maximum throughput within a given power budget. Researchers have noted that supercomputers are often underutilised, with average workloads only using around 60% of the available power [46]. Consequently, the concept of over-provisioning was introduced in which more nodes are procured than could actually run at full power, and the performance improvements are made by efficiently scheduling the work across a larger number of nodes running at lower power.

A natural progression of the work in this thesis is to target the phase-ordering problem. The optimisation sequences could be encoded relationally in Prolog and the task of ILP would be to infer rules that relate effective sequences or sub-sequences of passes to code structure. Extracting a new IR after each pass has been applied could be key to producing learning based approaches that take into account the effects of optimisations already applied in order to avoid negative interactions between flags (both in the optimisation selection and phase-ordering problems).

Finally, [31] showed how performance can be improved significantly by applying fewer of the passes defined in an existing standard optimisation level while preserving their original ordering. The task of finding the optimal point at which to stop applying the sequence of optimisations in Clang/LLVM's `-O2` (or any another optimisation level) poses an interesting problem for future research on machine learning based compiler tuning.

Bibliography

[1] Collective benchmark.
2012.
http://ctuning.org/wiki/index.php?title=CTools:CBench.

[2] MAchine Guided Energy Efficient Compilation Project (MAGEEC).
http://mageec.org, 2015.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams.
Using machine learning to focus iterative optimization.
In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[4] B. Alpern et al.
The Jikes Research Virtual Machine project: Building an open-source research community.
IBM Systems J., 44(2):399–417, 2005.

[5] ARM.
Cortex-M3 technical reference manual (Rev: r2p1), 2015.

[6] ARM.
Procedure call standard for the ARM architecture, 2015.

[7] ARM.
Arm mbed boards.
http://developer.mbed.org/platforms, 2017.

[8] A. H. Ashouri.
Compiler autotuning using machine learning techniques.
PhD thesis, Politecnico di Milano, 2016.

[9] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos.
Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning.
ACM Trans. Archit. Code Optim., 14(3):29:1–29:28, Sept. 2017.

[10] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano.
Cobayn: Compiler autotuning framework using bayesian networks.
ACM Transactions on Architecture and Code Optimization, 13(2):21:1–21:25, June 2016.

[11] C. Blackmore.
Declarative machine learning for energy efficient compiler optimisations.
Master's thesis, University of Bristol, 2014.
Supervised by Oliver Ray and Kerstin Eder.

[12] C. Blackmore, K. Eder, and O. Ray.
Declarative machine learning for energy efficient compiler optimizations.
Presented at the International Conference on Inductive Logic Programming, 2014.

[13] C. Blackmore, O. Ray, and K. Eder.
A logic programming approach to predict effective compiler settings for embedded software.
Theory and Practice of Logic Programming, 15(4-5):481–494, 2015.

[14] C. Blackmore, O. Ray, and K. Eder.
Automatically discovering human-readable rules to predict effective compiler settings for embedded software.
Automated Reasoning Workshop, 2017.

[15] C. Blackmore, O. Ray, and K. Eder.
Automatically tuning the gcc compiler to optimize the performance of applications running on embedded systems.
arXiv:1703.08228 [cs.DC], 2017.

[16] C. Blackmore, O. Ray, M. Kull, M. G. Rahman, P. Flach, and N. Lachiche.
Reframing of Classification and Regression Tasks for Predicting the Effects of Compiler Settings on Multiple Embedded Systems.
In *2nd International Workshop on Learning over Multiple Contexts*, 2015.

[17] D. Brooks, V. Tiwari, and M. Martonosi.
Wattch: A framework for architectural-level power analysis and optimizations.
In *Proceedings of the International Symposium on Computer Architecture*, pages 83–94. ACM, 2000.

[18] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam.
Rapidly selecting good compiler optimizations using performance counters.
In *Proc. of the Int. Symp. on Code Generation and Optimization*, pages 185–197, 2007.

[19] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu.
Deconstructing iterative optimization.

ACM Transactions on Architecture and Code Optimization, 9(3):21, 2012.

[20] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer.
Protecting bare-metal embedded systems with privilege overlays.
In *IEEE Symp. on Security and Privacy*, 2017.

[21] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg.
Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment.
In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, pages 381–386. IEEE, 2015.

[22] K. D. Cooper, P. J. Schielke, and D. Subramanian.
Optimizing for reduced code space using genetic algorithms.
ACM SIGPLAN Notices, 34(7):1–9, May 1999.

[23] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather.
End-to-end deep learning of optimization heuristics.
In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232, 2017.

[24] G. B. Dantzig.
Linear programming and extensions.
Princeton University Press, 1963.

[25] K. Eder, J. P. Gallagher, P. López-García, H. Muller, Z. Banković, K. Georgiou, R. Haemmerlé, M. V. Hermenegildo, B. Kafle, S. Kerrison, M. Kirkeby, M. Klemen, X. Li, U. Liqat, J. Morse, M. Rhiger, and M. Rosendahl.
Entra: Whole-systems energy transparency.
Microprocessors and Microsystems, 47:278 – 286, 2016.

[26] P. Finn, S. Muggleton, D. Page, and A. Srinivasan.
Pharmacophore discovery using the inductive logic programming system progol.
Machine Learning, 30(2):241–270, Feb 1998.

[27] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. K. I. Williams, and M. O'Boyle.
Milepost GCC: Machine learning enabled self-tuning compiler.
Int. J. of Parallel Programming, 39(3):296–327, 2011.

[28] G. Fursin, A. W. Memon, C. Guillon, and A. Lokhmotov.

Collective mind, part II: towards performance- and cost-aware software engineering as a natural science.
CoRR, abs/1506.06256, 2015.

[29] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, et al.
Milepost GCC: machine learning based research compiler.
In *GCC Summit*, 2008.

[30] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey.
Horn clauses as an intermediate representation for program analysis and transformation.
Theory and Practice of Logic Programming, 15(4-5):526–542, 2015.

[31] K. Georgiou, C. Blackmore, S. Xavier-de Souza, and K. Eder.
Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption.
In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, pages 35–42, New York, NY, USA, 2018. ACM.

[32] S. V. Gheorghita, H. Corporaal, and T. Basten.
Using iterative compilation to reduce energy consumption.
Proceedings of the 10th Annual Conference of the Advanced School for Computing and Imaging, 2004.

[33] R. F. Gunst and R. L. Mason.
Fractional factorial design.
Wiley Interdisciplinary Reviews: Computational Statistics, 1(2):234–244, 2009.

[34] J. Gustafsson et al.
The Mälardalen WCET benchmarks – past, present and future.
In B. Lisper, editor, *WCET'2010*, pages 137–147, July 2010.

[35] M. Guthaus et al.
Mibench: A free, commercially representative embedded benchmark suite.
In *Proc. of 4th IEEE Int. Workshop on Workload Characterization*, pages 3–14, 2001.

[36] G. Hjort Blindell, R. Castañeda Lozano, M. Carlsson, and C. Schulte.
Modeling universal instruction selection.
In G. Pesant, editor, *Principles and Practice of Constraint Programming*, volume 9255 of *Lecture Notes in Computer Science*, pages 609–626. Springer International Publishing, 2015.

[37] K. Hoste and L. Eeckhout.
Microarchitecture-independent workload characterization.

IEEE Micro, 27(3):63–72, May 2007.

[38] K. Hoste and L. Eeckhout.
Cole: compiler optimization level exploration.
In *Proc. of the Int. Symp. on Code Generation and Optimization*, pages 165–174. ACM, 2008.

[39] K. Hoste and L. Eeckhout.
Cole: compiler optimization level exploration.
In *Proc. of the Int. Symp. on Code Generation and Optimization*, pages 165–174. ACM, 2008.

[40] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell.
Caffe: Convolutional architecture for fast feature embedding.
arXiv preprint arXiv:1408.5093, 2014.

[41] B. Kafle, J. P. Gallagher, and J. A. Navas.
Resource analysis of a C program by analysing its horn clause representation.
[Slides] Presented at 2nd ICT-ENERGY International Doctoral Symposium, 2015. `http://akira.ruc.dk/~kafle/presentations/ICT-ENERGY-Workshop15.pdf`.

[42] R. Karp.
Reducibility among combinatorial problems.
In R. Miller, J. Thatcher, and J. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[43] T. Kisuki, P. Knijnenburg, M. O'Boyle, F. Bodin, and H. Wijshoff.
A feasibility study in iterative compilation.
In C. Polychronopoulos et al., editors, *High Performance Computing*, volume 1615 of *LCNS*, pages 121–132. 1999.

[44] A. Krizhevsky.
Learning multiple layers of features from tiny images.
2009.

[45] S. Kulkarni and J. Cavazos.
Mitigating the compiler optimization phase-ordering problem using machine learning.
ACM SIGPLAN Notices, 47(10):147–162, Oct. 2012.

[46] S. Labasan, M. Larsen, H. Childs, and B. Rountree.
PaViz: A power-adaptive framework for optimal power and performance of scientific visualization algorithms.
In *Eurographics Symposium on Parallel Graphics and Visualization*, 2017.

[47] L. Lai, N. Suda, and V. Chandra.
CMSIS-NN: efficient neural network kernels for arm cortex-m cpus.
CoRR, abs/1801.06601, 2018.

[48] H. Leather, E. Bonilla, and M. O'boyle.
Automatic feature generation for machine learning–based optimising compilation.
ACM Transactions on Architecture and Code Optimization, 11(1):14:1–14:32, Feb. 2014.

[49] F. Li, F. Tang, and Y. Shen.
Feature mining for machine learning based compilation optimization.
In *Intl. Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 207–214, July 2014.

[50] A. Makhorin.
GNU linear programming kit.
`https://www.gnu.org/software/glpk/`, 2012.

[51] P. Mpeis, P. Petoumenos, and H. Leather.
Iterative compilation on mobile devices.
arXiv:1511.02603 [cs.PL] (presented at Intl. Workshop on Adaptive Self-tuning Computing Systems), 2016.

[52] S. Muggleton.
Inverse entailment and progol.
New Generation Computing, 13(3-4):245–286, 1995.

[53] S. Muggleton and L. D. Raedt.
Inductive logic programming: Theory and methods.
The J. of Logic Programming, 19:629–679, 1994.

[54] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund.
Practical aggregation of semantical program properties for machine learning based optimization.
In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 197–206. ACM, 2010.

[55] J. Pallister.
Exploring the fundamental differences between compiler optimisations for energy and for performance.
PhD thesis, University of Bristol, 2016.

[56] J. Pallister, K. Eder, and S. Hollis.
Optimizing the flash-ram energy trade-off in deeply embedded systems.

In *Proc. of the Int. Symp. on Code Generation and Optimization*, pages 115–124. IEEE Computer Society, 2015.

[57] J. Pallister, S. Hollis, and J. Bennett.
Identifying compiler options to minimize energy consumption for embedded platforms.
The Computer Journal, 2013.

[58] J. Pallister, S. J. Hollis, and J. Bennett.
BEEBS: Open benchmarks for energy measurements on embedded platforms.
arXiv:1308.5174v2 [cs.PF], 2013.

[59] Z. Pan and R. Eigenmann.
Fast and effective orchestration of compiler optimizations for automatic performance tuning.
In *Proc. of the Int. Symp. on Code Generation and Optimization*, 2006.

[60] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan.
Predictive modeling in a polyhedral optimization space.
International Journal of Parallel Programming, 41(5):704–750, Oct 2013.

[61] R. Pinkers, P. Knijnenburg, M. Haneda, and H. Wijshoff.
Statistical selection of compiler options.
In *Proc of the Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 2004.

[62] L. Pouchet.
Polybench: The polyhedral benchmark suite.
2012.
`http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`.

[63] S. Purini and L. Jain.
Finding good optimization sequences covering program space.
ACM Transactions on Architecture and Code Optimization, 9(4):56:1–56:23, Jan 2013.

[64] O. Ray.
Nonmonotonic abductive inductive learning.
Journal of Applied Logic, 7(3):329–340, 2009.
Special Issue: Abduction and Induction in Artificial Intelligence.

[65] G. Sher, K. Martin, and D. Dechev.
Preliminary results for neuroevolutionary optimization phase order generation for static compilation.
In *Proc. of the 11th Workshop on Optimizations for DSP and Embedded Systems*, pages 33–40, 2014.

[66] A. Srinivasan.
The aleph manual.
`http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph`, 2007.

[67] K. Stanley and R. Miikkulainen.
Evolving neural networks through augmenting topologies.
Evolutionary computation, 10(2):99–127, 2002.

[68] M. Tartara and S. Crespi Reghizzi.
Continuous learning of compiler heuristics.
ACM Transactions on Architecture and Code Optimization, 9(4):46:1–46:25, Jan. 2013.

[69] The GCC team.
GCC, the gnu compiler collection.
`http://gcc.gnu.org/`, 2018.

[70] The LLVM Team.
http://clang.llvm.org, 2017.

[71] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August.
Compiler optimization-space exploration.
In *Int. Symp. on Code Generation and Optimization*, 2003.

[72] Xilinx.
Xilinx software development kit (SDK) user guide: System performance analysis (v2017.2), 2017.

[73] V. Zivojnovic et al.
Dspstone: A dsp-oriented benchmarking methodology.
In *Proc. of ICSPAT*, volume 94, 1994.