POLITECNICO DI MILANO
DEPARTMENT OF COMPUTER SCINCE AND ENGINEERING (DEIB)
DOCTORAL PROGRAMME IN 2016

COMPILER AUTOTUNING USING MACHINE LEARNING TECHNIQUES

Doctoral Dissertation of:
Amir Hossein Ashouri

Advisor:
Prof. Cristina Silvano

Co-advisors:
Prof. Gianluca Palermo
Prof. John Cavazos

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

December 2016 - 28th PhD Cycle

Acknowledgments

ACKNOWLEDGMENTS. This PhD thesis has been carried out at the Department of Electronics and Computer (DEIB) at Politecnico di Milano University within the period of four years from January 2013 to December 2016. The work has been performed at System Architecture Group (SAG) under advisement of professors Cristina Silvano and Gianluca Palermo, who provided limitless support during the course of my PhD. Moreover, for a period of 7 months from September 2014 to March 2015, I was a visiting scholar at computer department of university of Delaware, DE, USA and worked under supervision of prof. John Cavazos. I am truly thankful for all his supports and the collaboration since then.

I would like to thank all my colleagues including postdoctoral and PhD fellows both at Milan, Italy and Newark, DE, USA with whom I had the opportunity to work and co-author a publication, specifically: Dr. Giovanni Mariani, Dr. Sotiris Xydis, Prof. Marco Alvarez, Dr. Eunjung Park, Dr. Sameer Kulkarni, William Kilian and Robert Searles. The team work was truly fun and challenging at the same time. I learned a lot and met many top-nudge researchers who shared their knowledge and experience which I am very grateful and would like to thank.

Last but not least, I would like to appreciate lifetime support of my lovely family; Mother, Father and younger brother whom always been my backbone during the hard-times and the good-times. Thank you for giving me the positive energy to carry on and thanks for urging me to choose this path for my life.

P.S: My PhD thesis is partially supported by the European Commission call H2020-FET-HPC program under the grant ANTAREX-671623 and the grants FP7-HARPA-612069 and FP7-CONTREX-611146. Moreover, I would like to thank HiPEAC (EU ICT-287759) for the short-term collaboration grant no.6811 provided in 2014-15.

Amir H. Ashouri
Milan, December 2016

Abstract

ABSTRACT. Recent developments in silicon production and fabrication led to the creation of much faster computational units such as CPUs, GPUs, FPGAs, and similar chips with varying instruction set architectures (ISAs). Software (SW) programming paradigms including OpenMP, MPI, OpenCL, and OpenACC allow software developers to exploit Hardware (HW) parallelism to port legacy serial codes on these emerging platforms to attain application speedups. Compilers struggle to keep up with the increasing development pace of ever-expanding hardware and software programming paradigms. Additionally, growing complexity of the modern compilers and the concern over security are among the more serious problems that compilers should answer. Moore's law states that transistor density should double every two years; however, the rate of compilers, which are faced with many open-research problems, have not been able to improve more than a few percentage points each year.

Diversity of today's architectures have forced programmers to spend additional effort to port and tune their application code across different platforms. Compilers within this process need additional tuning which is a hard task itself. Recent compilers offer a vast number of multilayered optimizations, capable of targeting different code segments of an application. Choosing among these optimizations can significantly impact the performance of the code being optimized. The selection of the right set of compiler optimizations for a particular code segment is a very hard problem, but finding the best ordering of these optimizations adds further complexity. In fact, finding the best ordering is a long standing problem in compilation research called the phase-ordering problem. The traditional approach of constructing compiler heuristics to solve this problem simply can not cope with the enormous complexity of choosing the right ordering of optimizations for every code segment in an application.

In this PhD thesis, we provide break-through approaches to tackle and mitigate the well-known problems of compiler optimization using design space exploration and machine learning techniques. We show that not all the optimization passes are beneficial to be used within an optimization sequence and in fact many of the available passes are obliterating the effect of one another when ordering of the phases are taken into account. Experimental results show major improvement in performance metrics when

our customized prediction models are in place versus standard fixed optimization passes predefined within state-of-the-art compiler frameworks e.g. GCC, LLVM, etc. We perform application specific optimization based on the characteristics of applications under analysis and we show that this methodology is beneficial to mitigate the hard problem of selecting the best compiler optimizations and the phase-ordering problem.

Late but not least, we hope that the proposed approaches in this PhD thesis will be useful for a wide range of readers, including computer architects, compiler developers, researchers and technical professionals.

Preface

PREFACE. My academic story began from my home-town Tehran, Iran where I obtained my B. Sc. in Information Technology (IT) Engineering on September 2009 at "Iran University of Science and Technology" (IUST). The last-three years G.P.A (Major GPA) was 3.08/4. My B. Sc thesis entitled "Media Server Evaluations and Real-Time Tests" was done at Research center of Information Technology (RCIT) under advisory of prof. Ahmad Akbari , which was accepted with the score of 20/20.

On September 2010, I moved to Italy for start my M. Sc. at Politecnico Di Milano where I got the overall GPA of 3.73/4 (109/110 Italian scale) and worked on my thesis about "Analysis of Compiler Transformations in VLIW architecture" under advisory of Vittorio Zaccaria and Cristina Silvano. As of January 2013, I have been a PhD student and researcher of DEIB, specifically in the Computer Architecture Group of Cristina Silvano and Gianluca Palermo.

During my PhD candidacy, I mainly investigated the exploration of machine learning techniques on compiler auto-tuning problems. The excellent results of the research carried out during the past years have been published with the title name of COBAYN (Compiler auto-tuning using machine learning) on the top journal of my field, the ACM Transactions Architecture and Code Optimization (TACO). The obtained result were the outcome of a collaboration I had with John Cavazos's high performance and compiler group at University of Delaware, USA. I was able to establish the collaboration with the aforementioned group upon winning a EU-funded grant from HiPEAC (European Network on High Performance and Embedded Architecture and Compilation) organization. HiPEAC is the leading organization to steer the European research in the area of high performance and embedded computing. Having the chance to collaborate with a top-level research group helped me a lot to gain adjacent expertise in auto-tuning field and I could start building more ideas towards mitigating the current open problems such as the phase ordering of the compiler optimization [1].

[1] My collaboration report in published at: (http://home.deib.polimi.it/ashouri/hipeacinfo44.pdf#page=11

As of now, I still am continuing my collaboration with the aforementioned group and we have submitted another TACO journal, namely MiCOMP (Mitigating the Compiler Phase-ordering), plus a survey journal to the prestigious journal of ACM computing survey. We hope that these works will be useful for a wide range of readers, including computer architects, compiler developers, researchers and technical professionals.

Last but not least, I believe having the PhD in a top EU university helped me a lot to integrate more with the scientific and industrial community and built strong motivation towards having a research career in academia.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

INTRODUCTION. Usually, software applications are developed in a high-level programming language (such as C, C++, or Fortran) and then passed through the compiler to emit an executable binary. Compilers have been used for the past 50 years [9, 93] for generating machine-dependent executable binary from high-level programming languages. Compiler developers typically design optimization passes in order to transform each code segment of a program to produce an optimized version of an application. The optimizations can be applied at different stages of the compilation process since compilers have three main layers: (i) *front-end* (ii) *intermediate-representation* (IR) and (iii) *backend*. At the same time, optimizing source code by hand is a tedious task. Compiler optimizations provide an automatic methods to transform code. To this end, optimizing the intermediate phase plays an important role on the performance metrics. Enabling compiler optimization parameters (e.g. loop unrolling, register allocation, etc.) could substantially benefit several performance metrics. Depending on the objectives, these metrics could be execution time, code size, or power consumption. A holistic exploration approach to trade-off these metrics also represents a challenging problem [173].

Recent developments in silicon production and fabrication led to the creation of much faster computational units such as CPUs, GPUs, FPGAs, and similar chips with varying instruction set architectures (ISAs). Software (SW) programming paradigms including OpenMP, MPI, OpenCL, and OpenACC allow software developers to exploit Hardware (HW) parallelism to port legacy serial codes on these emerging platforms to attain application speedups. Compilers struggle to keep up with the increasing development pace of ever-expanding hardware and software programming paradigms. Additionally, growing complexity of the modern compilers and the concern over security are among the more serious problems that compilers should answer. Moore's

law [202] states that transistor density should double every two years; however, the rate of compilers, which are faced with many open-research problems, have not been able to improve more than a few percentage points each year [93].

Usually, software applications are developed in a high-level programming language (e.g. C, C++) and then passed through the compiler to emit an executable binary. Compilers have been used for the past 50 years [9, 93] for generating machine-dependent executable binary from high-level programming languages. Compiler developers typically design optimization passes in order to transform each code segment of a program to produce an optimized version of an application. The optimizations can be applied at different stages of the compilation process since compilers have three main layers: (i) *front-end* (ii) *intermediate-representation* (IR) and (iii) *backend*. At the same time, optimizing source code by hand is a tedious task. Compiler optimizations provide an automatic methods to transform code. To this end, optimizing the intermediate phase plays an important role on the performance metrics. Enabling compiler optimization parameters (e.g. loop unrolling, register allocation, etc.) might substantially benefit several performance metrics. Depending on the objectives, these metrics could be execution time, code size, or power consumption. A holistic exploration approach to trade-off these metrics also represents a challenging problem [173].

Autotuning addresses automatic code-generation and optimization by using different scenarios and architectures. It constructs techniques for automatic optimization of different parameters in order to maximize or minimize the satisfaction of an objective function. Historically, optimizations were mostly done in the backend where *scheduling*, *resource-allocation* and *code-generation* are done [42, 76]. The constraints and the resources form a linear system (ILP) that must be solved. Recently, researchers have shown increased effort in introducing front-end and IR-optimizations. This claim is supported by two facts: i) the complexity of a back-end compiler requires exclusive knowledge strictly by the compiler designers and ii) lower overheads with external compiler modification compared with back-end modifications. This latter process normally involves fine tuning compiler optimization parameters by a multi-objective optimization formulation. Nonetheless, each approach has its benefits and drawbacks and are subject to analysis under their own scope.

1.1 Problem Overview

The major challenge in choosing the right set of compiler optimization is the fact that these code optimizations are programming language, application, and architecture dependent. Additionally, the word optimization is a misnomer; there is no guarantee the transformed code will perform better than the original version. In fact, aggressive optimizations can degrade the performance of the code of which they are applied [227]. Understanding the behavior of the optimizations, the perceived effects on the source-code, and the interaction of the optimizations with each other are complex modeling problems. This understanding is particularly difficult because compiler developers must consider hundreds of different optimizations that can be applied during the different compilation phases. This optimization ordering dilemma creates the phase-ordering problem.

1.1.1 Research Challenges

Although there were several long-standing problems with optimizations, they have not been adequately addressed because optimizations were yielding performance improvements. These problems included knowing *what* optimizations to apply, and in which configuration (e.g., the tile size in loop tiling) and, in *which order* to apply them for the best improvement. The former yields the so-called problem of selecting the best compiler optimizations and the latter is the phase-ordering problem of compiler optimizations.

In optimization theory [217], a feasible set, search space, or solution space is the set of all possible points (sets of values of the choice variables) of an optimization problem that satisfy the problem's constraints, potentially including inequalities, equalities, and integer constraints. This is the initial set of candidate solutions to the problem before the set of candidates has been narrowed down. Compiler optimization problem polarizes over two major sub-problems based on (i) whether we take into account the enabling/disabling the optimizations only (optimization selection problem) or (ii) changing the ordering of those optimizations (phase-ordering problem). Here we briefly discuss the different optimization space of the two.

Challenge 1. Design Space Exploration of Compiler Parameters Embedded systems design traditionally exploits the knowledge of the target domain, e.g. telecommunication, multimedia, home automation etc., to customize the HW/SW coefficients found onto the deployed computing devices. Although the functionalities of these devices are differed, the computational structure and design are tightly connected with the platform in which they rely on. Platform-based design has been proposed as a promising alternative for designing complex systems by redefining the problem of designing into that of finely tuning specific parameters of the platform template.

Although a significant amount of research has been conducted on exploring and optimizing VLIW architectural parameters [16] and introducing specific compiler optimization for VLIW processors [69], [104], there are limited references regarding the analysis of the impacts of conventional compiler transformations onto VLIW architectures and moreover how these transformations are correlating with the underlying architectural configuration.

Challenge 2. The Selection Problem Several compiler optimizations form an optimization sequence. When we disregard the ordering of these optimizations and focus on whether or not to apply the optimization, we define the scope of selecting the best compiler optimizations. Previous researchers have shown that the interdependencies and interaction between enabling/disabling optimizations in a sequence can dramatically alter performance of a running code even by ignoring the order of phases [7, 20].

This process of selecting the right optimizations for each code segment is typically done manually and the sequence of optimizations is constructed with little insight into the interaction between the preceding compiler optimizations in the sequence. The task of constructing heuristics to select the right sequence of compiler optimizations is infeasible given the ever growing number of compiler optimizations being integrated into compiler frameworks. As an example, GCC has more than 200 compiler passes,

referred to as compiler *options* [1], and LLVM-clang and LLVM-opt both have more than 100 transformation passes [2] each. Additionally, these optimizations are applied at very different phases of the compilation, including analysis passes and loop-nest passes. Most optimization flags are turned off by default and compiler developers rely on software developers to know, which optimizations will be beneficial for their code. Compiler developers provide standard optimization levels, e.g. `-O1`, `-O2`, `-Os`, etc. to introduce a fixed-sequence of compiler optimizations that, on average, bring good performance on a set of benchmarks compiler developers tested the optimization levels with. However, using predefined optimizations usually is not good enough to bring the best achievable application-specific performance. One of the key approaches that are used recently in literature in order to find the best optimizations to apply given an application is inducing prediction models using different classes of machine learning [13]. Approaches which leverage machine learning to find the best optimizations to apply will be the center focus of this chapter.

Challenge 3. The Phase-ordering Problem The *phase-ordering* problem has been an open-problem in the field of compiler research for many decades [128, 130, 230, 233]. The inability of researchers to solve the phase-ordering problem has led to advances in the simpler problem of selecting the right set of optimizations, but even this problem has yet to be solved [20, 28, 46].

Compiler designers must consider the order in which optimization phases are performed; a pair of phases may be interdependent in the sense that each phase could benefit from information produced by the other. When having both the selection and the ordering are of importance, the phase-ordering problem is formed. It is once of the longstanding problems of compilation field and has its peer problems in numerous other sub-fields of compiler design such as register allocation, code-generating and compaction [138, 230].

1.1.2 Dissertation Contribution

Addressing Challenge. 1 In order to address the exploration of design space parameters in embedded domain, we propose a methodology that provides the designer with an integrated framework to automatically (i) generate optimized application-specific VLIW architectural configurations and (ii) analyze compiler level transformations, enabling application-specific compiler tuning over customized VLIW system architectures. We based the aforementioned analysis on a Design of Experiments (DoEs) procedure that captures in a statistical manner the higher order effects among different sets of activated compiler transformations. Applying the proposed methodology onto real-case embedded application scenarios, we show that (i) only a limited set of compiler transformations exposes high confidence level (over 95%) in affecting the performance and (ii) using them we could be able to achieve gains between (16-23)% in comparison to the default optimization levels. Chapter 3 discusses more in details the aforementioned contribution.

[1] https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
[2] http://llvm.org/docs/Passes.html

Addressing Challenge. 2 In order to address the problem of selecting the right set of compiler optimizations, we proposes *COBAYN*: COmpiler autotuning framework using BAYesian Networks, an approach for a compiler autotuning methodology using machine learning to speed up application performance and to reduce the cost of the compiler optimization phases. The proposed framework is based on the application characterization done dynamically by using independent micro-architecture features and Bayesian networks. The chapter also presents an evaluation based on using static analysis and hybrid feature collection approaches. In addition, the chapter compares Bayesian networks with respect to several state-of-the-art machine-learning models. Chapter 4 discusses more in details the aforementioned contribution.

Addressing Challenge. 3 Mitigating the phase-ordering problem, we propose two different approaches: (i) an intermediate speedup predictor and (ii) a complete sequence predictor approaches. We elaborate on the pros and cons of each approaches and provide extensive experimental comparison against standard optimization levels and state-of-the-art techniques. Chapters 5 and 6 discusses more in details the aforementioned contribution.

1.1.3 Dissertation Outline

In this PhD thesis, I have tackled the major problems of compiler autotuning. I have used machine learning, DSE, and meta-heuristic techniques to construct efficient and accurate models to induce prediction models. This PhD thesis is organized as following:

Chapter 2. First we provide an extensive review on the literature including the proposed approaches and the results in Chapter 2. We elaborated more than hundred papers for the past twenty five years on the topic, as of the first application of machine learning has been introduced for compiler optimization.

Chapter 3. Following the literature review, in Chapter 3 we provide an co-exploration approach using design space exploration technique on an embedded domain, namely VLIW. We show that this technique can bring speedup by using certain optimizations passes over our proposed VLIW micro-architecture.

Chapter 4. In this chapter we present Cobayn, a novel machine learning approach on selecting the most promising compiler optimizations using Bayesian networks. This technique significantly improves application's performance against using fixed optimization available at GCC where our Bayesian network select the most promising compiler passes.

Chapters 5. In this chapter we introduce an intermediate speedup prediction approach to tackle the problem of phase-ordering. We show that using our intermediate predictor, we can outperform LLVM's default setting by up to 5% while exploring only a fraction of the space..

Chapter 6. In this chapter, we present MiCOMP, our novel machine learning predictive models on how to tackle the phase-ordering problem. It showcases a complete sequence speedup predictor on the very problem. We introduce a clustering technique and a simple mapper so enable us to use traditional prediction techniques on the optimization's space.

Chapter 7. Finally in this chapter we conclude with the conclusion and the bibliography.

A Note on The Experimental Setup

Due to the fact that this dissertation is carried out on a period of few years using different platforms, compilers, methodologies, standard bench-suites and datasets and different performance monitoring tools, it is noteworthy to mention that the experimental setup is varied across different chapters. E.g. the results are averaged from 9-25 times including/excluding a second of sleep, and/or flushing the memory in between the executions and different generations. Nevertheless, the main goal of the experimental setup was to make sure the results are 100% accurate and are generated fair among the whole process. Certain limitations, i.e. time constraints, memory and space constraints might be the reason for this variance which can be a normal factor in research. I would like to stress the fact that we made sure on each chapter the experimental results are all within the same generation policy and are fair.

CHAPTER 2

Background

2.1 Summary

Since the mid 1980s, researchers have been trying to use machine-learning based approaches to solve a number of different compiler optimization problems. The techniques primarily enhance the quality of the obtained results and, more importantly, make it feasible to tackle two main compiler optimization problems: optimization selection (choosing which optimizations to apply) and phase-ordering (choosing the order of applying optimizations). The compiler optimization space continues to grow due to the advancement of applications, increasing compiler optimizations, and new target architectures. Generic optimization passes in compilers cannot fully leverage newly introduced optimizations and, therefore, cannot keep up with the pace of increasing options. This chapter summarizes and classifies the recent advances in using machine learning for the compiler optimization field, particularly on the two major problems of (i) selecting the best optimizations and (ii) the phase-ordering of optimizations. It highlights the approaches taken, the obtained results, holistic comparisons among different approaches and finally, the visionary path towards the near future.

2.2 Introduction

Contribution. In this chapter, we provide an extensive survey of techniques and approaches done by the authors tackling the aforementioned problems. We elaborate more than 100 recent papers proposed for compiler autotuning when Machine Learning (ML) was involved. To the best of our knowledge, the first application of machine learning for the compiler auto-tuning problem was done by [52, 124, 160]. However, there were other original works

which tackled the problem of compiler autotuning without the use of machine learning technique [147, 171, 186, 230, 235] and we believe them to be the driving force of using machine learning on the existing problems. Thus we decided to consider the past 25 years as it covers the whole time span of the literature on the very field.

Additionally, this chapter can be a connecting-point for the already available surveys [24, 205] on the compiler optimization field.

We first discuss the motivation for and challenges involved in the compiler optimization research in Section 2.3, followed by analysis on the optimization space for the two major optimization problems. Then, we review the existing characterization techniques and the classification of those in Section 2.4. Furthermore, we discuss the machine learning models used in Section 2.5 and provide a full classification of different prediction techniques used in the recent research in Section 2.6. In Section 2.7, we discuss the different Design Space Exploration (DSE) techniques on how the optimization configurations are generated for traversing the optimization space. In Section 2.8, we discuss the different target architectures and compiler frameworks involved in tuning process.

That includes a brief review on the Polyhedral compilation framework in Section 2.8. We discuss comparisons on certain selected papers in Section 2.9. The last section includes a complementary discussion on the influential papers of the field classifying by the number of citations, ground-breaking novelty and influence they had on the succeeding work. and finally we conclude the chapter with discussion and future trends.

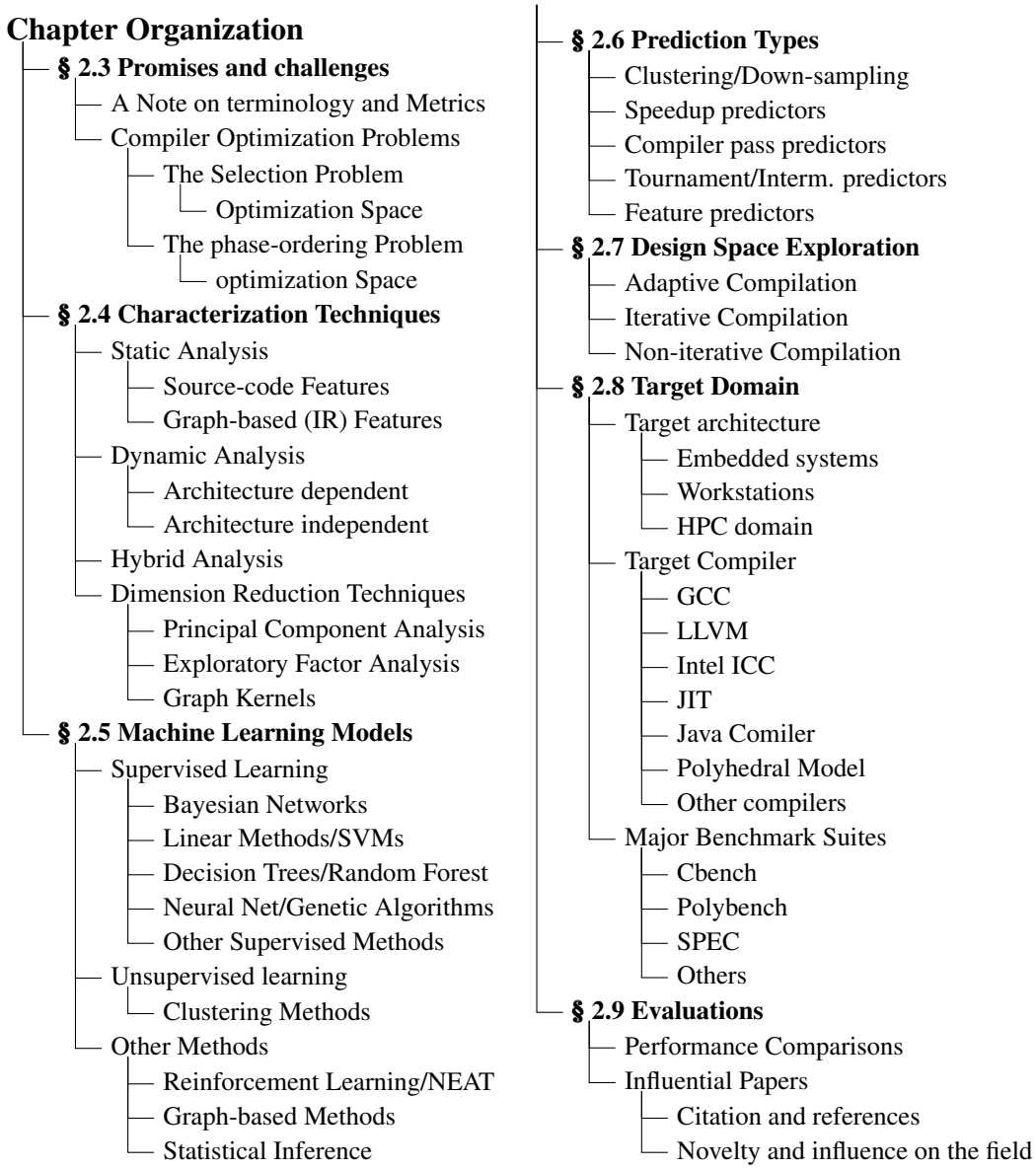


Figure 2.1: *Organization of the survey in different sections*

We hope that this chapter will be useful for a wide range of readers, including computer architects, compiler developers, researchers and technical professionals.

Scope of the chapter. In these sections, we organize the works in different categories to underscore their similarities and differences. Note that the works presented in these sections are deeply intertwined and while we study a work under a single group, several of these works belong to multiple groups. We organize the survey in a way that all research papers corresponding to a specific type of classification are cited. However, we selectively picked the more notable works under each section and we provide more elaboration on their contribution. As an example, Cavazos et. al. [41] proposed to use performance counters to characterize an application. The vector of features then can

be used to construct prediction models. Since this work was using this novel way of characterization, it has been elaborate more on the Section 2.4.2, however, due to our classification policy this work has been cited in different classification tables whenever it was placed under the right category.

2.3 Compiler Optimizations

2.3.1 A Note on terminology and Metrics

Since publications mentioned in this survey originated from varying authors, terminologies were locally defined and may not be strictly defined. We clarify terms used in this survey here and relate them to the publications discussed. Compiler optimization field has been referred to as compiler autotuning, compilation, code optimization, collective optimization, and program optimization. To maintain clarity, we do not use all these terms but rather use optimizing compilers, or compiler autotuning. Moreover, under each classified subsection, we will point out the other nomenclatures that has been used widely and our reference subsequently.

2.3.2 Compiler Optimization Benefits and Challenges

The majority of potential speedup no longer arrives at the increase of processor core clock frequencies. Automatic methods of exploiting parallelism and reducing dependencies are needed. Compiler optimizations allow a user to affect the generated code without changing the original high-level source code. Such optimizations, when applied, may run better on a target architecture. The user is not able to manual tune a large code, so automatic methods need introduced. Additionally, manual tuning is not portable – transformations applied to code running on one architecture is not guaranteed to yield the same performance increase on another architecture. Advanced compilers' goals of increased instruction-level parallelism and data-parallelism are hard or impossible to reach. Optimal instruction packing/scheduling for Very Long Instruction Word (VLIW) architectures [70] is NP-hard, and approximation algorithms are still very complex.

Research One clear benefit of optimization passes is their portability – if necessary, they can be easily adapted to newer architectures. However, there are some optimizations that have been researched more than others. Specifically, we still have not reached the once "holy grail" of auto-parallelizing compilers, but we have made significant progress. Polyhedral loop analysis and transformations paved the way for safe transformations leading to auto-parallelizable code segments. The polyhedral model also aided with the generation of architecture-dependent, cache-friendly access patterns.

Users and compiler writers Compiler writers expose general-purpose transformations to end users. Ultimately, a subset of these transformations lead to better architecture fitting given source code. Over time, we've seen overall improvement of compilers and related tools. New high-level languages and languages extensions make additional optimizations possible. Directive-based programming languages, such as OpenMP and OpenACC , automatically transform users' code to exploit parallelism. One of the most

Table 2.1: *A Classification Based on the type of the Problem*

Classification	References
The Selection Problem	[4, 7, 11, 18, 36, 37, 39, 41, 47, 48, 52, 60, 61, 63, 74, 78, 81–85, 87, 95, 100, 118, 122, 139, 146, 149, 150, 163, 174, 175, 177, 181, 185, 187–189, 195, 198, 204, 213–215, 218, 224, 226, 229, 232, 239, 245]
The Phase-ordering Problem	[7, 19, 126, 128, 130, 152, 166, 178, 191, 192, 227, 230, 233, 235]

advantageous contributions has been the introduction of easily expandable compiler infrastructures such as LLVM. An advanced compiler infrastructure makes it possible to introduce new optimizations with minimal effort.

2.3.3 Compiler Optimization Problems

The problem of interdependency among phases of compiler optimizations is not unique to compiler optimization field. Phase inter-dependencies have been noted in traditional optimizing compilers between constant folding and flow analysis, and between register allocation and code generation [138, 230].

In optimization theory [217], a feasible set, search space, or solution space is the set of all possible points (sets of values of the choice variables) of an optimization problem that satisfy the problem's constraints, potentially including inequalities, equalities, and integer constraints. This is the initial set of candidate solutions to the problem before the set of candidates has been narrowed down. Compiler optimization problem polarizes over two major sub-problems based on (i) whether we take into account the enabling/disabling the optimizations only (optimization selection problem) or (ii) changing the ordering of those optimizations (phase-ordering problem). Here we briefly discuss the different optimization space of the two.

Table 2.1 classifies the existing literature based on the type of the problem. As we mentioned earlier, the inability of researchers to solve the phase-ordering problem has led to advances in the more simple problem of selecting the right set of optimizations and that is the reason recent work are tacking the former more.

The Problem of Selecting the Best Compiler Optimizations

Several compiler optimizations form an optimization sequence. When we disregard the ordering of these optimizations and focus on whether or not to apply the optimization, we define the scope of selecting the best compiler optimizations. Previous researchers have shown that the interdependencies and interaction between enabling/disabling optimizations in a sequence can dramatically alter performance of a running code even by ignoring the order of phases [7, 20].

Optimizations Space Let us define a Boolean vector $\boldsymbol{o}$ whose elements o_i are the different compiler optimizations. Each optimization o_i can be either enabled $o_i = 1$ or disabled $o_i = 0$. A compiler optimization sequence to be *selected* is represented by the vector $\boldsymbol{o}$ belongs to the n dimensional Boolean space of:

$$|\Omega Selection| = \{0, 1\}^n \tag{2.1}$$

For the application a_i being optimized and n represents the number of compiler optimizations under study. Therefore, the mentioned research problem consists of an exponential space as its upper-bound. Having n = 10, drive us to a total space (2^n) up to $|\mathcal{O}selection|$ = 1024 options to select among per interested target application a_i to be optimized and this number itself would be multiplied by different applications A = $a_0...a_N$ under study.

Extended version of the current definition in Equation 2.1 is the case where we have more than a binary choice (enabling/disabling). Certain compiler optimizations offer multiple levels of optimization to choose among, i.e. `-loop-unrolling`, `loop-tiling`, etc. in many compiler frameworks with different values such as $4, 8, 16, m$ etc.. Consequently, we have have the previous equation as:

$$|\Omega_{Selection\_Extended}| = \{0, 1, ..., m\}^n \tag{2.2}$$

where m defines the number of different optimization levels a compiler optimization have.

The Phase-ordering Problem

Compiler designers must consider the order in which optimization phases are performed; a pair of phases may be interdependent in the sense that each phase could benefit from information produced by the other. When having both the selection and the ordering are of importance, the phase-ordering problem is formed. It is once of the longstanding problems of compilation field and has its peer problems in numerous other sub-fields of compiler design such as register allocation, code-generating and compaction [138, 230].

Optimizations Space A phase-ordering optimization sequence represented by the vector $\boldsymbol{o}$ belongs to the n dimensional factorial space of:

$$|\Omega_{Phases}| = n! \tag{2.3}$$

where n represents the number of compiler optimizations under study. However, the mentioned bound is for a simplified phase-ordering problem having a fixed length optimization sequence length and no repetitive application of optimizations. Allowing optimizations to be repeatedly applied and a variable length sequence of optimizations will expand the problem space to:

$$|\Omega_{Phases\_Repetition\_variableLength}| = \sum_{i=0}^{m} n^i \tag{2.4}$$

Where n is the number of optimizations under study and m is the maximum desired length for the optimization sequence. Even for reasonable values for n and m, the entire search space is enormous. For example, assuming n and m are both equal to 10, this leads to an optimization search space of more than 11 billion different optimization sequences to select from for each piece of code being optimized [19] [1].

[1] The problem of phase-ordering does not have deterministic upper-bound in the case of an unbounded length.

Table 2.2: *A Classification Based on Application Characterization Techniques*

Classification		References
Static	Source-code Features	[7, 20, 21, 36, 39, 40, 60, 75, 81–84, 87, 93, 130, 131, 134, 139, 144, 149–151, 159, 163, 174, 177, 180, 181, 192, 195, 213, 216, 226, 241, 242]
	Graph-based (IR) Features	[36, 37, 51, 60, 75, 84, 124, 134, 139, 149, 150, 163, 164, 174, 180, 181, 214, 216, 218, 226, 233]
Dynamic	Architecture Dependent	[41, 47, 58, 61, 78, 80, 84, 118, 150, 174, 178–181, 195, 214, 216, 226, 229, 232]
	Architecture Independent	[17, 19–21, 47, 60, 178]
Hybrid features		[20, 21, 58, 134, 139, 150, 174, 180, 181, 195, 226]

2.4 Application characterization techniques

For computer architects and compiler designers, understanding the characteristics of applications running on current and future computer systems is of utmost importance during design. Applications of machine learning on the compiler optimization fields needs a quantitative model of application, kernels, or nested loops to be processed by an algorithm to induce a prediction mode. Thus, a tool is needed that scans an application and collect its features. To obtain a more accurate model, compiler researchers have been trying to understand the behavior of programs/kernels better and build a vector of features that best represents pair functionality. In general, (i) the derived feature vector must be representative enough of its program/kernel, and (ii) different programs/kernels must not have the same feature vector. Thus, construction of a large, inefficient feature vector slows down, or even halts, the ML process ultimately reducing the precision.

In this survey, we present different program characterization techniques used in referenced literature: (i) Static Analysis of the features, (ii) Dynamic Feature collection, and (iii) Hybrid feature collection, which uses a combination of the previous two or other extraction methods. Table 2.2 refers to such classification.

2.4.1 Static Analysis

Static Analysis, or static features collection, tries to collect features that are non-functional to a code being run. Static analysis involves parsing source code at the front-end, intermediate representation (IR), the backend, or any combination of the three. Collecting static features doesn't require the code to be executed and is considered to be one of the strongest support cases for its use. We briefly classify source code features leveraged in prior research.

Source Code Features

Source code (SRC) features are abstractions of some selected properties of an input application or the current compiler intermediate state when other optimizations have already been applied. They range from simple information such as the name of the current function to the values of compiler parameters to the pass ordering in the current run of the compiler. There are numerous source-code feature extractors used in the literature. Fursin et. al. proposed Milepost GCC [81, 82], as a plugin to GCC compiler

Table 2.3: *Available SRC Features in Milepost GCC [82] Tool*

FT	Description	FT	Description
ft1	Number of basic blocks in the method	ft34	Number of switch instructions in the method
ft2	Number of basic blocks with a single successor	ft35	Number of unary operations in the method
ft3	Number of basic blocks with two successors	ft36	Number of instruction that do pointer arithmetic in the method
ft4	Number of basic blocks with more then two successors	ft37	Number of indirect references via pointers ("*" in C)
ft5	Number of basic blocks with a single predecessor	ft38	Number of times the address of a variables is taken ("&" in C)
ft6	Number of basic blocks with two predecessors	ft39	Number of times the address of a function is taken ("&" in C)
ft7	Number of basic blocks with more then two predecessors	ft40	Number of indirect calls (i.e. done via pointers) in the method
ft8	Number of basic blocks with a single predecessor and a single successor	ft41	Number of assignment instructions with the left operand an integer constant in the method
ft9	Number of basic blocks with a single predecessor and two successors	ft42	Number of binary operations with one of the operands an integer constant in the method
ft10	Number of basic blocks with a two predecessors and one successor	ft43	Number of calls with pointers as arguments
ft11	Number of basic blocks with two successors and two predecessors	ft44	Number of calls with the number of arguments is greater then 4
ft12	Number of basic blocks with more then two successors and more then two predecessors	ft45	Number of calls that return a pointer Number of calls that return an integer
ft13	Number of basic blocks with number of instructions less then 15	ft46	Number of occurrences of integer constant zero
ft14	Number of basic blocks with number of instructions in the interval [15, 500]	ft47	Number of occurrences of 32-bit integer constants
ft15	Number of basic blocks with number of instructions greater then 500	ft48	Number of occurrences of integer constant one
ft16	Number of edges in the control flow graph	ft49	Number of occurrences of 64-bit integer constants
ft17	Number of critical edges in the control flow graph	ft50	Number of references of a local variables in the method
ft18	Number of abnormal edges in the control flow graph	ft51	Number of references (def/use) of static/extern variables in the method
ft19	Number of direct calls in the method	ft52	Number of local variables referred in the method
ft20	Number of conditional branches in the method	ft53	Number of static/extern variables referred in the method
ft21	Number of assignment instructions in the method	ft54	Number of local variables that are pointers in the method
ft22	Number of unconditional branches in the method	ft55	Number of static/extern variables that are pointers in the method
ft23	Number of binary integer operations in the method	ft57	Number of unconditional branches in the method
ft24	Number of binary floating point operations in the method	ft57	Cyclomatic complexity
ft25	Number of instructions in the method	ft58	HALSTEAD's metrics
ft26	Average of number of instructions in basic blocks	ft59	Hn2 is number of distinct operands (Halstead n2)
ft27	Average of number of phi-nodes at the beginning of a basic block	ft60	N is num var defs (should be == Halstead n2 or Halstead N2?)
ft28	Average of arguments for a phi-node	ft61	HN1 is total number of operators (Halstead N1) (approx due to abstraction)
ft29	Number of basic blocks with no phi nodes	ft62	Hn1 is number of distinct operators (Halstead n1) (approx due to abstraction)
ft30	Number of basic blocks with phi nodes in the interval [0, 3]	ft63	Approximation of Halstead difficulty
ft31	Number of basic blocks with more then 3 phi nodes	ft64	Approximation of Halstead volume
ft32	Number of basic block where total number of arguments for all phi-nodes is in greater then 5	ft65	Approximation of Halstead effort
ft33	Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5]		

to extract source-level features [2]. Table 2.3 shows the different features the tool can extract from a source-code.

Graph-based Features

Graph-based representation makes data and control dependencies explicit for each operation in a program. Data dependence graphs have provided some optimizing compilers with an explicit representation of the definition-use relationships implicitly present in a source program [170]. A control flow graph (CFG) [10] has often been the representation for the control flow relationships of an application. The control conditions on which an operation depends can be derived from such a graph. The program dependence graph explicitly represents both the essential data relationships, as present in the

[2] http://ctuning.org/wiki/index.php/CTools:MilepostGCC:StaticFeatures:MILEPOST_V2.1

data dependence graph, and the essential control relationships, without the unnecessary sequencing present in the control flow graph [68]. There are numerous tools to extract a control flow graph of a kernel, a function, or an application. MinIR [157], LLVM's Opt [3], IDA pro [62] are such examples of these tools available.

Koseki et. al. [124] used the CFG and dependency graph to understand the unrolling factor. According to the formula for determining the efficiency of loop execution P, P increases monotonically until it saturates. Therefore, this algorithm never chooses directions that lead to local maximum points. Moreover, they showed, it can find the point for which unrolling is performed the fewest times, as it chooses the nearest saturated point to determine the number of times and the directions in which loop unrolling is performed.

Park et. al. [177] introduced a novel way of characterizing programs using MinIR, a graph-based characterization, which uses the program's intermediate representation and an adapted learning algorithm to predict good optimization sequences. The authors constructed feature vectors of an application using graph kernels, namely, shortest path graph kernels [32]. To evaluate different characterization techniques, they focus on loop-intensive programs and construct prediction models that drive polyhedral optimizations, such as auto-parallelism and various loop transformations.

Nobrea et. al. [164] proposed an iterative compilation approach using graph-based features of the code being optimized to mitigate selecting and ordering of compiler optimization passes in LLVM. The authors proposed heuristics to reduce the search space and converge faster.

2.4.2 Dynamic Characterization

Dynamic characterization involves extracting the performance counters (PC) that are used to provide information as to how well an application is performing. The counter data can help determine system bottlenecks and fine-tune system and application performance. Applications can also use counter data to determine how much system resources to consume. For example, an application that is data cache intensive can be tuned by exploiting more cache locality. Here we briefly describe and classify the different types of collecting performance counters. Refer to Table 2.2 for full classification on different application characterization techniques.

Architecture Dependent Characterization

Modern processors are often equipped with a special set of registers that allow for measuring performance counter events with no disruption to the running program. These events can describe several characteristics of the running program, such as, cache hits and misses and branch prediction statistics. For instance, on the AMD Athlon, there are 4 registers for measuring performance counter events, but up to 60 different events can be measured. It is possible to collect anywhere between 4 and 60 types of events per run by multiplexing the use of the special registers [41]. However, this way of collecting the application characterization is solely for the specific type of the platform the data has been collected on. In other word, a platform dependent characterization is not portable to other platforms.

[3] http://llvm.org/docs/Passes.html#id12

Moreover, there are tools publicly available to collect such metrics, i.e. PAPI tool [161] is able to specify a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. These counters exist as a small set of registers that count events, which are occurrences of specific signals related to the processor's function. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. PFMon [108] is another notable performance monitoring tool for Intel Itanium platform.

Cavazos et. al. [41] proposed an offline machine learning based model which can be used to predict the good set of compiler optimization sequences. It uses performance counters as a means of determining good compiler optimization setting. The authors showed that using this they could outperform existing techniques used by static code features, as dynamic characterization takes into account the memory and cache behaviours at execution time.

Architecture Independent Characterization

The information obtained from dynamic characterization is a compact summary of a program's dynamic behavior. In particular, they summarize important aspects of a program's performance, e.g., cache misses or floating point unit utilization. However, the information can be inaccurate or misleading if the application is run on different target architectures because of variances found such as cache size, execution ports, and scheduling algorithms. Architecture dependent counters, while accurate, are short of being used in cross-platform manner. Because of this limitation, researchers proposed a different way of collecting dynamic behaviours which can be ported to other platform if

they have the same instruction set architecture (ISA), i.e. X86_64. This way of collecting features are known as instrumentation and are done using the dynamic program analysis tools.

Intel Pin [148] is a noteworthy framework that enables such collection. Pin is a dynamic binary instrumentation tool. Instrumentation is performed at run time on the compiled binary files. A major advantage of Pin is that it requires no recompiling of source code and can support instrumenting programs that dynamically generate code. The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications in Linux and Windows. Micro Architecture Independent Characterization of Applications (MICA) [102] is an example of a Pintool which is capable of collecting a number of program characteristics to quantify runtime program behavior [4]. These program characteristics are totally independent of the micro-architecture, i.e. cache configuration, branch predictor, etc., on which the measurements are done, in contrast to other workload characterization techniques using simulation or hardware performance counters. Table 2.4 demonstrates the type of characterization and features MICA can collect by instrumenting the executable binary at runtime.

Ashouri et. al. [17] proposed a Bayesian network approach to address the problem of selecting the best compiler optimizations suitable for a embedded processor to gain speedup versus the fixed standard optimizations available at different levels of GCC

[4] http://kejo.be/ELIS/mica/

Table 2.4: *Available Dynamic Architecture Independent Features in MICA [102] (A Pintool Plugin)*

Feature Category	Features
Instruction Mix	totInstruction, total_ins_count_for_hpc_alignment, totInstruction, mem-read, mem-write, control-flow, arithmetic, floating-point, stack,shift, string, sse, other, nop
ILP	ILP32, ILP64, ILP128, ILP256
Register Traffic	memReuseDist0-2, memReuseDist2-4, memReuseDist4-8, memReuseDist8-16, memReuseDist16-32, memReuseDist32-64, memReuseDist64-128, memReuseDist128-256, memReuseDist256-512, memReuseDist512-1k, memReuseDist1k-2k, memReuseDist2k-4k, memReuseDist4k-8k, memReuseDist8k-16k, memReuseDist16k-32k, memReuseDist32k-64k, memReuseDist64k-128k, memReuseDist128k-256k, memReuseDist256k-512k, memReuseDist512k-00
Memory Foot-print	InstrFootprint64, InstrFootprint4k, DataFootprint64, DataFootprint4k, mem_access
Data Stream Strides	mem_read_local_stride_0, mem_read_local_stride_8, mem_read_local_stride_64, mem_read_local_stride_512, mem_read_local_stride_4096, mem_read_local_stride_32768, mem_read_local_stride_262144, mem_read_global_stride_0, mem_read_global_stride_8, mem_read_global_stride_64, mem_read_global_stride_512, mem_read_global_stride_4096, mem_read_global_stride_32768, mem_read_global_stride_262144, mem_write_cnt, mem_write_local_stride_0, mem_write_local_stride_8, mem_write_local_stride_64, mem_write_local_stride_512, mem_write_local_stride_4096, mem_write_local_stride_32768, mem_write_local_stride_262144, mem_write_global_stride_0, mem_write_global_stride_8, mem_write_global_stride_64, mem_write_global_stride_512, mem_write_global_stride_4096, mem_write_global_stride_32768, mem_write_global_stride_262144
Branch Predictability (PPM)	total_num_ops, instr_reg_cnt, total_reg_use_cnt, total_reg_age,reg_age_cnt_1, reg_age_cnt_2, reg_age_cnt_4, reg_age_cnt_8, reg_age_cnt_16, reg_age_cnt_32, reg_age_cnt_64, mem_read_cnt, GAg_mispred_cnt_4bits, PAg_mispred_cnt_4bits, GAs_mispred_cnt_4bits, PAs_mispred_cnt_4bits, GAg_mispred_cnt_8bits, PAg_mispred_cnt_8bits, GAs_mispred_cnt_8bits, PAs_mispred_cnt_8bits, GAg_mispred_cnt_12bits, PAg_mispred_cnt_12bits, GAs_mispred_cnt_12bits, PAs_mispred_cnt_12bits, total_brCount, total_transactionCount, total_takenCount, total_num_ops

compiler. The authors used MICA features of applications under analysis to induce a prediction model and to infer from the model.

2.4.3 Hybrid Characterization

The hybrid characterization technique consists of the combination of the previously known technique in a way that adds more information on the application under analysis. In some cases [20, 181], hybrid feature selection can capture the application behaviors better as it takes into account different levels of feature extraction.

Park et. al. [181] used a technique to characterize a program using a pattern-driven system named HERCULES [117]. This characterization technique not only helps a user to understand programs by searching pattern-of-interests, but also can be used for a predictive model that effectively selects the proper compiler optimizations. The authors formulated 35 loop patterns, and evaluated the characterization technique by comparing the predictive models constructed using HERCULES to three other state-of-the-art characterization methods and achieved up to 67% of the best possible speedup achievable with the optimization search space they evaluated.

2.4.4 Dimension Reduction Techniques

In the studied literature, dimension-reduction process is important for three main reasons: (a) it eliminates the noise that might perturb further analyses, (b) it significantly reduces the training time of a model, and (c) it becomes easier to visualize the data when reduced to very low dimensions such as 2D or 3D. The techniques used are mainly Principal Component Analysis (PCA) [111], Exploratory Factor Analysis (EFA) [90] and Graph-based kernels, i.e. Shortest Path Kernels [32].

Experimental results in many recent work show that the selection of a good dimension-reduction technique can have a significant impact on the final model quality [7, 20]. For instance, in the original work proposed in Ashouri et al. [17], PCA was used. On an

Table 2.5: *A Classification Based on Dimension Reduction Techniques*

Classification	References
Principal Component Analysis	[7, 17, 19–22, 47, 60, 122, 177, 178, 180, 224]
Factor Analysis	[20]
Graph Kernels	[124, 164, 177]

extended-version of the work [20], the authors changed the model by exploiting EFA and observed benefits of using EFA with respect to PCA for the specific problem addressed. We elaborate more on these methods in this chapter. Table 2.5 classifies the use of each technique in the studied literature.

Let $\boldsymbol{\gamma}$ be a characterization vector storing all data of an application run. This vector stores l variables to account for either the static, dynamic or a hybrid analyses. Let us consider a set of known application profiles A consisting of m vectors $\boldsymbol{\gamma}$. The application profiles can be organized in a matrix P with m rows and l columns. Each vector $\boldsymbol{\gamma}$ (i.e. a row in P) includes a large set of characteristics, such as the instruction count per instruction type (for both static and dynamic analysis), information on the memory access pattern and information characterizing the control flow (e.g. the number and length of the basic blocks, average and maximum loop nesting, etc.). Many of these application characteristics (columns of matrix P) are correlated to each other in a complex way. A simple example of this correlation is the instruction mix information collected during the static analysis and the instruction mix information collected during the dynamic profiling (even though these are not completely the same). A less intuitive example is between the distribution of basic block lengths and data related to the instruction memory reuse distance. The presence of many correlated columns in P implies that the information stored in a vector $\boldsymbol{\gamma}$ can be well represented with a vector $\boldsymbol{\alpha}$ of smaller size.

Both PCA and FA are statistical techniques aimed at identifying a way to represent $\boldsymbol{\gamma}$ with a shorter vector $\boldsymbol{\alpha}$ while minimizing the information loss. Nevertheless, they rely on different concepts for organizing this reduction [90, 223]. In both cases, output values are derived by applying the dimension reduction and are no longer directly representing a certain feature. While in PCA the components are given by a combination of the observed features, in EFA the factors are representing the hidden process behind the feature generation. In both cases, there is no way to indicate by name the output columns, since they are not directly observable.

Principal Component Analysis

In PCA, the goal is to identify a summary of $\boldsymbol{\gamma}$. To this end, a second vector $\boldsymbol{\rho}$ of the same length of $\boldsymbol{\gamma}$ (i.e. l) is organized by a variable change. Specifically, the elements of $\boldsymbol{\rho}$ are obtained through a linear combination of the elements in $\boldsymbol{\gamma}$. The way to combine the elements of $\boldsymbol{\gamma}$ for obtaining $\boldsymbol{\rho}$ is decided upon the analysis of the matrix P, and is such that all elements in $\boldsymbol{\rho}$ are orthogonal (i.e. uncorrelated) and are sorted by their variance. Thus the first elements of $\boldsymbol{\rho}$ (also named principal components) carry most of the information of $\boldsymbol{\gamma}$. The reduction can be obtained by generating a vector $\boldsymbol{\alpha}$ to keep only the first most significant principal components in $\boldsymbol{\rho}$, because the least significant

ones carry little information content. Note that principal components in $\boldsymbol{\rho}$ (thus in $\boldsymbol{\alpha}$) are not meant to have a meaning; they are only used to summarize the vector $\boldsymbol{\gamma}$ as a signature.

Factor Analysis

In FA, this relationship explains the correlation between the different variables in $\boldsymbol{\gamma}$; that is, correlated variables in $\boldsymbol{\gamma}$ are likely to depend on the same hidden variable in $\boldsymbol{\alpha}$. The relationship between the latent $\boldsymbol{\alpha}$ and the observed variables is regressed by exploiting the maximum likely method based on the data in matrix P. When adopting PCA, each variable in $\boldsymbol{\alpha}$ tends to be a mixture of all variables in $\boldsymbol{\gamma}$. Therefore, it is rather hard to tell what a component represents. When adopting EFA instead, the components $\boldsymbol{\alpha}$ tend to depend on a smaller set of elements in $\boldsymbol{\gamma}$ that are correlated with each others. That is, when applying EFA, $\boldsymbol{\alpha}$ is a compressed representation of $\boldsymbol{\gamma}$, where elements in $\boldsymbol{\gamma}$ that are correlated (i.e. that carry the same information) are compressed into a reduced number of elements in $\boldsymbol{\alpha}$. Note that reducing the profile size by means of FA results in a $\boldsymbol{\alpha}$ that better describes the type of application under analysis in reference to PCA [110].

Graph Kernels

These techniques construct a low-dimensional data representation using a cost function that retains local properties of the data, and can be viewed as defining a graph-based kernel for Kernel PCA. The use of kernel functions is very attractive because the input data does not always need to be expressed as feature vectors [124, 164, 177]

Park et. al. [177] proposed the use of graph kernels to characterize an application. They tried to avoid simply flattening the information into feature vectors, because this would removed important information about the structure of the graphs. Such information is useful because it allows the learning algorithm to effectively capture the similarities between two different program Thus, the authors are used discrete structures data as inputs, e.g., control flow graphs with Shortest Path kernels [32] to construct the vectors and induce a prediction model.

2.5 Machine Learning Models

Machine learning explores the study and construction of algorithms that can learn from and make predictions on data [158]. Many types and sub-fields of machine learning exist and here we classify them based on the broad categories: (i) Supervised learning (ii) Unsupervised learning, and (iii) Other Methods (including reinforcement learning, graph-based technique and statical methods). The classification of all machine learning models used is depicted in Table 2.6. In each subsection we provide an overview of the method and we mention the major tools and works involved. The general formulation of the optimization problem is to construct a function that takes as input the features of the unoptimized program being compiled. In other words, this model takes as an input a tuple (F, T) where F is the feature vector of the collected instrumentation of the program being optimized; and T is one of the several possible compiler sequences predicted to perform well on this program. Its output is a prediction of the speedup T should achieve when applied to the original code.

Table 2.6: *A Classification Based on Machine Learning Models*

Classification		References
Supervised Learning	Bayesian Networks	[17, 20]
	Linear Models / SVMs	[19, 20, 178, 179, 195, 214]
	Decision Trees / Random Forests	[58, 75, 79, 79, 81, 82, 131, 144, 149, 159, 181, 229]
	Neural Networks / Genetic Algorithm	[7, 11, 38, 40, 51–53, 60, 88, 100, 101, 122, 128, 130, 131, 134, 139, 144, 152, 164, 166, 177, 180, 188, 191, 213, 215, 216, 232]
	Others	[20, 21, 36, 80, 84, 118, 134, 163, 168, 178–180, 213, 216, 224, 226, 229, 232]
UnSup	Clustering Methods	[22, 151, 152, 191, 224]
Others Methods	Reinforcement Learning / NEAT	[50, 130, 131, 153, 154, 216]
	Graph-based Methods	[151, 152, 177]
	Statistical Methods	[4, 7, 17, 18, 20–22, 36, 40, 41, 53, 60, 61, 63, 74, 75, 78, 81–84, 87, 95, 130, 131, 144, 146, 163, 175, 185, 195, 204, 213, 216, 218, 229, 232]

There are numerous tools and packages are associated with practicing machine learning application and adapting on existing problems and more specifically on compiler autotuning problems. WEKA [94] , C5 [176], Matlab [55], R [221], Scikit-learn [184], Pybrian [203], Tensorflow [5], MLlib [155], etc..

2.5.1 Supervised learning

Supervised learning is the machine learning task of inferring a function from labeled training data [57, 158]. The learner receives a set of labeled examples as training data and makes predictions for all unseen points. This is the most common scenario associated with classification, regression and ranking problems.

Bayesian Netowrks

Bayesian Networks (BN) [77, 183] are powerful classifiers to represent the probability distribution of different variables that characterize a certain phenomenon such as the optimality of compiler optimization sequences. A Bayesian Network is a direct acyclic graph whose nodes represent variables and whose edges represent the dependencies between these variables. The probability distributions of the two optimizations depend on the program features represented by α. Additionally, the probability distribution of o_2 depends on whether the optimization o_1 is applied. Nodes representing observed variables whose value can be input as evidence to the network.

Ashouri et. al. [17, 20] proposed a Bayesian Network approach to address the problem of selecting the best compiler optimizations suitable for a embedded processor to gain speedup versus the fixed standard optimizations available at different levels of GCC compiler. They used static, dynamic and hybrid features to construct an

application feature vector and evaluated their approach with Cbench [86] and Polybench [91, 190] using BN to focus on iterative compilation and showed using the inferred compiler passes by the BN they could outperform GCC's `-O2` and `-O3` by around 50 %

Linear Models and SVMs

Linear models are one of the most popular supervised learning methods to be widely used by researchers in tackling many machine learning applications. Linear regression, nearest neighbor, and linear threshold algorithms are generally very stable [57]. Algorithms whose output classifier does not undergoes major changes in response to small changes in the training data. Moreover, SVMs are a supervised machine learning technique, used for both classification and regression, and it can apply linear techniques to non-linear problems. First, SVM transforms data into a linear space by using kernel functions, and uses a linear classifier to separate data with a hyperplane. SVM not only finds a hyperplane to separate data, but also finds the best hyperplane, so called maximum margin hyperplane, showing the largest separation from the set of hyperplanes [179].

Decision Trees and Random Forests

A binary decision tree is a tree representation of a partition of the feature space. Decision trees can be defined using more complex node questions resulting in partitions based on more complex decision surfaces [158]. Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct decision trees' habit of overfitting to their training set [57].

Fraser et al. [75] proposed to use machine learning to perform code compression. It uses IR structure of the codes to automatically infer a decision tree that separate IR code into stream that compress better. They evaluated their approach with GCC and used Opcodes which can also help predict elements of the operand stream.

Monsifrot [159] addressed the automatic generation of optimization heuristics for a target processor by machine learning. They evaluated the potential of this method on an always legal and simple transformation: loop unrolling. They used decision trees to learn the behavior of the loop unrolling optimization on the code being studied and drive to decide whether to unroll on UltraSPARC and IA-64 machines.

Fursin and Cohen [79] built an iterative and adaptive compiler framework on the SPEC applications using a modified GCC. Building a transparent framework which reuses all the compiler program analysis routines (from a program transformation database) to avoid duplications in external optimization tools.

Lokuciejewski et al. [144, 145] proposed an adaptive worst-case execution time (WCET)-aware compiler framework using Random forests for an automatic search of compiler optimization sequences that yield highly optimized code. Besides the objective functions ACET and code size, they consider the WCET which is a crucial parameter for real-time systems. To find suitable trade-offs between these objectives,

stochastic evolutionary multi-objective algorithms identifying Pareto optimal solutions for the objectives (WCET, ACET) and (WCET, code size) are exploited.

Luo et al. [149] proposed a technique to select a minimal set of representative optimization variants (function versions) for such frameworks while avoiding performance loss across available datasets and code-size explosion. They developed a novel mapping mechanism using popular decision tree or rule induction based machine learning techniques to rapidly select best code versions at run-time based on dataset features and minimize selection overhead.

Neural Networks and Genetic Algorithms

Neural networks (NN) are frequently employed to classify patterns based on learning from examples. Different neural network paradigms employ different learning rules, but all in some way determine pattern statistics from a set of training samples and then classify new patterns on the basis of these statistics [209]. A NN is a network inspired by biological neural networks which are used to estimate or approximate functions that can depend on a large number of inputs that are generally unknown. Artificial neural networks are typically specified using three components: (i) architecture, (ii) activity rule, and (iii) learning rule. Genetic Algorithm (GA) is a meta-heuristic inspired by the process of natural selection and can be paired with any other machine learning technique or work independently. A notable GA heuristic is NSGA which is a popular method for many optimization problems [54].

Cooper et al. [52, 53] in one of the early related work of literature, addressed the code size of the generated binaries by using genetic algorithm to find optimization sequences that generate small object codes. The solutions generated by this algorithm are compared to solutions found using a fixed optimization sequence and solutions found by testing random optimization sequences. Based on the results found by the genetic algorithm, a new fixed sequence is developed to reduce code size.

Knijnenburg et al. [122] proposed an iterative compilation approach to tackle the selection size of the tiling and the unrolling factors in an architecture independent manner. They evaluated their approach using several iterative strategies based on genetic algorithms, random sampling and simulated annealing and compared the results with static-techniques. The targeted compiler was native Fortran77 or `g77` compiler with full optimization on. The benchmarks used were Matrix-Matrix Multiplication (M$\times$M), Matrix-Vector Multiplication (M$\times$V) and Forward Discrete Cosine Transform.

Stephenson et al. [213] introduced Meta Optimization, a methodology for automatically fine-tuning compiler heuristics. Meta Optimization uses machine-learning techniques and specifically genetic programming to automatically search the space of compiler heuristics. The authors targeted IMPACT compiler with Spec and Mediabench [135] applications to evaluate their approach.

Cavazos and O'Boyle [38] developed a genetic algorithms based approach to automatically tune a dynamic compiler's internal inlining heuristic. The approach used a program's performance to guide the search. Genetic algorithms have been used candidate and the geometric mean of the performance of the `SPECjvm98` benchmarks was used as their fitness function.

Agakov et al. [7] introduced machine-learning models to focus on the exploration of the compiler optimization for the most promising region. Their methodology ex-

ploits a Markov chain oracle and an independent identically distributed (IID) probability distribution oracle. These two offline-learnt models bias certain optimizations over others and replace the uniform probability distribution they applied earlier for the RIC reference methodology. The authors reported significant speedup by coupling these machine-learning models with a nearest-neighbor-classifier. When predicting the probability distribution of the best compiler optimizations for a new application, the classifier first selects the training application having the smallest Euclidean distance in the feature vector space (derived by PCA). Then it learns the probability distribution of the best compiler optimizations for this neighboring application either by means of the Markov chain model or by using an IID model. These probability distribution learnt is then used as the predicted optimal distribution for the new application. It has been reported that the Markov chain oracle outperforms the IID oracle, followed by the RIC methodology using a uniform probability distribution.

Kulkarni et al. [128] used a depth-first search algorithm to produce the next sequence to evaluate in an exhaustive exploration of the phase ordering problem. In order to evaluate their method, the authors used hill-climbing, simulated annealing, genetic-algorithm and a random search on an embedded architecture.

Leather et al. [134] used grammatical evolution based on the genetic algorithm to describe the feature space and used predictive modeling on GCC 4.3.1 to evaluate the approach on Pentium 6 with mediabench to determine the loop-unrolling factor.

Purini et al. [191] have defined a machine learning based approach to downsample the compiler optimization sequences in LLVM's `-O2` and then applied machine learning to learn a model. The authors introduced a clustering algorithm to clustering sequences based on Sequence Similarity matrix by calculating the Euclidean distance between the two sequence vectors. In the experimental evaluation, they have mentioned the most frequent optimization passes with their fitness function (execution speedup) as well.

Other Supervised Methods

For conciseness proposes, we decided to classify other supervised learning methods under this subsection. These include Lazy learning, ANOVA, K-nearest neighbor, Gaussian process learning [158], and others.

Moss et al. [160] showed how to cast the instruction scheduling problem as a learning task, obtaining the heuristic scheduling algorithm automatically. They focused on the narrower problem of scheduling straight-line code (also called basic blocks of instructions). They used static and IR features of the basic block with the `SPEC` benchmark to experimentally evaluate their approach by using Geometric mean as fitness function and fold-cross-validation.

Cavazos and Moss [36] used JIT Java comiler and SPECjvm98 benchmark and rule set induction learning model to decide whether to schedule. The exploited supervised learning to induce heuristics to predict which blocks benefit from scheduling. The induced function chooses for each block between list scheduling or not scheduling the block at all. Using the induced function the authors obtained over 90% of the improvement of scheduling every block but with less than 25% of the scheduling effort.

Tournavitis et al. [226] proposed a technique using profile-driven parallelism detection in which they can overcome the limitations of static analysis, enabling to identify

more application parallelism and only rely on the user for final approval. The approach integrated profile-driven parallelism detection and machine-learning based mapping in a single framework. Moreover, the authors replaced the traditional target-specific and inflexible mapping heuristics with a machine-learning based prediction mechanism, resulting in better mapping decisions while providing more scope for adaptation to different target architectures. Finally, they have experimentally evaluated their approach against NAS and SPEC-OMP.

2.5.2 Unsupervised learning

Unsupervised learning is the machine learning task of inferring a function to describe hidden structure from unlabeled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution [96, 158]. Unsupervised learning is closely tighten with the problem of density estimation in statistics [207], however unsupervised learning also encompasses many other techniques that seek to summarize and explain key features of the data.

Clustering Methods

One of supervised learning's key subclasses is clustering. Clustering helps to downsample the chunk of unrelated compiler optimization passes into meaningful clusters that corresponds to each other, i.e. targets loop-nests or scalar values, or they should follow each other in a same sequence. The other importance of clustering/downsampling is to reduce the compiler optimization space which as mentioned in Section 2.3.3 are in tens of thousands order of magnitudes.

Thomson et al. [224] presents a new approach to reduce the training time of a machine learning based compiler. They focused on the programs which best characterize the optimization space and proposed to use a clustering technique, namely GustafsonKessel algorithm, after applying the dimension reduction process. They evaluated the clustering approach on the EEMBCv2 benchmark suite and show that we can reduce the number of training runs by more than a factor of seven.

Ashouri et al, [18,22] developed a hardware/software co-design toolchain to explore compiler design space jointly with microarchitectural properties of a VLIW processor. The authors have used clustering to derive to four (4) good hardware architectures followed by mitigating the selection of promising compiler optimization with statistical techniques such as kruskal-wallis test and pareto-optimal filtering. (This method involved with statistical methods as well. Refer to Section 2.5.3)

Martins et al. [151, 152] tackled the problem of phase-ordering by a DSE approach that uses a clustering-based selection method for grouping functions with similarities and exploration of a reduced search space resulting from the combination of optimizations previously suggested for the functions in each group. Authors used DNA encoding where program elements (e.g., operators and loops in function granularity) are encoded in a sequence of symbols, and followed by calculating the distance matrix and a tree construction of the optimization set. Consequently, they applied the compiler optimization passes already included in the DSE to measure the reduction in the total exploration time of the search space such as Genetic algorithm.

2.5.3 Other Machine Learning Methods

In this section we present the recent literature involved with using machine learning methods that could not be classified by supervised or unsupervised learning methods. Examples of these methods are Reinforcement learning, Graph-based methods and the statistical methods [158]. Indeed, some statistical methods are associated with supervised learning, i.e. Decision trees, etc, but we tried our best to provide a more comprehensive classification here, taking into account those approaches that could be classified on a third subsection.

Reinforcement Learning and NEAT

Reinforcement learning (RL) is an area of machine learning which can not be classified as supervised or unsupervised. It is inspired by behaviorist psychology and uses the notion of rewards or penalties so that a software agent interacts with an environment and maximizes his cumulative reward. The interesting different in RL is that the training ad testing phases are intermixed [158]. RL uses Markov decision process (MDP) [103] to adapt and interact with its environments. In this chapter we have provided the works done with RL in the field.

Adapting a compiler optimization problem using RL is a challenging task using this method [130]. However, we decided to group RL with a Neuroevolution approach called NEAT as suggested by [212] since NEAT can be a strong method on the pole-balancing benchmark reinforcement learning tasks. NEAT alters both the weighting parameters and structures of networks, attempting to find a balance between the fitness of evolved solutions and their diversity. It is based on applying three key techniques: (i) tracking genes with history markers to allow crossover among topologies, (ii) applying speciation (the evolution of species) to preserve innovations, and (iii) developing topologies incrementally from simple initial structures (complexifying).

McGovern et al. [153] presented two methods of building instruction scheduler using rollouts, an improved Monte carlo search [222], and reinforcement learning. The authors showed that the combined reinforcement learning and rollout approach could outperform the commercial Compaq scheduler on evaluated benchmarks from SPEC95 suite.

Coons et al. [50] used NEAT as a reinforcement learning tool for finding good instruction placements for an EDGE architecture. The authors showed that their approach could outperform state-of-the-art methods using simulated annealing in order to find the best placement.

Kulkarni et al. developed two approaches in order to tackle both the problem of selection [131] and the phase-ordering [130] of compiler optimizations. The approach for selecting the good compiler passes is done using NEAT and static features to tune Java hotspot server compiler with SPEC Java benchmarks (using two benchmarks for training and two for testing). The authors used NEAT to train decision trees for the caller and the callee whether to inline. The approach for the phase-ordering problem, formulates it as a Markov process and uses a characterization of the current state of the code being optimized to creating a better solution to the phase ordering problem. the authors technique uses NEAT to construct an artificial neural network that is capable of predicting beneficial optimization ordering for a piece of code that is being optimized.

Graph-based Methods

Graph-based methods are emerging recently as a means of exploiting many different machine learning applications on a wide range of applications from semi-supervised learning [43] to clustering and classification [35]. We decided to place them in the section related to the other machine learning methods to be more precise on our classification.

Park et al. [177] the authors introduced a novel way of of both characterizing programs using a graph-based characterization, which uses the program's intermediate representation and an adapted learning algorithm to predict good optimization sequences. In order to construct the feature vectors they used graph-kernels (refer to Section 2.4.1). The authors evaluated different characterization techniques, focusing on loop-intensive programs. They constructed prediction models that drive polyhedral optimizations, such as auto-parallelism and various loop transformation.

Nobre et al. [164] proposed an iterative compilation approach using graph-based features of the code being optimized to mitigate the selecting and the ordering of the compiler optimization passes in LLVM.

Statistical Methods

Terminology across fields is quite varied for statistical methods [89]. In statistics, where classification is often done with logistic regression or a similar procedure, the properties of observations are termed explanatory variables (or independent variables, regressors, etc.), and the categories to be predicted are known as outcomes, which are considered to be possible values of the dependent variable. In machine learning [13, 158], the observations are often known as instances, the explanatory variables are termed features (grouped into a feature vector), and the possible categories to be predicted are classes. Here we refer to the group of work in literature which involved with Frequentest procedures or multivariate distribution. Some references considered Bayesian networks, Decision trees or Random forests as a form of statistical methods, but we decided to have individual section for each to classify in a more fine-grained manner.

Pinker et al. [185] proposed an automatic iterative procedure to turn on or to turn off compiler options. This procedure is based on orthogonal Arrays that are used for a statistical analysis of profile information to calculate the main effect of the options. This approach can be used on top of any compiler that allows a collection of options to be set by the user. They showed that the proposed approach outperforms `-O3` of GCC on some SPEC benchmarks.

Haneda et al. [95] introduced a statistical technique to derive a methodology which trims down the search space considerably, thereby allowing a feasible and flexible solution for defining high performance optimization strategies. They show that the technique finds a single compiler setting for a collection of programs SPECint95 that performs better than the standard settings of GCC.

Namolaru et al. [163] proposed a general method for systematically generating numerical features from a program, and to implement it in a production compiler. This method does not place any restriction on how to logically and algebraically aggregate semantical properties into numerical features, offering a virtually exhaustive coverage of statistically relevant information that can be derived from a program. They have used static features of MilePost GCC and MiBench to evaluate their approach.

Table 2.7: *A Classification Based on Prediction Types*

Classification	References
Clustering / Down sampling	[22, 100, 122, 151, 152, 164, 165, 191, 224]
Speedup Prediction	[19–21, 40, 51, 58, 60, 79, 80, 84, 118, 131, 134, 178–181, 213, 216, 232]
Compiler Sequence Prediction	[7, 11, 17, 18, 20–22, 28, 36–41, 47, 51–53, 61, 63, 74, 78, 79, 81–83, 85, 87, 95, 100, 101, 114, 122, 127, 128, 130, 144, 146, 149–152, 159, 163, 165, 166, 168, 174, 177, 179, 180, 185, 187, 188, 191, 192, 195, 204, 213, 214, 216, 218, 224, 226, 229, 232]
Tournament / Intermediate Prediction	[19, 130, 178–180]
Feature Prediction	[58, 139, 180, 229]

Ashouri et al. [18, 22] introduced statistical technique to cluster and choosethe best compiler optimizations in a software/hardware co-design manner. The authors used multi-objective optimization and pareto-filtering to derive their micro-architectural parameters followed by ANOVA, Kruskal-wallis tes, and performance distribution graphs as their fitness function to select the compiler parameters of a VLIW [70, 72] architecture.

2.6 Prediction Types

In this chapter, we provide a full classification on the output of the aforementioned prediction models. The major classes of prediction type include the target function to predict (i) the right set of compiler optimizations to be used, (ii) the speedup of a compiler optimization sequence given an application, (iii) the right set of features to be used in order to characterize the application, (iv) the intermediate speedup (tournament) predictor, and (v) the reduction of the search space through down-sampling of optimizations. Table 2.7 shows the classification of the related literature.

Clustering and Downsampling

Cluster analysis, or clustering, is the task of grouping a set of objects in such a way that objects in the same group called a cluster, are more similar to each other than to those in other groups. It is a main task of exploratory data mining and a common technique for statistical data analysis. Cluster analysis is used in many fields, including unsupervised machine learning, pattern recognition, and compiler autotuning [158]. In order to approach reasonable solutions to the compiler optimization problems, the optimization search space needs reduced. Researchers try to find ways to reduce the large search space by orders of magnitudes. This technique is also known as downsampling.

Purini et al. [191] defined a machine learning based approach to downsample the compiler optimization sequences in LLVM's `-O2` and then applied machine learning to learn a model. They preferred not to use features from the application under analysis. Instead, their downsampling technique used various genetic algorithms or a uniform random search. Moreover, they have introduced a clustering algorithm to cluster the sequences based on Sequence Similarity matrix by calculating the Euclidean distance

between the two sequence vectors. In the experimental evaluation, they have mentioned the most frequent optimization passes with their fitness function (execution speedup).

Martins et al. [151, 152] tackled the problem of phase-ordering by a DSE approach that uses a clustering-based selection method for grouping functions with similarities. They explored a reduced search space resulting from the combination of optimizations previously suggested for the functions in each group. Authors used DNA encoding where program elements (e.g., operators and loops in function granularity) are encoded in a sequence of symbols and followed by calculating the distance matrix and a tree construction of the optimization set. Consequently, they applied the compiler optimization passes already included in the DSE to measure the reduction in the total exploration time of the search space.

Speedup Prediction

Speedup predictive modeling is the process of constructing, testing, and validating a model to predict an unobserved outcome. The model is constructed based on the characterization of a state. The state being characterized is the code being optimized and the predicted outcome corresponds to the speedup metric calculated by normalizing the execution time of the current optimization sequence by the execution time of the baseline optimization sequence. The general formulation of the optimization problem is to construct a function that takes as input the features of the unoptimized program being compiled. This model takes as an input a tuple, (F, T). F is the feature vector of the collected instrumentation of the program being optimized, and T is one of the several possible compiler sequences predicted to perform well on this program. Its output is a prediction of the speedup T should achieve when applied to the original code.

Dubach et al. [60, 61] presented a new machine learning based technique to automatically predict the speedup of a modified program using a performance model based on the code features of the tuned programs. The authors used static-features from SUIF compiler infrastructure [238] for VLIW and compared the result with non-feature-based alternative predictors such as mean predictors, sequence encoding-based predictors, and reaction based predictors.

Leather et al. [134] used grammatical evolution based on the genetic algorithm to describe the feature space and used predictive modeling on GCC 4.3.1 to evaluate the approach on Pentium P6 with Mediabench to determine the loop-unrolling factor.

Park et al. [177–181] proposed several predictive modeling approaches to tackle the problem of selecting the right set of compiler optimizations. Authors used static program features instead of hardware-dependent features on a polyhedral space [178]. In [177], the authors used Control Flow Graph (CFG) with graph kernel learning to construct a machine learning model. First, they constructed CFGs by using the LLVM compiler and convert the CFGs to Shortest Path Graphs (SPGs) by using the Floyd-Warshall algorithm. Then, they apply the shortest graph kernel method [32] to compare each one of the possible pairs of the SPGs and calculate a similarity score of two graphs. The calculated similarity scores for all pairs are saved into a matrix and directly fed into the selected machine-learning algorithm, specifically SVMs in their work. In [181], they used user-defined patterns as program features. They use a pattern-driven system named HERCULES [117] to derive arbitrary patterns coming from users. They focused on defining patterns related to loops: the number of loops having memory accesses,

having loop-carried dependencies, or certain types of data dependencies. These works use static program features mainly focusing on loop and instruction mixes.

Compiler Sequence Prediction

A compiler sequence predictor is a type of prediction model which output the best set of compiler passes or sequences to apply on a given application. Application characterization serves as input to a model, and the model predicts a probability distribution of optimizations to apply to that program. We term this model a sequence predictor because it can be used to construct a sequence of optimizations [179]. The general formulation of a sequence prediction is the optimal compiler optimization sequence $\bar{\boldsymbol{o}} \in \mathcal{O}$ that maximizes the performance of an application is generally unknown. However it is known that the effects of a compiler optimization o_i might depend on whether another optimization o_j has been applied. Additionally, it is known that the compiler optimization sequence that maximizes the performance of a given application depends on the application itself. The reason why the optimal compiler optimization sequence $\bar{\boldsymbol{o}}$ is unknown a priory is because it is not possible to capture in a deterministic way the dependencies among the variables in the vectors $\bar{\boldsymbol{o}}$ and $\boldsymbol{\alpha}$. There is no way to identify an analytic model to exactly fit the vector function $\bar{\boldsymbol{o}}(\boldsymbol{\alpha})$ [20].

Cooper et al. [52,53] described a prototype system that used biased random search to discover a program-specific compilation sequence that minimizes an explicit, external objective function. The result was a compiler framework that adapts its behavior to the application being compiled, to the pool of available transformations, to the objective function, and to the target machine.

Cavazos et al. [41] proposed a offline machine learning based model which can be used to predict the good set of comiler optimization sequences. It uses performance counters as a means of determining good compiler optimization setting. The authors showed using this they can outperform existing techniques used by static code features.

Ashouri et. al [20] proposed a Bayesian Network (BN) approach which was fed by either of static, dynamic or a hybrid characterization vector of an application. The BN model could subsequently induce a probabilistic model to infer from. The authors incorporate this model with iterative compilation to drive to good compiler sequences and showed to outperform the state-of-the-art models.

Tournament and Intermediate Prediction

A tournament predictor takes as input a triple corresponding to the characterization of the program and two optimization sequences. This model predicts whether the speedup of the first optimization sequence will be more or less than the second optimization sequence [179]. Since intermediate speedup predictors are behaving more or less the same as a tournament in a sense that it works on individual optimizations to be applied on the current status, we decided to bundle the two approaches in one classification. An intermediate speedup prediction (it has been referred to as Reaction-based modeling [40] as well), uses a model to predict the current best optimization (from a given set of optimizations) that should be applied based on the characteristics of code in its present state. Determining the correct phase ordering of optimizations in a compiler is a difficult problem to solve. In the absence of an oracle to determine the correct ordering of optimizations, we must use a heuristic to predict the best optimization to

use. An intermediate sequence approach needs multiple profiling of the application being optimized (based on the characteristics of code in its present state) predictions, therefore, on a long run it turns to its disadvantage comparing to other methods. nevertheless, it has been shown an effective method specifically to tackle the phase-ordering problems [19, 130].

Cavazos et al. [40] used Artificial Neural Networks (ANN) to predict speedup of an application with two different prediction types: a feature-based model and a Reaction-based. the feature-based model was fed by static features and predicts the program speedup and the reaction-based model was fed by a target transformation and spedup on canonical transfomtion and it predicts the transformation speedup.

Park et al. [179] have defined and used the term tournament predictor in order classify whether an optimization was better off to choose or not as immediate optimization. Park et al. evaluated three prediction models: sequence, speedup, and tournament using program counters (PC) derived by PAPI [161] on several benchmarks (PolyBench, NAS, etc.) using the Open64 compiler. They showed on many occasions tournament predictors can outperform other techniques.

Kulkarni and Cavazos [130] developed a new approach that automatically selects good optimization orderings on a per-method basis within a dynamic compiler. The approach formulates the phase-ordering problem as a Markov process and uses a characterization of the current state of the code being optimized to creating a better solution to the phase ordering problem. The authors technique uses neuro-evolution (NEAT) to construct an artificial neural network that is capable of predicting beneficial optimization ordering for a piece of code that is being optimized.

Ashouri et al. [19] demonstrated a predictive methodology in order to predict the intermediate speedup obtained by an optimization from the configuration space, given the current state of the application. The fitness function for the intermediate speedup was the ratio between the execution times of the program before and after the optimization process. They exploited predictive models to correlate the current state of the dynamic features of the application under study with the current state of the compiler optimization to come up with a speedup value and utilize heuristics to search that space.

Feature Prediction

Building empirical models is an automatic process that requires minimal user intervention and does not rely on any prior knowledge about the relationship between the predictor variables and the response. Due to the iterative nature of the process, empirical models with a desired level of accuracy can be built simply by collecting more data. Empirical models can also discover arbitrarily complex interactions between predictor variables. As a result, empirical models have a fair amount of interpretive value and can reveal interesting characteristics of the underlying system [229]. During this process choosing the right set o features to choose to characterize an application is crucial.

Vaswani et al. [229] built related program performance to settings of compiler optimization flags, associated heuristics and key microarchitectural parameters. Unlike traditional analytical modeling methods, this relationship is learned entirely from data obtained by measuring performance at a small number of carefully selected compiler/microarchitecture configurations. The authors evaluated different learning techniques in this context to use the generated models to (i) predict program performance at arbitrary

Table 2.8: *A Classification Based on Space Exploration Methods*

Classification	References
Adaptive Compilation	[4, 11, 14, 39, 41, 48, 51–53, 58, 61, 63, 74, 78, 79, 81–84, 87, 95, 100, 101, 126, 130, 134, 144, 146, 149, 150, 163, 174, 175, 185, 195, 198, 213, 214, 216, 218, 229, 239]
Iterative Compilation	[4, 7, 11, 14, 17, 18, 18–22, 28, 37–41, 44, 47, 51–53, 58, 60, 61, 63, 74, 78–85, 87, 95, 100, 101, 118, 122, 124, 126–131, 134, 144, 146, 146, 149–152, 163–166, 168, 174, 175, 177–181, 185, 187–189, 195, 198, 204, 213, 214, 216, 224, 226, 229, 232, 239, 241, 242]
Non-iterative Compilation	[75, 188, 189, 192, 216, 230, 233, 235]

compiler/microarchitecture configurations, (ii) quantify the significance of complex interactions between optimizations and the microarchitecture, and (iii) efficiently search for optimal settings of optimization flags and heuristics for any given microarchitectural configuration using SPEC benchmark suite within a 5% error threshold.

Li et al. [139] used a machine learning based compilation optimization focused on feature processing. The authors have adapted this method based on the application under analysis and Apart from user defined static features, they design a method to generate lots of static features by template and select best ones from them. Furthermore, they observed that feature value changes during different optimization phases and implement a feature extractor to extract feature values at specific phases and predict optimization plan dynamically.

Ding et al. [58] proposed an autotuning framework capable of incorporating different input in a two-level approach to overcome the challenge of input sensitivity. They leveraged the Petabricks language [14] and its compiler and used an input-aware learning technique to differentiate between inputs. The work clustered the space and chose its centroid for auto-tuning (i) to identify a set of configurations for each class of inputs and (ii) to identify a production classifier that is able to efficiently identify the best landmark configuration to use for a new input.

2.7 Optimization Space Exploration Techniques

As previously mentioned in Section 2.3.3, traversing the large compiler optimization space made by the different combinations of optimizations requires a proper exploration strategy. Random, iterative compilation, genetic algorithm, and DoE are among the most mentioned strategies in the literature. In a broader perspective, the strategies are derived by the type of chosen space exploration. Design space exploration (DSE) is the activity of exploring design alternatives before implementation. The ability to operate on the space of design candidates makes DSE useful for many engineering tasks, such as rapid prototyping, optimization, and system integration. In general, different applications could impose different energy and performance requirements. The overall goal of the DSE phase to optimally traverse and configure the exploration parameters [172, 173]. In this chapter we classify the different exploration strategies used by researchers in literature to overcome this challenge. Table 2.8 represents our fine-grain classification of the different related works based the exploration type.

2.7.1 Adaptive Compilation

Adaptive optimization is a technique in traversing the optimization space that performs dynamic recompilation of portions of a program based on the current execution profile. The profiling provides enough features so the compiler can decide on what portion of the code to be recompiled. With a simple implementation, an adaptive optimizer may simply make a trade-off between just-in-time compilation and the instructions being interpreted. An adaptive compiler uses a compile-execute-analyze feedback loop to find the combination of optimizations and parameters that minimizes some performance goal, such as code size or execution time [51]. As an example, adaptive optimization may take advantage of local data conditions to optimize away branches and to use inline expansion to decrease the cost of procedure calls.

Cooper et al. [51,53] developed an adaptive compiler, ACME, plus a GUI to control the process of recompilation and exploration of different compiler optimizations. The authors have developed a technique called virtual execution to address the issue of multiple execution of code being optimized. Virtual execution runs the program a single time and preserves information that allows us to accurately predict the performance of different optimization sequences without running the code again. This technique later called speedup prediction in the literature.

Fursin and Cohen [79] Built an iterative and adaptive compiler framework on the SPEC benchmark suit using a modified GCC. The authors also developed a transparent framework which reused all the compiler program analysis routines from a program transformation database to avoid duplicates in external optimization tools.

2.7.2 Iterative Compilation

In computer science, an iterative method is a mathematical procedure that generates a sequence of improving approximate solutions for a class of problems. A specific implementation of an iterative method, including the termination criteria, is an algorithm of the iterative method. Iterative compilation is by far the most commonly used exploration technique for the compiler optimization field. Many recent works found this technique interesting and successful either (i) alone [28, 124], (ii) combined with machine learning techniques [7, 20, 41], or (iii) combined with other search and meta-heuristics techniques such as random exploration [28], DSE techniques [151, 165], etc.. It is well known that Random Iterative Compilation (RIC) can improve application performance compared with static handcrafted compiler optimization sequences. Additionally, given the complexity of the iterative compilation problem, it has been proved that drawing compiler optimization sequences at random is as good as applying other optimization algorithms such as genetic algorithms or simulated annealing [7, 20, 41, 46].

Bodin et al. [28] investigated an early path towards analysis of the applicability of iterative search techniques in program optimization. The authors showed that iterative compilation, despite usually being too expensive for general purpose computing, is applicable to embedded applications where the cost is easily amortised over the number of embedded systems produced. They used profile feedback in the form of execution time and to downsample the space on restricted optimization passes. In this work, the authors investigated the unrolling, tiling and padding parameters, however, in the latter works may other researchers expanded the experimental parameters to scalar, loop-nest,

etc. optimization as well [18, 22, 63, 85, 119–121, 179, 188].

Knijnenburg et al. [122] proposed an iterative compilation approach to tackle the selection size of the tiling and the unrolling factors in an acchitecture-independant manner. They evaluated their approach using several iterative strategies based on genetic algorithms, random sampling and simulated annealing and compared the results with static-techniques. The targeted compiler was native Fortran77 or g77 compiler with full optimization on. The benchmarks used were Matrix-Matrix Multiplication (M $\times$ M), Matrix-Vector Multiplication (M $\times$ V) and Forward Discrete Cosine Transform.

Triantafyllis et al. [227] proposed the Optimization-Space Exploration (OSE) compiler organization, a practical iterative compilation strategy applicable to optimizations in general-purpose compilers. The authors used the compiler writer's knowledge encoded in the heuristics to select a small number of promising optimization alternatives for a given code segment and limited the compile time by evaluating only these alternatives for hot code segments using a general compile time performance estimator.

Killian et al. [118] investigated the ways to combine vectorization reports with iterative compilation and code generation. The authors summarized their insights and patterns on how the compiler vectorizes code and leveraged the obtained knowledge to design a Support Vector Machine classifier to predict the speedup of a program given a sequence of optimization.

Martins et al. [151] tackled the problem of phase-ordering by a DSE approach that uses a clustering-based selection method for grouping functions with similarities and exploration of a reduced search space resulting from the combination of optimizations previously suggested for the functions in each group. Authors used DNA encoding where program elements (e.g., operators and loops in function granularity) are encoded in a sequence of symbols, and followed by calculating the distance matrix and a tree construction of the optimization set. Consequently, they applied the compiler optimization passes already included in the DSE to measure the reduction in the total exploration time of the search space such as Genetic algorithm.

2.7.3 Non-iterative Compilation

Unlike iterative compilation, a non-iterative approach tries to presents a global optimization approach for a class of compiler optimization problem. Few works have been seen recently with compiler optimization problems. Moreover, in recent literature terms like wild-guess are used with this classification [138]. The high levels of indeterminism are making it hard problem to solve with non-iterative methods. As we noted in Section 2.2, the driving force towards using approximation methods such as iterative compilation and machine learning was the inability of researchers to tackle the phase-ordering problem by straightforward non-iterative approaches. Thus, this branch of compiler autotuning has suffered from further investigation. The polyhedral compilation community [5] has gained attention in many interesting direction and is an orthogonal approach of optimizing compilers. We will brief this recently built community in Section 2.8.

Vegdahl et al. [230] have used constant-unfolding to produce code sequences that can be more compacted on a horizontal target architecture. A constant-unfolding axiom replaces a constant by a constant expression of equal value. The goal is to make use of

[5] http://polyhedral.info/

Table 2.9: *A Classification Based on Target Platform*

Classification		References
Target Platform	Embedded Domain	[4, 7, 17, 18, 20–22, 40, 53, 60, 74, 75, 81, 82, 84, 124, 128, 151, 152, 163–166, 175, 180, 245]
	Desktop	[4, 7, 11, 14, 19, 21, 30, 36–39, 41, 44, 47, 51–53, 61, 74, 78–85, 87, 95, 100, 101, 118, 122, 126, 130, 131, 134, 139, 144, 144, 146, 149, 150, 159, 163, 164, 174, 175, 177–181, 185, 187–189, 191, 192, 195, 198, 204, 213–216, 218, 224, 226, 229, 232, 241, 242]
	HPC Domain	[4, 14, 58, 63, 78, 81, 82, 141, 156, 166, 177, 187, 188, 195, 204, 218, 225, 226, 241, 242]

constants which are hard-wired into the micromachine, replacing difficult-to-generate constants with expressions involving only hard-wired constants.

Whitfield et al. [235] proposed a framework includes a technique that facilitates an analytical investigation of code-improving transformations using the Gospel specifications [234]. It also contained a tool, Genesis, that automatically produces a transformer that implements the transformations specified in Gospel. The authors demonstrate the usefulness of the framework by exploring the enabling and disabling properties of transformations.

2.8 Target Domain

Choosing the right set of optimizations to apply on a given application is heavily correlated with the type of compiler, target architecture, and target platform to be tuned. Avoiding generic optimizations was one of the driving forces behind the compiler autotuning field. When incorporating machine learning techniques, a framework should be adaptable based on a given application or target platform. An optimized code segment for a given compiler or a target platform may not yield the same optimality on a different compiler or platform. Each compiler and target platform combination have different ways of generating the binaries and executing the code segments. Moreover, optimization techniques might be useful for a class of applications, e.g. security, office, etc., but they might not be for other classes. To this end, we classify the literature based on the very three aforementioned subclasses. These are shown in Tables 2.9, 2.10 and 2.13.

2.8.1 Target Platform

Today, essentially all programming for desktop and HPC application is done in high-level languages, as is most programming for embedded applications. this development mans that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target [182]. For example, consider special loop instructions found in an application. Assume that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. As a result the loop instruction may be a mismatch having different target instruction sets ISAs or architectures. Choosing the right set of optimizations given an architecture is a necessary task. A classification based on the type of target platform is shown in Table 2.9.

Embedded Domain

The simplest definition of embedded computing is that embedded is all computing that is not general purpose (GP), where general-purpose processors are the ones in today's notebooks, PCs, and high performance servers. This is not to say that general-purpose processors are not used in embedded applications (they sometimes are), but rather that any processor expected to perform a wide variety of very different tasks is probably not embedded. Embedded processors include a large number of interesting chips: those in cars, in cellular telephones, in pagers, in game consoles, in appliances, and in other consumer electronics [73]. Some of the more notable embedded architectures are Very Long Instruction Word (VLIW) [70,71], Big.Little heterogeneous architecture of ARM [109]. There are many recent low-cost implementation boards with different SoC specifications such as Raspberry Pi [228], Texas Instrument's Pandaboard [106], etc.. Readers can refer to the already available surveys on the field of embedded computing and FPGAs in [15,49,98].

One of the key differences of leveraging compiler optimization techniques for embedded domain is the trade-off between the application's code-size, performance and the power consumption. Code-size optimization specially in VLIW architecture has been extensively investigated [64, 146]. However, due to recent advancements in the embedded SoC, code-size is no longer a main issue, thus the focus is shifted towards the pareto-frontiers of performance, power and energy metrics [16,172,173]. Moreover, compiler optimization techniques can be exploited for this task as well [22,70,166].

Dubach et al. [60] presented a new machine learning based technique to automatically predict the speedup of a modified program using a performance model based on the code features of the tuned programs. The authors used static-features from SUIF infrastructure [238] as compiler toolchain for VLIW, and compared the result with Non-feature-based alternative predictors such as Mean predictor, Sequence encoding-based and reaction based predictors.

Namolaru et al. [163] proposed a general method for systematically generating numerical features from a program, and to implement it in a production compiler. This method does not put any restriction on how to logically and algebraically aggregate semantical properties into numerical features, offering a virtually exhaustive coverage of statistically relevant information that can be derived from a program. They have used static features of MilePost GCC and MiBench to evaluate their approach on an ARC 725D embedded processor.

Ashouri et al. [17] proposed a Bayesian network approach using hardware-independant features which could predict the right set of compiler optimizations for a given application and a chosen dataset. Since having a different dataset changes the runtime features of an application, this methodology is able to treat a new dataset as new application and still infer from the already trained model to select the best set of compiler optimizations. The authors evaluated their approach using GCC, Cbench and embedded Pandaboard and observed around 40% speedup against the GCC's standard optimization levels.

Desktop and Workstations

Although the majority of the research has been done for desktop computers, recently the focus has shifted towards the both ends of the architectural spectrum: embedded and high performance computing. To understand what falls into the category of embedded

computing, it is instructive to note what is not a requirement for embedded devices. Lifetimes of embedded devices are very different from the three-year obsolescence cycle of general-purpose machines. Few embedded devices have upgrade requirements. For example, avid automotive enthusiasts change the chips in their cars, but these are usually ROMs, not processors. Historically, computing power was a distinguishing factor, in that embedded electronics were mainly deployed to control the appliance in which they were embedded. Such applications had small requirements for speed or generality. In this chapter we have classified those works by their experimental setup where the target platform was not either of the two ends. This classification is shown in Table 2.9.

Thomson et al. [224] presented a new clustering approach to reduce the training time of a machine learning based compiler. This is achieved by focusing on the programs which best characterize the optimization space. The leveraged GustafsonKessel algorithm after applying the dimension reduction process. They evaluated the clustering approach with the EEMBCv2 benchmark suite and an Intel Core 2 Duo E6750 machine. They showed that they could reduce the number of training runs by more than a factor of 7.

Park et al. [179] proposed the term tournament predictor in order classify whether an optimization was better off to choose or not as immediate optimization. The authors evaluated all three prediction models namely, flags, speedup and tournament using program counters (PC) derived by PAPI on Polybench, NAS, etc. and other benchmarks using Open64 compiler. Their target machines were two different Intel Quads and an Intel Xeon E5335 and the authors observed the sequence that gived them good performance on one machine usually worked well on other machines as well.

HPC Domain

There are fundamental differences between a cluster and supercomputer. For instance, mainframes and clusters run multiple programs concurrently and support many concurrent users versus supercomputers which focus on processing power to execute a few programs or instructions as quickly as possible and to accelerating performance to push boundaries of what hardware and software can accomplish [34]. However, for conciseness purposes in this chapter we have placed the recent works having used mainframes and clusters together with those having supercomputers as their experimental setup.

Tiwari et al. [225] proposed a scalable and general-purpose framework for autotuning compiler-generated code. They combine Active Harmony's parallel search backend with the CHiLL compiler transformation framework [45] to generate in parallel a set of alternative implementations of computation kernels and automatically select the one with the best-performing implementation

Ding et al. [58] presented an autotuning framework capable of leveraging different input in two-level approach. It has been upon the Petabricks language and its compiler [14]. It uses input-aware learning technique to differentiate between inputs. It clusters the space and choose its centroid for auto-tuning. The two level approach consists of identify a set of configuration for each class of inputs and produce a classifier to efficiently identifies the best optimization to use for a new input.

Fang et al. [63] proposed an iterative optimization approach for the data center (IODC). They demonstrated that the data center offers a context in which the challenges

Table 2.10: *A Classification Based on Target Compiler*

Classification		References
Target Compiler	GCC	[4, 7, 17, 20, 21, 40, 47, 61, 63, 74, 75, 79, 81–83, 87, 95, 100, 114, 134, 139, 149, 150, 163, 174, 175, 177, 180, 181, 185, 187–189, 204, 216, 218, 224, 229, 232, 241, 242]
	LLVM	[18–22, 151, 152, 164–166, 178, 180, 191]
	Intel-ICC	[30, 47, 74, 118, 177, 180, 181, 188, 204, 218, 241]
	Just-in-time Compiler	[36–39, 101, 130, 131, 180, 192, 195, 204, 216]
	Java Compiler	[36–39, 101, 130, 192]
	Polyhedral Model	[29, 30, 178, 180, 187–189, 225, 241, 242]
	Others	[4, 7, 11, 14, 18, 21, 41, 44, 51–53, 60, 74, 78–82, 84, 85, 122, 126, 130, 131, 144, 144, 146, 151, 152, 159, 179, 180, 192, 195, 198, 199, 213–216, 225, 226, 232]

of iterative compilation such as requiring a large number of runs can be overcome. The idea was to spawn different combinations across workers and recollect performance statistics at the master, which then evolves to the optimum combination of compiler optimizations. Moreover, they have evaluated their approach using a MapReduce and a computer intensive approach

Nobre et al. [166] proposed to use iterative compilation to find good sequences of optimization in which they are beneficial to optimize energy consumption of a running application. In this work they authors evaluated the impact of compiler pass phases as a means to reduce the energy consumed by a set of programs/functions when comparing with the use of the standard compiler phase orders provided by, e.g., Ox flags. They used Clang plus LLVM compiler targeting a multicore ARM processor in an ODROID board and a dual x86 desktop representative of a node in a Supercomputing center.

2.8.2 Target Compiler

In ILP and Superscalar systems, the compiler has primary responsibility for finding and organizing parallelism. Parallelism is the key to performance, price/performance, power, and cost. Finding the right trade offs among these is an art, and the algorithms for analysis and transformation of programs are complicated and difficult to implement correctly. The typical investment for a compiler backend before maturity is measured in man-decades, and it is common to find compiler platforms with man-century investments [70, 72]. Table 2.10 classifies the recent literature based on the type of the compiler framework used.

GCC

GNU compiler collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain. The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example. GCC has been ported to a wide variety of processor architectures, and is widely deployed as a tool in the development of both free and proprietary software. GCC is also available for most embedded systems including ARM-based, AMCC, and Freescale Power Architecture-based chips. The

Table 2.11: *Default Compiler Passes Inside GCC's -O3*

Compiler Passes
-fauto-inc-dec -fbranch-count-reg -fcombine-stack-adjustments -fcompare-elim
-fcprop-registers -fdce -fdefer-pop -fdelayed-branch -fdse -fforward-propagate
-fguess-branch-probability -fif-conversion2 -fif-conversion -finline-functions-called-once
-fipa-pure-const -fipa-profile -fipa-reference -fmerge-constants -fmove-loop-invariants
-freorder-blocks -fshrink-wrap -fsplit-wide-types -fssa-backprop -fssa-phiopt
-ftree-bit-ccp -ftree-ccp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-dce
-ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre -ftree-phiprop -ftree-sink
-ftree-slsr -ftree-sra -ftree-pta -ftree-ter -funit-at-a-time -fthread-jumps
-falign-functions -falign-jumps -falign-loops -falign-labels -fcaller-saves
-fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively -fexpensive-optimizations -fgcse -fgcse-lm
-fhoist-adjacent-loads -finline-small-functions -findirect-inlining -fipa-cp
-fipa-cp-alignment -fipa-bit-cp -fipa-sra -fipa-icf -fisolate-erroneous-paths-dereference
-flra-remat -foptimize-sibling-calls -foptimize-strlen -fpartial-inlining -fpeephole2
-freorder-blocks-algorithm=stc -freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop -fsched-interblock -fsched-spec -fschedule-insns -fschedule-insns2
-fstrict-aliasing -fstrict-overflow -ftree-builtin-call-dce -ftree-switch-conversion
-ftree-tail-merge -fcode-hoisting -ftree-pre -ftree-vrp -fipa-ra -finline-functions
-funswitch-loops -fpredictive-commoning -fgcse-after-reload -ftree-loop-vectorize
-ftree-loop-distribute-patterns -fsplit-paths -ftree-slp-vectorize -fvect-cost-model
-ftree-partial-pre -fpeel-loops -fipa-cp-clone

compiler can target a wide variety of platforms and thus many research has been done using this framework [210, 211].

GCC out-of-the-box, does not support playing with the phase of compiler passes as its pass manager forces predefined ordering no matter what the requested order were. However, modifying the pass manager can theoretically enables the tackling the phase-ordering of the compiler optimizations. GCC optimizer is now supporting different predefined levels of fixed standard optimization levels such as `-Ofast, -O1, -O2 and -O3`. Refer to the Table 2.11 for the list of optimization passes inside GCC's `O3`.

LLVM

Low Level Virtual Machine (LLVM) is a collection of modular and reusable compiler and toolchain technologies used to develop compiler front ends and back ends. Latner and Vikram [133] described LLVM as a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, language-independent type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages uniformly and efficiently. Table 2.12 represents the fixed ordering of highest LLVM's standard optimization. There are as many as 157 compiler passes in `-O3`. Among those, some are analysis passes (i.e. basicaa, memdep, etc) which do not transform the code directly but rather provide certain information for the compiler. The rest are the transform passes (i.e. adce, licm, loop-rotate, etc) which do the actual transformation on the source-code. Table 2.12 represents the optimization passes inside LLVM's `O3`.

Recently, LLVM's community is becoming a vibrant research community towards porting and building new features into the different LLVM sub-modules e.g. LLVM's opt, LLVM's clang, LLVM's llc, etc. and there are many research papers associated

Table 2.12: *Default Compiler Passes Inside LLVM's -O3*

Compiler Passes
-tti -targetlibinfo -tbaa -scoped-noalias -assumption-cache-tracker -forceattrs -inferattrs -ipsccp -globalopt -domtree -mem2reg -deadargelim -basicaa -aa -domtree -instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inline -functionattrs -argpromotion -domtree -sroa -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -basicaa -aa -domtree -instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -basicaa -aa -licm -loop-unswitch -simplifycfg -basicaa -aa -domtree -instcombine -loops -scalar-evolution -loop-simplify -lcssa -indvars -aa -loop-idiom -loop-deletion -loop-unroll -basicaa -aa -mldst-motion -aa -memdep -gvn -basicaa -aa -memdep -memcpyopt -sccp -domtree -demanded-bits -bdce -basicaa -aa -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -basicaa -aa -memdep -dse -loops -loop-simplify -lcssa -aa -licm -adce -simplifycfg -basicaa -aa -domtree -instcombine -barrier -basiccg -rpo-functionattrs -elim-avail-extern -basiccg -globals-aa -float2int -domtree -loops -loop-simplify -lcssa -loop-rotate -branch-prob -block-freq -scalar-evolution -basicaa -aa -loop-accesses -demanded-bits -loop-vectorize -instcombine -scalar-evolution -aa -slp-vectorizer -simplifycfg -basicaa -aa -domtree -instcombine -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -basicaa -aa -instcombine -loop-simplify -lcssa -aa -licm -scalar-evolution -alignment-from-assumptions -strip-dead-prototypes -globaldce -constmerge

with using and building LLVM [6].

Intel-ICC

Intel's propriety compiler, also known as ICC or ICL, is a group of C and C++ compilers from Intel available for Windows, OS X, Linux and Intel-based Android devices [7]. The compilers generate optimized code for IA-32 and Intel 64 architectures, and non-optimized code for non-Intel but compatible processors, such as certain AMD processors. It provides Intel-ICC's compiler options provide general optimizations e.g. `O1, O2, O3` and processor's specific optimizations depending on the target platform [8]. Moreover, Interprocedural Optimization (IPO) is an automatic, multi-step process that allows the compiler to analyze your code to determine where you can benefit from specific optimizations. With IPO options, you may see additional optimizations for Intel microprocessors than for non-Intel microprocessors.

Franke et al. [74] proposed an approach called source-level transformations and the probabilistic feedback-driven search for good transformation sequences within a large optimization space in which it is based on exploring the optimization space, and is focused on localised search of good areas. The authors applied the technique to UTDSP benchmark [136] using an embedded processor and observed performance gain of 1.22 when they used ICC versus GCC.

Just-in-time Compiler

Software systems have been using just-in-time compilation (JIT) techniques since the 1960s. Broadly, JIT compilation includes any translation performed dynamically, after a program has started execution. JIT compilation, also known as dynamic translation, is compilation done during execution of a program at run time rather than prior to execution. Most often this consists of translation to machine code, which is then executed directly, but can also refer to translation to another format [23]. High level benefits of using a JIT compiler can be summarized as: (i) Compiled programs run faster, especially if they are compiled into a form that is directly executable on the underlying

[6] http://llvm.org/pubs/
[7] https://software.intel.com/en-us/intel-compilers
[8] http://scv.bu.edu/computation/bladecenter/manpages/icc.html

hardware. (ii) Interpreted programs tend to be more portable, and (iii) Interpreted programs can access tun time information. In general, JIT compilation is a form of dynamic compilation, and allows adaptive optimization such as dynamic recompilation. There are many implementation of JIT compilers targeting different programming languages. Majic a Matlab JIT compiler [12], OpenJIT [167] a Java JIT compiler, IBM's JIT compiler targeting Java virtual machine [219], etc..

Sanchez et al. [195] used SVMs to learn models to focus on auto-tuning the JIT compiler of IBM Testarossa and build compilation plan. The experimentally evaluated the learned and observed that the models outperforms out-of-the-box Testarossa on average for start-up performance, but underperforms Testarossa for throughput performance. They also generalized the learning process from learning on SPECjvm98 to DaCapo benchmakrs.

Java Compiler

A Java compiler is a compiler for the programming language Java. The most common form of output from a Java compiler is Java class files containing platform-neutral Java bytecode, but there are also compilers that emit optimized native machine code for a particular hardware/operating system combination. The Java virtual machine (JVM) loads the class files and either interprets the bytecode or just-in-time compiles it to machine code and then possibly optimizes it using dynamic compilation so it can be classifies under JIT compilers as well [6, 112]. Some of the notable research works [36, 101] including a work tackling the phase-ordering problem have been done using Java JIT compiler [130]. We have already mentioned these in the Section 2.5.3.

Polyhedral Model

Polyhedral compilation encompasses the compilation techniques that rely on the representation of programs, especially those involving nested loops and arrays, thanks to parametric polyhedra [66] or Presburger relations [236], and that exploit combinatorial and geometrical optimizations on these objects to analyze and optimize the programs. Initially proposed in the context of compilers-parallelizers, it is now used for a wide range of applications, including automatic parallelization, data locality optimizations, memory management optimizations, program verification, communication optimizations, SIMDization, code generation for hardware accelerators, high-level synthesis, etc. There has been experience in using such techniques in static compilers, just-in-time compilers, as well as DSL compilers.

Numerous scientific and compute-intensive applications spend most of their execution time in loop-nests that are suitable for high-level optimizations. Typical examples include: dense linear-algebra codes and stencil-based iterative methods [231]. Polyhedral compilation is a recent attempt to use a mathematical representation, focusing on the loop-nest [26, 30, 31, 143, 188]. We refrain from focusing more on this interesting topic as it it outside the scope of the survey.

Other Compilers

In this survey, we focused on the classification of the more widely known compiler framework in autotuning field, However, there are numerous other well known compil-

Table 2.13: *A Classification Based on Target Benchmark*

Classification		References
Target Benchmark	Cbench / MiBench	[17, 19–22, 41, 47, 61, 78, 81–83, 87, 126, 128, 134, 139, 144, 146, 159, 163, 191, 213, 215, 232]
	Polybench	[4, 20, 21, 30, 44, 118, 141, 166, 177–181, 187–189, 191, 218, 225, 241, 242]
	SPEC	[36–38, 41, 48, 79, 80, 84, 85, 100, 101, 131, 150, 159, 174, 175, 180, 185, 195, 198, 213–216, 226, 229, 232]
	Others	[4, 7, 11, 14, 18, 21, 37–40, 44, 47, 48, 52, 53, 58, 60, 63, 74, 75, 78, 81–84, 87, 95, 101, 114, 124, 130, 131, 134, 144, 144, 146, 149, 151, 152, 164, 179, 180, 187, 188, 192, 204, 213, 216, 218, 224–226]

ers which worth mentioning. We classified all work related to use of other compiler in Table 2.10 under Others subfield. Here we review a couple of them.

Stanford University Intermediate Format (SUIF) compiler [238] is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. SUIF is a fully functional compiler that takes both Fortran and C as input languages. The parallelized code is output as an SPMD (Single Program Multiple Data) parallel C version of the program that can be compiled by native C compilers on a variety of architectures. It has been used in many research works of recent literature [44, 48, 60].

Fortran compilers include many different implementations and different ports such as Open64 [56], GNU Fortran [206], XL Fortran [125], Salford Fortran 77 compiler [132], etc.. There are widely used in literature [11, 122, 198, 199].

2.8.3 Benchmarks

Developing a software kernel in which they can be representative of a class of generic applications has been always an essential task in computer benchmarking. Evaluating performance of a model wouldn't be taken place had the right benchmark wasn't to be used. A proposed mathematical approach might be fruitful on a few application but might not be yielding results on many others. Therefore, researchers have devoting efforts on developing better and more Representative benchmarks for every target domain and subfield.

Cbench - MiBench

The Cbench suite [86] is a collection of open-source programs with multiple data sets assembled by the community to enable realistic workload execution and targeted by many different compilers such as GCC, LLVM, etc.. The source code of individual programs is derived from MiBench [92] and simplified to facilitate portability; therefore, it has been targeted in autotuning and iterative compilation research work. Each application within the benchmark suite is coming with at least 20 different inputs and it makes it a more complete suite to run experimentation with. Table 2.14 shows the list of applications inside Cbench suite. Cbench is used in numerous recent literature and we classify them in Table 2.13 under its category.

Polybench

The Polybench benchmark suite [91, 190] consists of benchmarks with static control parts. The purpose is to make the execution and monitoring of applications uniform.

Table 2.14: *Full Cbench's Applications List (CTuning CBench suite v1.1)*

No.	cBench list	Description
1	automotive_bitcount	Bit counter
2	automotive_qsort1	Quick sort
3	automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corners
4	automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Edges
5	automotive_susan_s	Smallest Univalue Segment Assimilating Nucleus Smoothing
6	security_blowfish_d	Symmetric-key block cipher Decoder
7	security_blowfish_e	Symmetric-key block cipher Encoder
8	security_rijndael_d	AES algorithm Rijndael Decoder
9	security_rijndael_e	AES algorithm Rijndael Encoder
10	security_sha	NIST Secure Hash Algorithm
11	security_pgp_d	public key cryptography for the masses
12	security_pgp_e	public key cryptography for the masses
13	telecom_adpcm_c	Intel/dvi adpcm coder/decoder Coder
14	telecom_adpcm_d	Intel/dvi adpcm coder/decoder Decoder
15	telecom_gsm	gsm encoder/decoder
16	telecom_CRC32	32 BIT ANSI X3.66 CRC checksum files
17	consumer_jpeg_c	JPEG kernel
18	consumer_jpeg_d	JPEG kernel
19	consumer_lame	MP3 encoding engine
20	consumer_mad	MPEG audio decoder
21	consumer_tiff2bw	convert a color TIFF image to grey scale
22	consumer_tiff2rgba	convert a TIFF image to RGBA color space
23	consumer_tiffdither	convert a TIFF image to dither noisespace
24	consumer_tiffmedian	convert a color TIFF image to create a TIFF palette file
25	network_dijkstra	Dijkstra's algorithm
26	network_patricia	Patricia Trie data structure
27	office_stringsearch1	Boyer-Moore-Horspool pattern match
28	office_ghostscript	Aladdin Ghostscript
29	office_ispell	An interactive spelling corrector
30	office_rsynth	Klatt synthesizer
31	bzip2d	Burrows Wheeler compression algorithm
32	bzip2e	Burrows Wheeler compression algorithm

One of the main features of the Polybench suite is that there is a single file per application, tunable at compile-time and used for kernel instrumentation. It performs extra operations such as cache flushing before the execution, and can set real-time scheduling to prevent OS interference. We have defined two different data sets for each individual application to expose the main function with different input loads. Polybench has a variety of benchmarks, i.e. 2D and 3D matrix multiplication, vector decomposition, etc.. This suite is also suitable for parallel programming. Polybench is used in numerous recent literature and we classify them in Table 2.13 under its category.

SPEC

The System Performance Evaluation Cooperative, now named the Standard Performance Evaluation Corporation (SPEC), was founded in 1988 by a small number of workstation vendors who realized that the marketplace was in need of realistic, standardized performance tests [59]. The OSG is the original SPEC committee. This group focuses on benchmarks for desktop systems, high-end workstations and servers running open systems environments and currently they have 7 subcommittees, namely Cloud, CPU, Java, handheld, Power, SFS and Virtualization. In this survey we have aggregate all within the sole SPEC class and present them in Table 2.13.

Other Benchmarks

There are numerous other standard benchmark suites available which suitable for different domains. Here for conciseness purposes we don't classify them all, however it is worth mentioning a couple of other widely used benchmarks under this subsection.

Table 2.15: *Full Polybench's Applications List*

No.	PolyBench list	Description
1	2mm	2 Matrix Multiplications (D=A×B; E=C×D)
2	3mm	3 Matrix Multiplications (E=A×B; F=C×D; G=E×F)
3	adi	Alternating Direction Implicit solver
4	atax	Matrix Transpose and Vector Multiplication
5	bicg	BiCG Sub Kernel of BiCGStab Linear Solver
6	cholesky	Cholesky Decomposition
7	correlation	Correlation Computation
8	covariance	Covariance Computation
9	doitgen	Correlation Computation
10	durbin	Toeplitz system solver
11	dynprog	Dynamic programming (2D)
12	fdtd-2d	2-D Finite Different Time Domain Kernel
13	fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
14	gauss-filter	Gaussian Filter
15	gemm	Matrix-multiply C = aA×B + bC
16	gemver	Vector Multiplication and Matrix Addition
17	gesummv	Scalar, Vector and Matrix Multiplication
18	gramschmidt	Gram-Schmidt decomposition
19	jacobi-1D	1-D Jacobi stencil computation
20	jacobi-2D	2-D Jacobi stencil computation
21	lu	LU decomposition
22	ludcmp	LU decomposition
23	mvt	Matrix Vector Product and Transpose
24	reg-detect	2-D Image processing
25	seidel	2-D Seidel stencil computation
27	symm	Symmetric matrix-multiply
28	syr2k	Symmetric rankâĹŠ2k operations
29	syrk	Symmetric rankâĹŠk operations
30	trisolv	Triangular solver
31	trmm	Triangular matrix-multiply

Mediabench [135] is composed of full applications, not toy programs or code kernels. Each component of MediaBench is available through the Internet. Furthermore, each of these applications is coded in a high level language and has been compiled by multiple independent compilers for multiple processor architectures. Mediabench has been widely used in the recent literature [134, 146, 213].

NAS parallel benchmark suite [25] is a set of benchmarks has been developed for the performance evaluation of highly parallel supercomputers. These consist of five "parallel kernel" benchmarks and three "simulated application" benchmarks. Together they mimic the computation and data movement characteristics of large-scale computational fluid dynamics applications. NAS is used for compiler autotuning in the literature [114, 159, 179, 218, 226, 232]

2.9 Evaluations

Evaluating a research work is no easy task and involved human error. However, in this survey we tried to evaluate and present influential work of by means of their scientific breakthrough, novelty, and a competitive citation metric. We discrete the process by presenting the influential papers by their corresponding topic and elaborate more on their proposed approach. Some had effects on their succeeding work and this was taken into consideration as well.

2.9.1 Influential Papers

Breakthroughs by topic

The following paragraphs highlight novel research in the areas of: initial introduction of learning methods with compiler optimizations, genetic algorithms, phase ordering, iterative compilation, dynamic and hybrid features, and creating optimization groups with bayesian learners.

Introducing Learning Methods [138,230] was the first to perform non-iterative optimization without leverage machine learning. [233] extended this work by proposing intelligent ordering of a subset of compiler optimizations. [235] continued their phase-ordering work with a formal language, Gospel, which could be used to automatically generate transformations. The first usage of machine learning techniques arrived with [124] and their work with predicting the optimal unroll size for nested loops. [160] was the first to use machine learning techniques to construct flexible instruction schedules, paving the way for continued efforts leveraging machine learning techniques.

Genetic Algorithms [52] expanded machine learning efforts with optimization selection using genetic algorithms with iterative compilation. [53] expanded their prior work by switching to adaptive compilation and the one of the earliest works creating an adaptive compilation framework. Predictive modeling was first introduced by [227] where they applied iterative compilation of the SPEC benchmarks. [122] proposed iterative compilation to select tile and unroll factors using genetic algorithms, random sampling and simulated annealing. They were able to show that their method worked on many different architectures.

Phase Ordering [126] were one of the first to propose solving the phase-ordering problem using machine learning by combining iterative compilation and meta-heuristics. [130] tackled the phase-ordering problem within the JIKES Java virtual machine. They leveraged static features fed into a neural network generated with NEAT to construct good optimization sequence orders.

Iterative compilation [28] proposed an intelligent iterative compilation method which explored less than 2% of the total space in a non-linear search space. [7] used markov chains to focus iterative optimization using static features. Using a relatively small exhaustive search space for learning (14^5) and a large test space for testing (80^{20}), they were able to achieve up to 40% speedup.

Dynamic and Hybrid Features The first use of dynamic features for learning were introduced by [41]; they showed that using dynamic features for learning outperformed the use of static features. The advancement of multivariate (static, dynamic, hybrid) feature selection and learning algorithms paved the way for tournament predictors introduced by [179].

Optimization Groups and Bayesian Learners Massive dataset analysis on over 1000 benchmarks was performed by [46, 47]. They proposed optimization sequence groups (beyond traditional compilers' `-O3` group) that are, on average, useful to use on the appli-

Table 2.16: *A Classification of Top 15 Influential Papers by their Citation Count*

No.	Reference	Cit.	ACPY	Keywords
1	Agakov et al. [7]	317	28	iterative compilation, static features, Markov chain oracle, compiler pass prediction, PCA
2	Cooper et al. [52]	282	15	genetic algorithm, iterative compilation, reduced code-size
3	Triantafyllis et al. [227]	240	17	iterative compilation, SPEC, predictive modeling
4	Stephenson et al. [213]	234	21	iterative compilation, genetic programming, Mediabench, metaheuristics
5	Knijnenburg et al. [122]	211	15	iterative compilation, unrolling factor, architecture-independent manner
6	Cooper et al. [53]	208	13	iterative compilation, biased random search, compiler sequence predictor
7	Tournavitis et al. [226]	173	21	static-analysis, profile-driven parallelism, NAS, SPEC
8	Almagor et al. [11]	164	12	adaptive compilation, compiler sequence predictor, SPARC architecture
9	Cavazos et al. [41]	164	16	iterative compilation, dynamic characterization, PC, compiler sequence predictor
10	Tiwari et al. [225]	146	18	CHill framework, iterative compilation, compiler sequence predictor
11	Monsifrot et al. [159]	144	9	decision trees, loop-unrolling, boosting, abstract loop representation
12	Stephenson et al. [214]	138	11	supervised learning, unrolling factor, multiclass classification
13	Pan et al. [175]	126	11	combined elimination, iterative compilation, SPEC
14	Bodin et al. [28]	124	6	iterative compilation, multi-objective exploration
15	Cooper et al. [51]	95	7	adaptive compilation, metaheuristic, genetic algorithm

cations in their dataset. Most recently, [17, 20] used the output of the passes suggested by [46] to construct a bayesian network to selecting the best compiler flags using static, dynamic, and hybrid features. The bayesian network generated optimization sequences resulted in application performance outperforming preexisting models.

Citation Metric

In this section we classify the top 15 most-cited papers among the plus 100 papers we elaborated [9]. We present their citation count and the average citation per year (ACPY) with a few keywords representing their methodology we already covered in this survey. Table 2.16 depicts the classification.

2.10 Discussion & Conclusions

In this chapter, we have synthesized the research work on compiler autotuning using machine learning by showing the broad spectrum of the use of machine learning techniques and their key research ideas and applications. We surveyed research works at different levels of abstraction, viz., application characterization techniques, algorithm and machine learning models, prediction types, space exploration, target domains, etc. We discussed both major problems of compiler autotuning, namely the selection and the phase-ordering problem along with the benchmark suits proposed to evaluate them. It is hoped that this chapter will be highly beneficial to computer architects, researchers, and application developers and will inspire novel ideas and open promising research avenues.

[9] All citation data has been extracted from Google Scholar on September 2016 and they are subjected to change.

2.11 Dissemination of The Chapter

The excerpt of this survey has been submitted to ACM Transaction on Computing Surveys (CSUR) under the title *A Survey on Compiler Autotuning using Machine Learning* and is currently under review. Refer to the Chapter 7.2 for the list of publications of my PhD dissertation.

CHAPTER 3

Design Space Exploration of Compiler Passes: A Co-exploration Approach for Embedded Domain

3.1 Summary

Very Long Instruction Word (VLIW) application specific processors represent an attractive solution for embedded computing, offering significant computational power with reduced hardware complexity. However, they impose higher compiler complexity since the instructions are executed in parallel based on the static compiler schedule. Therefore, finding a promising set of compiler transformations and defining their effects have a significant impact on the overall system performance. The proposed methodology provides the designer with an integrated framework to automatically (i) generate optimized application-specific VLIW architectural configurations and (ii) analyze compiler level transformations, enabling application-specific compiler tuning over customized VLIW system architectures. We based the aforementioned analysis on a Design of Experiments (DoEs) procedure that captures in a statistical manner the higher order effects among different sets of activated compiler transformations. Applying the proposed methodology onto real-case embedded application scenarios, we show that (i) only a limited set of compiler transformations exposes high confidence level (over 95%) in affecting the performance and (ii) using them we could be able to achieve gains between (16-23)% in comparison to the default optimization levels.

3.2 Introduction

Embedded systems design traditionally exploits the knowledge of the target domain, e.g. telecommunication, multimedia, home automation etc., to customize the HW/SW coefficients found onto the deployed computing devices. Although the functionali-

ties of these devices are differed, the computational structure and design are tightly connected with the platform in which they rely on. Platform-based design has been proposed as a promising alternative for designing complex systems by redefining the problem of designing into that of finely tuning specific parameters of the platform template.

The scientific and commercial urge to use VLIW technology seems to be raising again after three decades of their existence [70]; VLIW processor templates are being used especially in embedded processors, designed to perform special-purpose functions, usually for real-time or hardware acceleration. Being able to use VLIW power-saving cores in CPUs seems to be using day by day. However, the trade-offs between right parallel execution and the speedup managed by compiler instead of hardware is becoming a very complex task. VLIW can achieve far higher performance, offering high degree of Instruction Level Parallelism (ILP) with low silicon and power costs. On the one hand, architecture configurability of VLIW platforms offers significant advantages regarding portability, sizing and parameter tuning provided to the designer [70, 72]. On the other hand, it introduces a lot of complexity during optimization due to multi-objective nature of the solution space and the multi-parametric structure of the design space.

Although a significant amount of research has been conducted on exploring and optimizing VLIW architectural parameters [16] and introducing specific compiler optimization for VLIW processors [69], [104], there are limited references regarding the analysis of the impacts of conventional compiler transformations onto VLIW architectures and moreover how these transformations are correlating with the underlying architectural configuration. Nowadays, the existence of modular and reusable compiler tool-chains LLVM and ROSE [193] raises the opportunity for system designers to exploit sophisticated compiler passes and customize their compiler infrastructure accordingly. Given the large decision space provided by the modern compiler infrastructures, the designer has to traverse to find the best trade-off points, thus a fine-grained and automatic characterization of the effects that each compiler transformation has onto the application's behaviour, is considered of great importance. Empirical evaluation of the effects, by simply activating and deactivating compiler passes cannot be considered adequate, since a lot of inter-transformation interactions and second order effects are neglected. Due to the complexity of characterizing the solution space, there is a necessity to extend conventional exploration approaches by applying sophisticated analysis and data-mining for extracting knowledge from statistical results [67]. The problem becomes more demanding in the embedded computing domain, which requires different optimizations related to each platform configuration customized for a specific application domain. The main contribution of this chapter consists of proposing a compiler/architecture methodology that provides to the designer an integrated environment to automatically (i) generate optimized application specific architectural configurations of VLIW-based platforms and (ii) analyse onto them the effects of compiler level transformations in a statistical manner.

The proposed methodology targets the design problem of compiler/architecture co-exploration found in embedded computing, and it is focused on enabling application-specific compiler tuning over customized VLIW system architectures. First, a multi-objective exploration loop targeting application-specific micro-architectural customiza-

tion is applied for extracting the best VLIW architecture candidates. We utilize the newly introduced Roof-Line processor architecture model [237] for characterizing the differing architectural solutions onto various resource constraints. The optimized VLIW architectural configurations are then propagated to the compiler analysis phase in which the statistical effects of the applied compiler transformations are characterized in a fine grained manner. The developed exploration framework integrates the LLVM compiler infrastructure [142] as a source to source code transformation tool together with the VEX compiler-simulator for mapping the transformed code onto custom VLIW architecture instances. We evaluated the overall methodology (customized architecture selection and statistical compiler level analysis) using a GSM codec application as the driving use case. We show that only a limited set of compiler transformations has significant effect on optimizing performance across a set of GSM specific VLIW processors. In addition to the application specific scenario, we present results regarding the multiple embedded applications onto a single VLIW instance, showing that the proposed analysis can be used to extract promising compiler transformations regarding both an intra- and cross-application manner.

The rest of the chapter is organized as follows. Section 3.3 provides a brief discussion on related work and current state of the art in the field. In Section 3.4, we introduce the basic methodology for architecture customization and statistical compiler level analysis. Section 3.4.2 presents experimental evaluation of the proposed methodology on differing customized VLIW architectures and benchmark applications. Section 3.5 summarizes of the work and concludes the chapter.

3.3 Background

Although, we have entered the era of multi-core systems, the high degree of instruction parallelism offered by VLIW architectures seems to make them an interesting alternative for a large set of commercial embedded systems [70], [197], [65]. VLIW architectures are also emerging in the modern many-core embedded accelerator devices, i.e. KALRAY MPPA256 [3], for image and signal processing applications.

Several research works have been presented targeting to the generation of Pareto optimal VLIW architectural configurations [16], [197] by exploring the space using preallocated compiler sequences over differing architecture instances. Towards the same direction of VLIW architectural configuration, Wong et al [240] introduced r-VEX, a reconfigurable and extensible VLIW processor. Source code is mapped using the VEX (VLIW Example) environment [1], which forms a compilation-simulation system that targets a wide class of VLIW processor architectures, and enables compiling, simulating, analyzing and evaluating C programs [72].

In current literature, there is a lot of attention on iterative compilation and predictive compiler modeling to predict the potential speedup of compiler transformed programs utilizing code features provided by static program analysis as mentioned in the Chapter 2. However, there is a lack of comprehensive analysis regarding the impact of applying differing conventional compiler transformations on customized VLIW architectures. Although, in VLIW compilation infrastructures [1] there are available batch compiler optimization modes, fine-grained analysis of compiler effects for VLIW architectures and its relation with architecture customization is not adequately targeted.

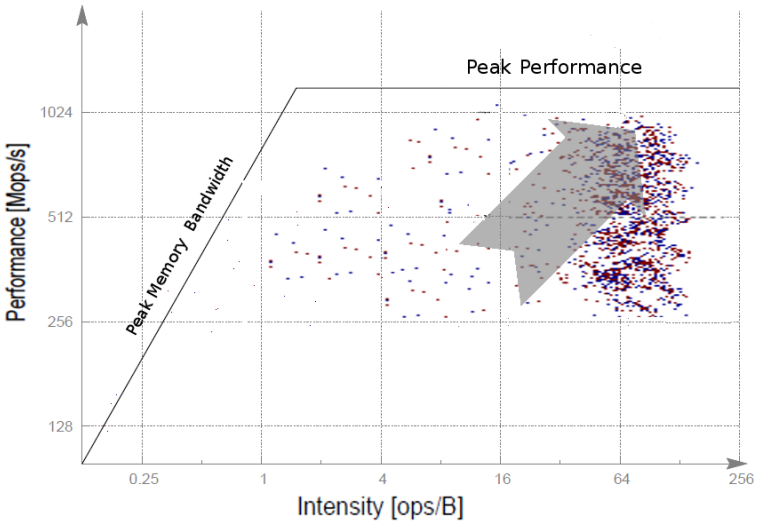


Figure 3.1: *Roof-Line example*

3.4 Methodology for Compiler Analysis of Customized VLIW Architectures

In this section, we describe the proposed methodology for compiler analysis of customized VLIW architectures. The proposed methodology consists of two phases: (i) Customized VLIW architecture selection and (ii) Statistical analysis of compiler transformations. From a high level point of view, we first generate a set of promising VLIW architectural candidates that tailors to the characteristics of the target application, optimizing on the performance-intensity trade-off curve with respect to the overall hardware allocated resource. Then, statistical analysis of distributions generated over the compiler transformation space is performed on the set of these selected customized VLIW solutions. This enables the designer to characterize the effects of each compiler transformation in both an architecture specific manner and a cross-architecture manner.

We used the Roof-Line performance model [237] as the basis for both generating the custom architecture configurations and characterizing the effect of the compiler passes. Roof-Line relates processor performance to off-chip memory traffic. It characterizes processor architectures in a two-dimensional space, i.e. performance (Mops/sec) vs. operational intensity (ops/Byte). *Operational intensity* is defined as operations per byte of DRAM traffic, defining total byte accessed as those bytes that go to the main memory after been filtered by the cache hierarchy. The advantage of using Roof-Line model is twofold: (i) it provides the designer with an intuitive insight visual metric for fast evaluation of the architectural optimality of the configuration and (ii) it is useful to characterize the impact of applied compiler transformations onto a specific architecture. For example, Figure 3.1 presents the Roof-Line model of a specific VLIW configuration and the superposition of application configurations derived by an experimental campaign of 4K different compiler parameter combinations. A general trend (highlighted by the arrow in Figure 3.1) can be easily detected towards higher performance and operational intensity points. Given this visual representation a designer can

detect promising compiler passes to be applied. A custom exploration and analysis framework (Figure 3.2) has been developed based on the integration of open source tools to implement the proposed methodology. Specifically, we used Multicube Explorer [2, 243] as the central DSE engine. Given the architectural and compiler design space descriptions, it manages to automatically generate configuration vectors according to the specified DoE – random DoE during the phase of custom VLIW architecture selection and random effect DoE during the compiler transformation analysis phase. The LLVM compiler infrastructure [1] is integrated within the framework – specifically the LLVM C front-end and the opt tool – as a source to source transformation tool of the original application code after applying the compiler transformations instructed by the DSE engine. The transformed code of the application is mapped onto the VLIW processor using the VEX [1] VLIW compiler-simulator tool, which is used for both generating different VLIW architectural configurations and mapping code onto these custom VLIW processors. Custom scripts have been developed to evaluate each examined configuration according to the Roof-Line model. Statistical analysis and visualization of results are performed using the R statistical language R [221].

3.4.1 Custom VLIW Architecture Selection

Application-specific customization of architecture's parameters is one of the early system design optimization phases for defining platform configurations that meet the desired performance specifications. Given the large number of parameters that usually defines a processor architecture and the delay required for simulating each possible

[1] LLVM projected supported its C source-to-source compiler frontend till v2.8

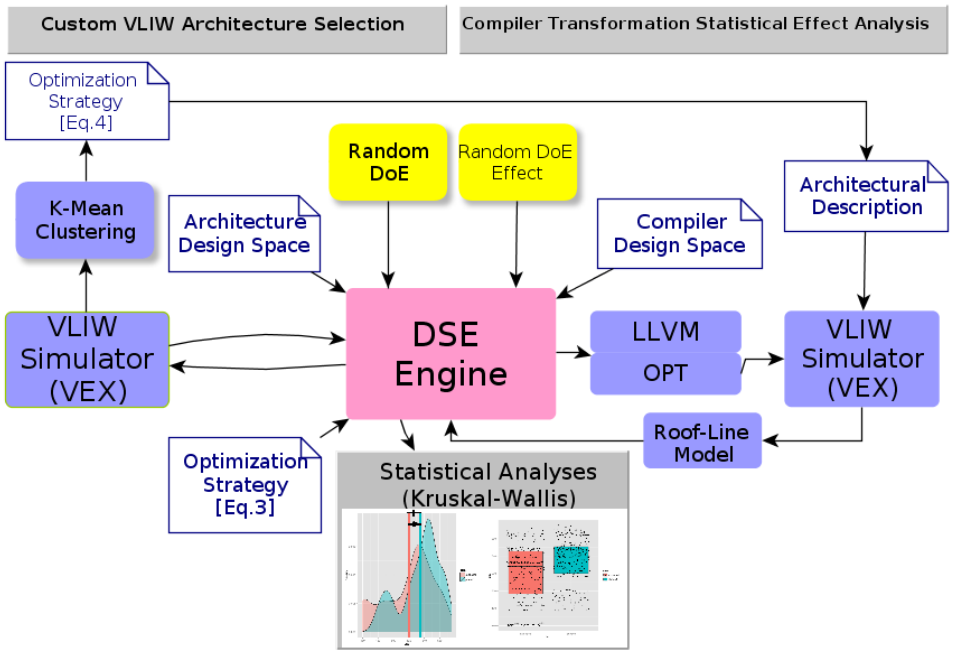


Figure 3.2: *Tool-chain implementing the proposed methodology*

configuration, the task of optimal micro-architectural parameter selection forms an extremely challenging exploration problem that for reasonably representative design space definitions becomes intractable, regarding the time required for exhaustive evaluation. Several research works utilizing well-known meta-heuristics [16, 172] have been already proposed for generating the Pareto optimal sets of the aforementioned optimization problem.

In this chapter, however, we slightly shift the focus of exploration from delivering the optimal set of architectural configurations to discover custom architecture configurations that do not correspond to the boundaries of Pareto regions, i.e. very low cost architectures with very poor performance or very expensive architectures that deliver very high gains regarding performance. Thus, in this chapter we invoke a relaxed optimization search strategy that is based on a random sampling of the targeted design space rather than on a optimization oriented strategy, e.g. simulated annealing or NSGA-II genetic optimization [172] etc.

Table 3.1 shows the micro-architectural design space, Ω, considered for the custom VLIW architecture selection phase. In the first step, we randomly sample the Ω design space. Each explored solution is stored in the database of explored solutions, X after being characterized according to the performance and operational intensity metrics defined within the Roof-Line model, where:

$$Performance(\mathbf{x}) = \frac{\#Operations(\mathbf{x})}{\#NumCycles(\mathbf{x}) \times ClkFreq(\mathbf{x})} \tag{3.1}$$

$$Intensity(\mathbf{x}) = \frac{\#Operations(\mathbf{x})}{\#CacheMisses(\mathbf{x}) \times CacheLineSize(\mathbf{x})} \tag{3.2}$$

After the formation of the X, we are interesting in finding those explored architectures that maximize the performance and operational intensity of the application while using minimum computational and memory resources. In order to extract the desired architectural configurations, we perform Pareto filtering on the solution space defined with the X, by considering the following multi-objective optimization problem:

$$\min_{\mathbf{x}\in\Omega} \begin{bmatrix} \frac{1}{Performance(\mathbf{x})} \\ \frac{1}{Intensity(\mathbf{x})} \\ \#CompResources(\mathbf{x}) \\ \#MemResources(\mathbf{x}) \end{bmatrix} \tag{3.3}$$

where computational resources are (i) number of ALUs and (ii) number of multipliers, while memory resources are (i) data cache size, (ii) instruction cache size and (iii) register file size. Although, in Eq. 3.3 we present the unconstrained version of the target optimization problem, we note that our exploration infrastructure permits also the inclusion of arbitrary constraints either on the objectives itself or on specific parameter combinations that the designer has a-priori evaluated as not interesting.

The outcome of the optimization procedure defined in Eq. 3.3 is a Pareto surface, X_p, of the explored X, thus exhibiting a large number of VLIW architectural configurations. In order to restrict the number of VLIW configuration that will be characterized as the representative customized VLIW solutions that will be propagated to the statistical compiler analysis phase, we perform a clustering on the performance - intensity solution space. We used k-means [115] clustering for the aforementioned procedure,

Table 3.1: *VLIW MIcroarchitectural Design Space*

Parameters	Values (Integer Range)
lg2CacheSize	[11-30]
lg2Sets	[0,3]
lg2LineSize	[5,9]
lg2ICacheSize	[11,30]
lg2ICacheSets	[0,3]
lg2ICacheLines	[5,9]
ClkFreq	[300,500]
NumCaches	[1,2]
IssueWidth	[1,16]
NumAlus	[1,16]
NumMuls	[1,4]
RegisterFile	[32,128]
BranchRegister	[32,128]

with a configurable number of clusters, k, decided by the designer. The clustering procedure partitions the X_p solution space into k regions of interest, $X_p^{c_i}$, e.g. region of high intensity and high performance, or region of low intensity and high performance etc. Eventually, each cluster should deliver one representative VLIW architecture, that forms the optimal solution within the cluster. We define this optimal solution per cluster as the architectural configuration that minimizes area cost of the processor while maximizing both the metrics of performance and operational intensity. In order to extract this optimal configuration from within each cluster, we iteratively apply the following single-objective minimization problem in every $X_p^{c_i}$ produced by the k-means clustering:

$$\min_{\mathbf{x} \in X_p^{c_i}} \quad \frac{Area(\mathbf{x})}{Performance(\mathbf{x}) \times Intensity(\mathbf{x})} \tag{3.4}$$

For the calculation of the area cost in Eq. 3.4, the area model provided by the McPAT [140] micro-architecture framework has been used, assuming an process technology of 90 nm.

Deriving to an architectures which is optimized by using right set of compiler optimizations is an essential task to mitigate. However, reaching this goal has its own tolerance and trade-off. Occasionally it happens to sacrifice the code size for better performance or portability versus code size. Consequently, there should be a precaution when using these options otherwise it ends up heavier and less-usable. Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. The compiler performs optimization based on the knowledge it has of the program. Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them. Not all optimizations are controlled directly by a flag. In this research work the 15 selected compiler passes supported by LLVM compiler are as described in the table 6.2.

Table 3.2: *Selected Compiler Transformations From LLVM Framework*

Compiler Transformation	Abbreviation	Short Description
Constant Propagation	constprop	Instructions involving only constant operands are replaced with a constant value and propagated
Dead Code Elimination	dce	It checks instructions that were used by removed instructions to see if they are newly dead
Function Integration/Inlining	inline	Bottom-up inlining of functions into callees
Combine Redundant Instruction	instcombine	Combine instructions to form fewer, simple instructions. This pass does not modify the CFG and is where algebraic simplification happens
Loop Invariant Code Motion	licm	Attempting to remove as much code from the body of a loop as possible. It does this by either hoisting code into the pre-header block, or by sinking code to the exit blocks if it is safe
Loop Strength Reduction	loop-reduce	It performs a strength reduction on array references inside loops that have as one or more of their components the loop induction variable
Rotates Loops	loop-rotate	A simple loop rotation transformation
Unroll Loops	loop-unroll	This pass implements a simple loop unroller
Unswitch Loops	loop-unswitch	This pass transforms loops that contain branches on loop-invariant conditions to have multiple loops
Promote Memory To Register	mem2reg	It promotes memory references to be register references.
Memorycopy Optimizations	memcpyopt	It performs various transformations related to eliminating memcpy calls, or transforming sets of stores into memset's
Reassociate Expressions	reassociate	It reassociates commutative expressions in an order that is designed to promote better constant propagation
Scalar Replacement of Aggregates	scalarrepl	It breaks up alloca instructions of aggregate type (structure or array) into individual alloca instructions for each member if possible
Sparse Conditional Constant Propagation	sccp	It assumes values are constant and Basic Blocks are dead unless proven otherwise. It proves values to be constant, and replaces them with constants and Proves conditional branches to be unconditional
Simplify the Control Flow Graph	simplycfg	Performs dead code elimination and basic block merging

DoE

Having faced a huge multi-objective optimization problem, there is a necessity for using such methods like Taguchi Design of experiment [194]. DoEs are the basic components for building the exploration strategies. The DoE used in this work was based on Random factors which generated a set of random designed points. In addition, the optimization algorithm used here was parallel DoE (PDoE) which was based on the possibility of performing concurrent evaluation of the different design points; i,e, in the experimental analyses, for each compiler transformations per benchmark, the number of exploration was 500, therefore, it would have given enough points for the system to use for DoE and Optimizer to generates the effects and metrics beside the Pareto points (if exists).

3.4.2 Compiler Transformation Statistical Effect Analysis

This second phase of the proposed methodology receives as input the generated custom VLIW architectures as described in the previous section, and for each of the set of micro-architectural points it evaluates the statistical effects of the compiler transformations in a fine grained manner. In this research work we focus on 15 of the compiler passes supported by LLVM (see Table 6.2).

As a first step in our analysis, we have to determine a reasonable amount of samples to produce a robust analysis of the main effects associated with the 15 compiler parameters. In the following, each configuration of these compiler parameters, or set of compiler options, will be defined as a vector of 15 values, where each value represents a compiler pass option.

To accommodate our goal, we defined a randomized design of experiments $D_N(p)$ for each compiler parameter p. $D_N(p)$ is a list of *options sets*:

$$D(p) = [o_{1+}, o_{1-}, o_{2+}, o_{2-}, ..., o_{N+}, o_{N-}] \tag{3.5}$$

where o_{n+} corresponds to the n-th random option set in which compiler pass $p \in \{OFF, ON\}$ is set to its maximum value (ON) while all the others compiler passes are randomly chosen. In a dual way, o_{n-} is equal to o_{n+} except that p assumes its minimum value (OFF).

By applying this DoE, we can easily measure how much the impact of the transition ($- \rightarrow +$) for parameter p impacts (in average over all the considered options sets on the performance without requiring a full-factorial design. As an example, Figure 3.3 depicts the generated performance distributions by activating and deactivating the 'licm' and 'reassociate' compiler transformations for a GSM codec application. It can be observed that while the activation of 'licm' has a clear positive effect on performance – the median is shifted towards higher performance values, this is not the case for the 'reassociate' transformation for which the activation and deactivation distributions have almost the same shape and density, thus not permitting the designer to recognize a clear trend.

As the second step, for each options set in $D(p)$ we evaluate the vector of performance responses with the actual architecture synthesis after the compilation and simulation of the target application. We consider the hypothesis whether the mean of the performance given by the options sets where p was minimum (or *off*) is *different* from the mean where p was maximum (or *on*). In practice, this is framed as a *null-hypothesis statistical test*, which, given the *non-parametric* (or non-gaussian) nature of the underlying distributions[2], cannot be assessed with as a simple ANOVA but, instead, with a Kruskal-Wallis test [33]. To complete the hypothesis test, the designer sets an acceptance ratio of $p - value\%$ meaning that the probability of 'measuring' different means when the underlying distributions are equal (or the chance of a false positive) is less than 5%.

Statistical Analysis

As mentioned in the section 3.3, there has been several works involving the machine learning techniques and predictions [7, 40, 60]. In this research work we have focused

[2]Since the distributions are built based on empirical/experimental data, the distribution is considered in general non-parametric

on analyzing the effects of applying the specific compiler transformations on the design space. The probability of certainty about the effects of a specific compiler transformation on performance metric could be done using some statistical tests; ANOVA, Kruskal-Wallis. ANOVA [223] test has been widely used as a reliable tester for normal distributions. In addition, using Kruskal-Wallis [33], is a good test tool as it assumes the distribution to be non-parametric. This method is used for testing whether samples originated from the same distribution or not. In this work, since dealing with empirical data on experimental results, we assumed the models as non-parametric, therefore, Kruskal-wallis was employed. The algorithm goes as:

- 1- Rank all the groups from 1 to N together
- 2- Statistical test is elaborated among the group to calculate the value K which contains the square of the average ranks
- 3- Finally the *p-value* is approximated as $Pr(\chi^2_{g-1} \geq K)$
- 4- If the statistic is not significant, then there is no real evidence of difference between samples and could be deduced the samples are comply with the model.

In this work, the global threshold was set as high as 5% in order to increase the robustness of the results. Therefore, a test is deduced as passed regarding Kruskal-wallis test in which it has the p-value smaller than 0.05. In this case, a model is passed if and only if it had confidence threshold over 95%; experimental analyses represented in Figure 3.5 will be scrutinized later in this chapter.

In this section, we experimentally evaluate the proposed methodology. We consider the GSM codec embedded application as the driving use case, automatically generating four representative application specific architectures after applying the custom VLIW architecture selection. We use these VLIW architectures for statistically analysing the effects of compiler transformations across differing VLIW configurations. Furthermore we analyse the compiler transformation effects in a cross application manner, by considering a larger set of embedded applications mapped onto a default (non application specific) VLIW processor configuration.

The first subsection, introduce the experimental setup and the framework. the second subsection will contain the architectural selection based on the method described in section 3.4.1 and exploration on standard benchmark regarding the derived configurations will be presented. Eventually, there will be a comparison of the default architecture among 5 other benchmarks will be discussed and depicted with the statistical consolidations.

We apply the overall proposed methodology considering the GSM codec as the driving application. We apply the custom VLIW architecture selection phase to generate optimized representative VLIW architectures in an application specific manner. The considered architectural design space is depicted in Table 3.1. We configure the search procedure to randomly generate and evaluate 30K configurations, using a uniform sampling over the targeted configuration space (Table 3.1). Applying the multi-objective optimization problem defined in Eq. 3.3 over the 30K solutions, the Pareto surface of the configurations that maximize performance and operational intensity while minimizing resources is generated. Without loss of generality, we consider the generation

Table 3.3: *VLIW Architecture Configurations*

Parameters	Arch-HL	Arch-LH	Arch-HH	Arch-LL	Arch-User
lg2CacheSize	15	12	13	12	16
lg2Sets	1	3	0	1	2
lg2LineSize	7	5	5	5	5
lg2ICacheSize	16	14	16	14	16
lg2ICacheSets	1	3	3	2	2
lg2ICacheLines	6	8	7	5	6
ClkFreq	400	450	450	300	500
NumCaches	2	1	1	1	1
IssueWidth	6	6	14	9	8
NumAlus	4	6	7	3	8
NumMuls	1	4	4	14	2
MemLoad	4	3	6	5	4
MemStore	2	8	4	6	4
RegisterFile	104	100	32	76	64
BranchRegister	76	84	88	48	64

of k=4 clusters over the generated Pareto surface, aiming at the generation four GSM-specific VLIW architectures. Figure 3.4 shows results of clustering of the extracted Pareto surface and its mapping onto the two-dimensional performance vs. intensity space. Each cluster has been characterized according to its position on the performance vs. intensity space as: (i) HH for the cluster placed to the high intensity and high performance region, (ii) LH for the low intensity and high performance region, (iii) LL for the low intensity and low performance region and (iv) HL for the high intensity and low performance region, respectively.

The final k=4 representative VLIW architectures are derived after applying within each cluster the optimization operator of Eq. 3.4. Table 3.3 reports the architectural configuration for each of the k=4 application specific VLIW architectures.

For each of the k=4 application specific VLIW architectures, we explore the compiler level design space, defined in Table 6.2. We generate the non-parametric distribution of the performance and intensity for each compiler transformation considering 500 samples per transformation. As described in section 3.4.2, the non-parametric distributions are analysed based on Kruskal-Wallis test to specify the statistical effect, i.e. if the inclusion or exclusion of a specific transformation impacts in a specific and robust manner the two considered metrics. Table 3.4 summarizes the results of Kruskal-Wallis statistical tests for each compiler transformation over the four examined architecture configurations. As shown, four compiler passes (*inline, licm, loop-reduce and loop-rotate*), over the fifteen initially considered, have a clear and significant impact on performance when activated. In addition, Figure 3.5, shows the confidence level for each of the considered compiler transformations. It is shown that the four mentioned compiler transformations exhibit a high confidence level $>99\%$. Therefore, it could be implied that activating these specific transformations, the designer can be around 99% confident that the effect on performance will be the same as the one determined by the exploration.

In the second set of experiments, we perform statistical analysis in a cross-application manner. For this experimental campaign, we assume a larger set of applications (namely GSM, AES encryption engine, ADPCM codec, JPEG decoder and Blowfish block ci-

Table 3.4: *Summary of Kruskal-Wallis Analysis on Performance for GSM-specific VLIW architectures*

CompilerTransformation	Arch-HL	Arch-LH	Arch-HH	Arch-LL
Constprop	-	-	-	-
Dce	-	-	-	-
Inline	√	√	√	√
Instcombine	-	-	-	-
Licm	√	√	√	√
Loop reduce	√	√	√	√
Loop rotate	√	√	√	√
Loop unroll	-	-	-	-
Loop unswitch	-	-	-	-
Mem2reg	-	-	-	-
Memcpyopt	-	-	-	-
Reassociate	-	√	√	√
Scalarrepl	-	-	-	-
Sccp	-	-	-	-
Simplyfycfg	-	-	-	-

pher). The performance of aforementioned applications has been evaluated considering a user specified VLIW architecture, Arch-User, defined in the last column of Table 3.3. For each benchmark the compiler transformation statistical effect analysis (section 3.4.2) is applied, considering distributions of 500 samples per compiler transformation. Table 3.5 summarizes in an aggregated manner the results of the Kruskal-Wallis analysis considering in each case a confidense leve $\geq$5%. For the specific setup, we observe that there is a set of four compiler parameters (*licm, loop reduce, loop rotate and mem2reg*), with significant effect on performance and with a high confidence level over all the examined application use cases. Furthermore, examining each application in isolation, the designer can derive which are the compiler parameters that need to be pre-allocated, thus reducing significantly the design-time required to optimize the performance of the targeted application during iterative compilation exploration. As an example, we depict in the Figure 3.6, the normalized speedup gains achieved by activating the compiler transformations proposed by our methodology in comparison with several well-known compilation strategies. It is shown that the proposed methodology defined speedup gains in all the examined cases between 16-23%.

3.5 Conclusions and Future Work

This chapter presents a new customization and analysis methodology for compiler/architecture co-exploration of VLIW platform design. The proposed methodology provides the designer with an integrated environment to automatically (i) generate optimized application specific VLIW architectural configurations and (ii) analyze in a fine-grained manner the effects of compiler level transformations regarding the performance and operational intensity trade-offs. Being focused more on the analysis part, we showed that the adoption of the specific methodology either in a cross-architecture and/or cross-application manner, can deliver significant application specific insights thus enabling the designer to guide through decisions regarding the architecture and the compilation optimization strategy. Future work is aligned with our strong belief that the proposed methodology can be exploited in a straightforward manner within automated

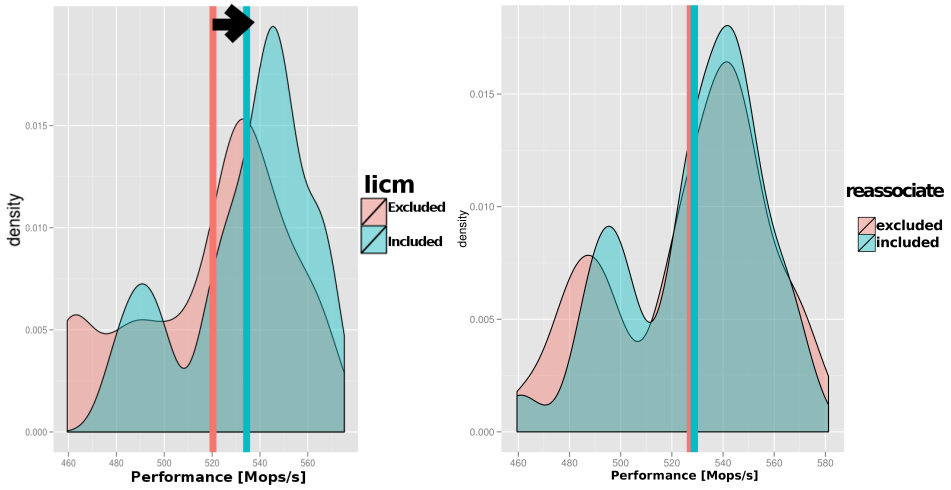
Table 3.5: *Kruskal-Wallis Analysis on Performance for Multiple Applications*

CompilerTransformation	GSM	AES	ADPCM	JPEG	Blowfish
Constprop	-	-	-	-	-
Dce	-	-	-	-	-
Inline	√	-	√	√	-
Instcombine	√	-	√	√	√
Licm	√	√	√	√	√
Loop reduce	√	√	√	√	√
Loop rotate	√	√	√	√	-
Loop unroll	-	-	-	-	-
Loop unswitch	-	-	-	-	-
Mem2reg	√	√	√	√	√
Memcpyopt	-	-	-	-	-
Reassociate	√	-	-	-	-
Scalarrepl	√	-	-	-	√
Sccp	-	-	-	-	-
Simplyfycfg	-	-	-	-	-

design frameworks focusing on performance optimization though iterative compilation and architecture specialization.

3.6 Dissemination of The Chapter

The excerpt of this chapter has been published in IEEE - VLSI-SoC 2013 held Istanbul, TURKEY, p124-129 under the title *A Framework For Compiler Level Statistical Analysis over Customized VLIW Architecture*. Refer to the Chapter 7.2 for the list of publications of my PhD dissertation.



(a) *licm* **(b)** *reassociate*

Figure 3.3: *Visualization of (a) licm's significant positive effect, (b) reassociate's no significant effect.*

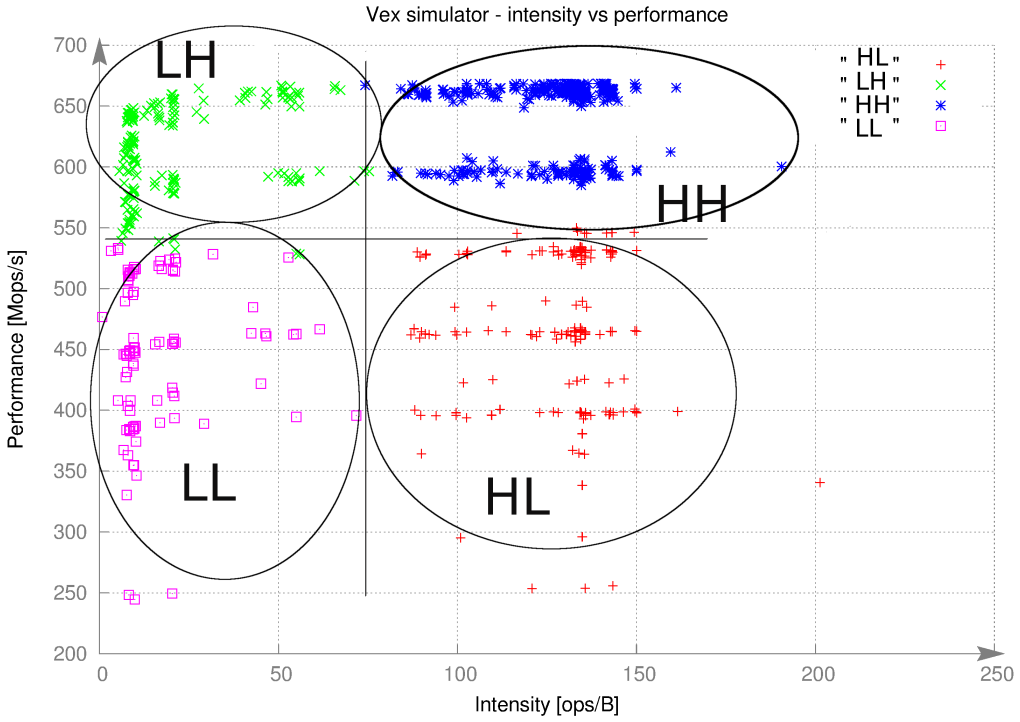


Figure 3.4: *Four Clustered Pareto-sets*

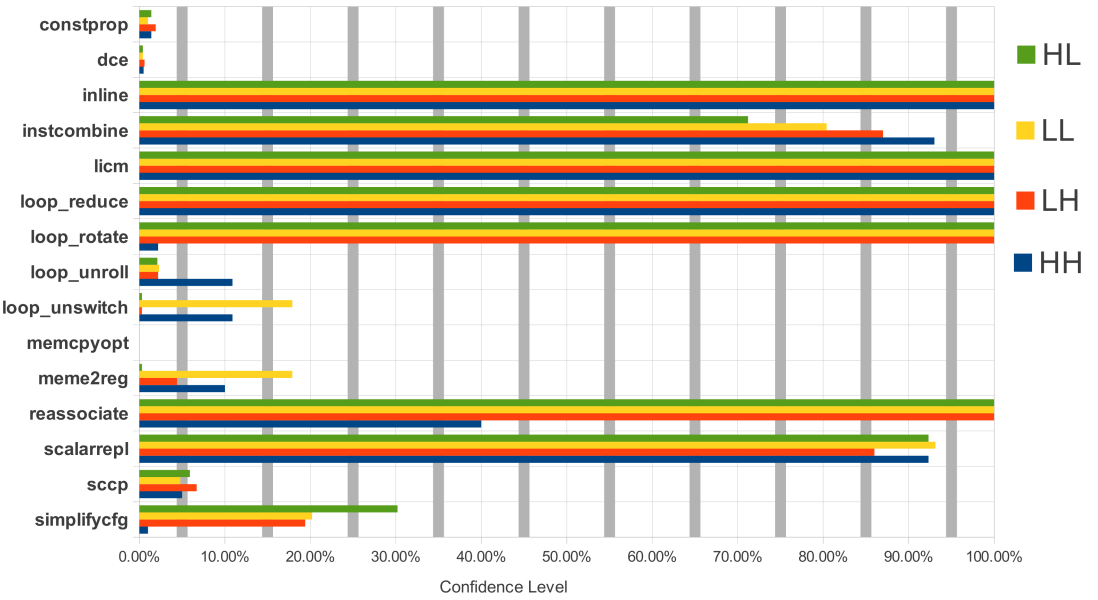


Figure 3.5: *Confidence level characterization of compiler transformations regarding the effect on performance for each on of the GSM specific VLIW architectures, resulted after Kruskal-Wallis statistical test*

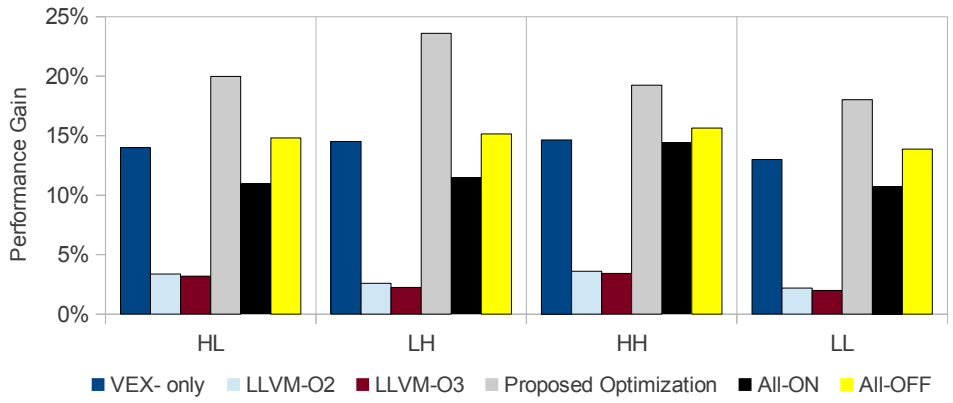


Figure 3.6: *The gained speed-up we gained comparing to the default LLVM-O1 optimization level in GSM benchmark*

CHAPTER 4

Selecting the Best Compiler Optimizations: A Bayesian Network Approach

4.1 Summary

The variety of today's architectures forces programmers to spend efforts for porting and tuning application codes across different platforms. Compilers themselves need additional tuning which has considerable complexity as the standard optimization levels, usually designed for the average case and the specific target architecture, quite often fail to bring the best results.

This chapter proposes *COBAYN*: COmpiler autotuning framework using BAYesian Networks, an approach for a compiler autotuning methodology using machine learning to speed up application performance and to reduce the cost of the compiler optimization phases. The proposed framework is based on the application characterization done dynamically by using independent micro-architecture features and Bayesian networks. The chapter also presents an evaluation based on using static analysis and hybrid feature collection approaches. In addition, the chapter compares Bayesian networks with respect to several state-of-the-art machine-learning models.

Experiments were carried out on an ARM embedded platform and GCC compiler by considering two benchmark suites with 39 applications. The set of compiler configurations selected by the model (less than 7% of the search space), demonstrated an application performance speedup of up to 4.6× on Polybench (1.85× on average) and 3.1× on cBench (1.54× on average) with respect to standard optimization levels. Moreover, the comparison of the proposed technique with (i) random iterative compilation, (ii) machine learning-based iterative compilation and (iii) non-iterative predictive modeling techniques, shows on average, 1.2×, 1.37× and 1.48× speedup, respectively. Finally, the proposed method demonstrates 4× and 3× speedup, respectively on cBench and

Polybench, in terms of exploration efficiency given the same quality of the solutions generated by the random iterative compilation model.

4.2 Introduction

Usually, software applications are developed in a high-level programming language (e.g. C, C++) and then passed through the compilation phase to get the executable binary. Optimizing the second phase (*compiler optimization*) plays an important role for the performance metrics. In other words, enabling compiler optimization parameters (e.g. loop unrolling, register allocation, etc.) might lead to substantial benefits in several performance metrics. Depending on the strategy, these performance metrics could be *execution time, code size* or *power consumption*. A holistic exploration approach to trade-off these metrics also represents a challenging problem [173].

Application developers usually rely on compiler intelligence for software optimization, but they are unaware of *how* the compiler itself does the job. Compiler interface usually has some standard optimization levels which enable the user to automatically include a set of predefined optimization sequences for the compilation process [100]. These standard optimizations (e.g. -O1, -O2, -O3 or -Os) are known to be beneficial for performance (or code size) in most cases. In addition to the above-mentioned standard optimizations, there are other compiler optimizations which are not included in the predefined optimization levels. Their effects on the software are quite complex and mostly depend on the features of the target application. Therefore, it is rather hard to decide whether to enable specific compiler optimizations on the target code. Considering application-specific embedded systems, the compiler optimization task becomes even more crucial because the application is compiled once and then deployed on millions of devices on the market.
So far, researchers proposed two main approaches for tackling the problem of identifying the best compiler optimizations: i) *iterative compilation* [46] and ii) *machine-learning predictive modeling* [7, 17, 20]. The former approach relies on several recompilation phases and then selecting the best set of optimizations. Obviously this approach, although effective, has high overhead as it needs to be evaluated iteratively. The latter approach focuses on building machine-learning predictive models to predict the best set of compiler optimizations. It relies on software features that are collected either *offline* or *online*. Once the model has been trained, given a target application, it can predict a sequence of compiler optimization options to maximize performance. Machine learning approaches need fewer compilation try-outs, but the downside is typically represented by the performance of the final execution binary, which is worse than the one found with iterative compilation.

In this chapter, we propose an approach to tackle the problem of identifying the compiler optimizations that maximize the performance of a target application. Differently from previous approaches, the proposed work starts by applying a statistical methodology to infer the probability distribution of the compiler optimizations to be enabled. Then, we start to drive the iterative compilation process by sampling from this probability distribution. We use two major sets of training application suites to learn the statistical relations between application features and compiler optimizations. To

the best of our knowledge, in this work, *Bayesian Networks* (BN) are used for the first time in this field to build the statistical model. Given a new application, its features are fed into the machine-learning algorithm as *evidence* on the distribution. This evidence imposes a bias on the distribution, and because compiler optimizations are correlated with the software features, we can iteratively sample the distribution obtaining the most promising compiler optimizations, by then exploiting an *iterative compilation* process.

The experiments carried out on an embedded ARM-based platform outperformed both standard optimization levels and the state-of-the-art iterative and not iterative (based on prediction models) compilation techniques, while using the same number of evaluations. Moreover, the proposed techniques demonstrated significant exploration efficiency improvement of up to 4× speedup compared with random iterative compilation when targeting the same performance. To summarize, our work contributes to the following:

- The introduction of a BN capable of capturing the correlation between the application features and the compiler optimizations. This enables us to represent the relation by an acyclic graph, which can be easily analyzed graphically.
- The integration of the BN model in a compiler optimization framework. Given a new program, the probability distribution of the best compiler optimizations can be inferred by means of BN to focus on the optimization itself.
- The integration of both dynamic and static analysis feature collections in the framework as hybrid features.

Furthermore, the experimental evaluation section reports the assessment of the proposed methodology on an embedded ARM-based platform and the comparison of the proposed methodology with several state-of-the-art machine learning algorithms on 39 different benchmark applications.

The remainder of the chapter is organized as follows. Section 4.3 presents a quick review of recent related literature on the very topic. Readers are referred to Chapter 2 for a holistic review. Section 4.4 presents how the BN model can infer the probability of the distribution. Section 4.4.1 presents different techniques for collecting program features. Section 4.5 elaborates on the proposed framework. Sections 4.5.4 and 4.5.5 will introduce the results obtained on the application suites selected. Finally Section 4.5.6 presents the comparison of the proposed methodology with state-of-the-art models.

4.3 Previous work

Optimizations carried out at compilation have been broadly used, mainly in embedded computing applications. We address the holistic review on the survey in the Chapter 2. However, for completeness we point out a few related work to re-iterate on the topic. This makes such techniques especially interesting, and researchers are investigating more efficient techniques for identifying the best compiler optimizations to be applied given the target architecture. There are two major classes of optimization in the field of compiler: (i) The problem of *selecting the best compiler optimizations* and (ii) The *phase-ordering* problem of compiler optimizations. As the target of this work is in the scope of *selection*, here we mostly refer to these areas. However, there are notable works to be mentioned that support the seminal concepts of the current work.

The related work in this field can be categorized into two sub-classes: *(a) iterative compilation* [28] and *(b) machine-learning* based approaches [52, 120]. Nonetheless, these two approaches have also been combined in many ways [7] that they cannot be distinguished easily.

Iterative compilation was introduced as a technique capable of outperforming static handcrafted optimization sequences, those usually exposed by compiler interfaces as *optimization levels*. Since its introduction [28, 121], the goal of iterative compilation has been to identify the most appropriate compiler passes for a target application.

design space exploration for VLIW architectures [22]. The intuition was that the performance of a computer architecture depends on the executable binary which in turn, depends on the optimizations applied at compilation time. Thus, by studying the two problems jointly, the final architecture is optimal in terms of the compilation technique in use and the effects of different compiler optimizations are identified at the early design stages. and work balancing.

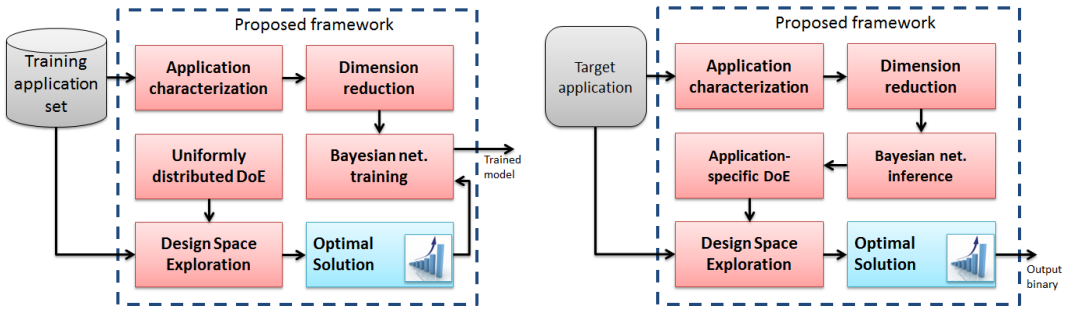
Given that compilation is a time-consuming task, several groups proposed techniques to predict the best compiler optimization sequences rather than applying a trial-and-error process, such as in iterative compilation. These prediction methodologies are generally based on *machine-learning* techniques [7, 20, 41, 52, 213].

More recent literature on using different program features with machine learning have been proposed by [177] where the authors collected *IR (intermediate representation)* of the kernels and utilized *graph-kernels* to derived the similarities between those fetched *IRs*.

Our approach is significantly different from the previous ones given that it applies a statistical methodology to learn the relationships between application features and compiler optimizations as well as between different compiler optimizations where *machine-learning* techniques are used to capture the probability distribution of different compiler transformations. In this work, we propose the use of *BN* as a framework enabling statistical inference on the probability distribution given the evidence of application features. Given a target application, its features are fed to *Bayesian Networks* to induce an application-specific bias on the probability distribution of compiler optimizations.

Most recent machine-learning works aim at the generation of prediction models that, given a target application, predict the performance of the application for any set of compiler transformations applied to it. In contrast, in our work the machine-learning methodology aims directly at predicting the best compiler optimizations to be applied for a target application without going through the predictions of the resulting application performance.

Additionally, in our approach, program features are dynamic and obtained through micro-architecture-independent characterization [102] and compared with the results using the static profiling [82]. The adoption of dynamic profiling provides insight into the actual program execution with the purpose of giving more weight to the code segments executed more often (i.e. code segments whose optimization would lead to higher benefits according to Amdahl's law).



(a) *Training the Bayesian network* **(b)** *Optimization process for a new target application*

Figure 4.1: *Overview of the proposed methodology.*

4.4 Proposed Methodology

The main goal of the proposed approach is to identify the best compiler optimizations to be applied to a target application. Each application is passed through a characterization phase that generates a parametric representation of the application under analysis in terms of its main features. These features are pre-processed by means of statistical *dimension reduction* techniques to identify a more compact representation, while not loosing important information. A statistical model based on *BN* correlates these reduced representations to the compiler optimizations to maximize application performance.

The optimization flow is shown in Figure 4.1 and consists of two main phases. During the initial *training phase*, the *Bayesian network* is learned on the base of a set of training applications (see Figure 4.1a). During the *exploitation phase*, new applications are optimized by exploiting the knowledge stored in the *Bayesian Network* (see an example of a BN topology in Figure 4.3).

During both phases, an optimization process is necessary to identify the best compiler optimizations to achieve the best performance. This is done for learning purposes during the *training phase* and for optimization purposes during the *exploitation phase*. To implement the optimization process, a Design Space Exploration (DSE) engine has been used. The DSE engine automatically compiles, executes and measures application performance by enabling/disabling different compiler optimizations. Which compiler optimizations will be enabled is decided in the Design of Experiments (DoE) phase. In our approach, the DoE is obtained by sampling from a given probability distribution that is either a *uniform distribution* (during the training phase as in Figure 4.1a) or an *application-specific distribution* inferred through the BN (during the *exploitation phase* as in Figure 4.1b).

The *uniform distribution* adopted during the *training phase* allows us to explore the compiler optimization space $\mathcal{O}$ uniformly to learn what the most promising regions of this space are. The *application-specific distribution* used during the *exploitation phase* allows us to speed up the optimization by focusing on the most promising region of the compiler optimization space $\mathcal{O}$.

4.4.1 Applying Program Characterization

The classic *supervised* Machine Learning (ML) approach deals with fitting a model exploiting a function f of program characterization. Function f might use a variety of comparison/similarity functions, such as *nearest-neighbor* and *graph-kernels*. To obtain a more accurate fitting, compiler researchers have been trying to understand the behavior of programs/kernels better and derive a *feature vector* that represents pair functionality efficiently. As a rule of thumb, the derived feature vector must be i) representative enough of its program/kernel, and ii) different programs/kernels must not have the same feature vectors as this will confuse the subsequent machine-learning process. Thus, building a huge non-efficient feature vector slows down the ML process and obtain less-precision.

Another goal of this work is to exploit the efficient use of different program characterization techniques and demonstrate their performance and effectiveness. Three characterization techniques have been selected among state-of-the-art works, namely, i) *dynamic feature selection* using *MICA* [102], ii) *static analysis* using *MilePost* [82] framework, and iii) our handcrafted combination of those two as hybrid analysis.

MICA. *Microarchitecture-independent workload characterization* represents a recent work on dynamic workload characterization [102]. It is a plugin for the Linux-PIN tool [148] and is capable of characterizing the fed kernels *independently* from its running architecture as it monitors the *non-hardware* features of the kernels. This feature is of interest for targeting embedded domain as one might not be able to exploit PIN tools on the board. The main categories of MICA include *Instruction-Level-Parallelism (ILP), Instruction Mix (ITypes), Branch Predictability (PPM), Register Traffic (REG), Data Stream Stride (Stride), Instruction and Data Memory Footprint (MEMFootprint) and Memory Reuse Distances (MEMReusedist)*.

MilePost. This recent tool [81, 82] was built as a plugin on top of *GCC* to capture static features of the programs. One advantage of *static analysis* is that the compiler researchers do not have to run the actual binary just like what they have to do by a dynamic feature technique. On the other hand, *static analysis* fails to capture any correlations between the source code and memory hierarchy and different data streams fed as input dataset.

Hybrid. The third characterization technique consists of the combination of the two previous ones. We believe that, in some cases, hybrid feature selection can capture the kernel behaviors better as it takes into account both feature-selection methods.

4.4.2 Dimension-Reduction Techniques

In the proposed approach, the *dimension-reduction* process is important for two main reasons: *a)* it eliminates the noise that might perturb further analyses, and *b)* it significantly reduces the training time of the BN. The techniques used are *Principal Component Analysis* (PCA) and *Exploratory Factor Analysis* (EFA). The experimental results show that the selection of a good dimension-reduction technique has a significant impact on the final model quality. In the original work proposed in [17], PCA was used. In this work, we changed the model by exploiting *Exploratory Factor Analysis* (EFA) as explained in the following paragraphs. Experimental results will show the benefits of using EFA with respect to PCA for the specific problem addressed herein. For a

quantitative comparison the readers is referred to Section 4.5.4 Table 4.5.

Let $\boldsymbol{\gamma}$ be a characterization vector storing all data of an application run. This vector stores l variables to account for either the static, dynamic or both analyses. Let us consider a set of known application profiles A consisting of m vectors $\boldsymbol{\gamma}$. The application profiles can be organized in a matrix P with m rows and l columns. Each vector $\boldsymbol{\gamma}$ (i.e. a row in P) includes a large set of characteristics, such as the instruction count per instruction type (for both static and dynamic analysis), information on the memory access pattern and information characterizing the control flow (e.g. the number and length of the basic blocks, average and maximum loop nesting, etc.). Many of these application characteristics (columns of matrix P) are correlated to each other in a complex way. A simple example of this correlation is the instruction mix information collected during the static analysis and the instruction mix information collected during the dynamic profiling (even though these are not completely the same). A less intuitive example is between the distribution of basic block lengths and data related to the instruction memory reuse distance. The presence of many correlated columns in P implies that the information stored in a vector $\boldsymbol{\gamma}$ can be well represented with a vector $\boldsymbol{\alpha}$ of smaller size.

Both PCA and EFA are statistical techniques aimed at identifying a way to represent $\boldsymbol{\gamma}$ with a shorter vector $\boldsymbol{\alpha}$ while minimizing the information loss. Nevertheless, they rely on different concepts for organizing this reduction [90, 223]. In both cases, output values are derived by applying the dimension reduction and are no longer directly representing a certain feature. While in PCA the components are given by a combination of the observed features, in EFA the factors are representing the hidden process behind the feature generation. In both cases, there is no way to indicate by name the output columns, since they are not directly observable.

In PCA, the goal is to identify a summary of $\boldsymbol{\gamma}$. To this end, a second vector $\boldsymbol{\rho}$ of the same length of $\boldsymbol{\gamma}$ (i.e. l) is organized by a variable change. Specifically, the elements of $\boldsymbol{\rho}$ are obtained through a linear combination of the elements in $\boldsymbol{\gamma}$. The way to combine the elements of $\boldsymbol{\gamma}$ for obtaining $\boldsymbol{\rho}$ is decided upon the analysis of the matrix P, and is such that all elements in $\boldsymbol{\rho}$ are orthogonal (i.e. uncorrelated) and are sorted by their variance. Thus the first elements of $\boldsymbol{\rho}$ (also named principal components) carry most of the information of $\boldsymbol{\gamma}$. The reduction can be obtained by generating a vector $\boldsymbol{\alpha}$ to keep only the first most significant principal components in $\boldsymbol{\rho}$, because the least significant ones carry little information content. Note that principal components in $\boldsymbol{\rho}$ (thus in $\boldsymbol{\alpha}$) are not meant to have a meaning; they are only used to summarize the vector $\boldsymbol{\gamma}$ as a signature.

In EFA, the elements in the vector of reduced size Îś are meant to explain the structure underlying the variables $\boldsymbol{\gamma}$, while $\boldsymbol{\alpha}$, represents a vector of *latent variables* that cannot be directly observed. The variables $\boldsymbol{\gamma}$ are expected to be a linear combination of the variables in $\boldsymbol{\alpha}$. In EFA, this relationship explains the correlation between the different variables in $\boldsymbol{\gamma}$; that is, correlated variables in $\boldsymbol{\gamma}$ are likely to depend on the same hidden variable in $\boldsymbol{\alpha}$. The relationship between the latent $\boldsymbol{\alpha}$ and the observed variables is regressed by exploiting the maximum likely method based on the data in matrix P.

When adopting PCA, each variable in $\boldsymbol{\alpha}$ tends to be a mixture of all variables in $\boldsymbol{\gamma}$. Therefore, it is rather hard to tell what a component represents. When adopting EFA instead, the components $\boldsymbol{\alpha}$ tend to depend on a smaller set of elements in $\boldsymbol{\gamma}$ that are cor-

related with each others. That is, when applying EFA, $\boldsymbol{\alpha}$ is a compressed representation of $\boldsymbol{\gamma}$, where elements in $\boldsymbol{\gamma}$ that are correlated (i.e. that carry the same information) are compressed into a reduced number of elements in $\boldsymbol{\alpha}$. Note that reducing the profile size by means of EFA results in a $\boldsymbol{\alpha}$ that better describes the type of application under analysis in reference to PCA [110].

Consequently, having obtained $\boldsymbol{\gamma}$ through any of the characterization techniques, a pre-processing filtering should be applied to ensure that the least noise has come through and the final P is eligible to be summarized by *EFA*. That implies manually i) removing the zero columns in l and ii) removing the redundant columns of l given that no column l is a linear combination of another l. In contrast, the algorithmic approach to tackle this is that P needs to be transformed, as to obtain the final $\boldsymbol{\gamma}$ in *positive-definite covariance* form [27]. Different techniques have been described in the literature on how to transform a *non-positive-definite* matrix to a *positive-definite* one which exceeds the scope of this chapter, but interested readers can refer to [137, 220] or use packages in *R* statistical tool [221] i.e., *nearPD* to compute nearest positive definite matrix.

4.4.3 Bayesian Networks

Bayesian Networks are powerful to represent the probability distribution of different variables that characterize a certain phenomenon. The phenomenon to be investigated in this work is the optimality of compiler optimization sequences.

Let us define a Boolean vector $\boldsymbol{o}$, whose elements o_i are the different compiler optimizations. Each optimization o_i can be either enabled, $o_i = 1$, or disabled, $o_i = 0$. In this work, the *phase ordering* problem [129] is not taken into account. but rather we consider how different optimizations o_i are organized in a predefined order embedded in the compiler. A compiler optimization sequence represented by the vector $\boldsymbol{o}$ belongs to the n size Boolean space $\mathcal{O} = \{0, 1\}^n$, where n represents the number of compiler optimizations under study.

An application is parametrically represented by the vector $\boldsymbol{\alpha}$ of the k reduced components computed either via PCA or via EFA from its software features. Elements α_i in vector $\boldsymbol{\alpha}$ generally belong to the continuous domain.

The optimal compiler optimization sequence $\bar{\boldsymbol{o}} \in \mathcal{O}$ that maximizes the performance of an application is generally unknown. However it is known that the effects of a compiler optimization o_i might depend on whether another optimization o_j has been applied. Additionally, it is known that the compiler optimization sequence that maximizes the performance of a given application depends on the application itself.

The reason why the optimal compiler optimization sequence $\bar{\boldsymbol{o}}$ is unknown a priory is because it is not possible to capture, in a deterministic way, the dependencies among the variables in the vectors $\bar{\boldsymbol{o}}$ and $\boldsymbol{\alpha}$. There is no way to identify an analytic model to exactly fit the vector function $\bar{\boldsymbol{o}}(\boldsymbol{\alpha})$. As a matter of fact, the best optimization sequence $\bar{\boldsymbol{o}}$ depends also on other factors that are somewhat outside our comprehension, *the unknown*. It is exactly to deal with *the unknown* that we propose not to predict the best optimization sequence $\bar{\boldsymbol{o}}$ but rather to infer its probability distribution. The uncertainty stored in the probability distribution models the effects of *the unknown*.

As underlying probabilistic model, we selected *BN* because of the following features of interest for the target problem:

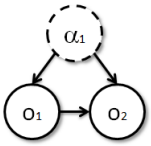


Figure 4.2: *A Bayesian Network example.*

- Their expressiveness allows one to include heterogeneous variables in the same framework such as Boolean variables (in the optimization vector $\boldsymbol{o}$) and continuous variables (in the application characterization $\boldsymbol{\alpha}$).
- Their capabilities to model *cause-effect* dependencies. Representing these dependencies is suitable for the target problem, as we expect that the benefits of some compiler optimizations (effects) are due to the presence of some application features (causes).
- It is possible to graphically investigate the model to visualize the dependencies among different compiler optimizations. If needed, it is even possible to manually edit the graph for including some *a priori* knowledge.
- It is possible to bias the probability distribution of some variables (the optimization vector $\boldsymbol{o}$) given the *evidence* on other variables (the application characterization $\boldsymbol{\alpha}$). This enables one to infer an *application-specific distribution* for the vector $\boldsymbol{o}$ from the vector $\boldsymbol{\alpha}$ observed by analyzing the target application.

A *Bayesian Network* is a direct acyclic graph whose nodes represent variables and whose edges represent the dependencies between these variables. Figure 4.2 reports a simple example with one variable α_1 representing the application features and two variables o_1, o_2 representing different compiler optimizations. In this example, the probability distributions of the two optimizations depend on the program features represented by α. Additionally, the probability distribution of o_2 depends on whether the optimization o_1 is applied. Dashed lines are used for nodes representing observed variables whose value can be input as evidence to the network. In this example, the variable α_1 can be observed and, by introducing its evidence, it is possible to bias the probability distributions of other variables.

Training the Bayesian model. Tools exist to construct *BN* automatically by fitting the distribution of some training data [162]. To do so, first the graph topology is identified and then the probability distribution of the variables including their dependencies is estimated.

The identification of the graph topology is particularly complex and time consuming. The *dimension reduction* technique applied on the SW features plays a key role in obtaining reasonable training times by limiting to k elements in the vector $\boldsymbol{\alpha}$, thus reducing the number of nodes in the graph.

For efficiency reasons, the algorithm used for selecting the graph topology is an heuristic algorithm, named *K2*, initialized with the *Maximum Weight Spanning Tree* (MWST) ordering method as suggested in the Matlab toolbox in use [162]. The initial ordering of the nodes for the MWST algorithm is given to let the elements $\boldsymbol{\alpha}$ to

appear first and then the elements of $\boldsymbol{o}$. Even if the final topological sorting of the nodes changes according to the algorithm described in [97], by using this initialization criterion, it always happens that the dependencies are directed from elements of $\boldsymbol{\alpha}$ to elements of $\boldsymbol{o}$ and not vice versa. When using the K2 algorithm, the network topology is selected as follows. The graph is initialized with no edges to represent the fact that each variable is independent. Then, for each variable i, following their initial ordering, each possible edge from j to i (where $j < i$) is considered as candidate to be added to the network. A candidate edge is added to the topology if it increases the probability that the training data were generated from the probability distribution the new topology describes. This method has a polynomial complexity with respect to the number of variables involved and the number of lines in the training data set.

During the model training, we consider the *softmax* function for modeling the cumulative probability distribution of the Boolean elements in vector $\boldsymbol{o}$ [162]. This is a mathematical necessity to map in the *Bayesian framework* the dependencies of Boolean variables in $\boldsymbol{o}$ with respect to continuous variables in $\boldsymbol{\alpha}$. In particular, thanks to the use of *softmax* variables, we can express the conditional probability $P(o_i = b \mid \alpha_j = x)$, where o_i is a Boolean variable and α_j is a continuous variable.

The coefficients of the functions describing the probability distribution of each variable as well as their dependencies are tuned automatically to fit the distribution in the training data [162]. Training data are gathered by analyzing a set A of training applications (Figure 4.1a). First, application features are computed for each application $a \in A$ to enable the principal component analysis. Thus, each application is characterized by its own principal component vector $\boldsymbol{\alpha}$. Then, an experimental compilation campaign is carried out for each application by sampling several compiler optimization sequences from the compiler optimization space $\mathcal{O}$ with a uniform distribution. For each application, we select the 15% best-performing compilation sequences among the sampled ones. The distribution of these sequences is learned by the *Bayesian Network* framework in relation to vector $\boldsymbol{\alpha}$ characterizing the application.

Inferring an application-specific distribution. Once the *Bayesian Network* has been trained, the principal component vector $\boldsymbol{\alpha}$ obtained for a new application can be fed as evidence to the framework to bias the distribution of the compiler optimization vector $\boldsymbol{o}$. To sample a compiler optimization sequence from this biased distribution, we proceed as follows. The nodes in the direct acyclic graph describing the *Bayesian Network* are sorted in topological order, i.e. if a node at position i has some predecessors, those appear at positions $j, j < i$. At this point, all nodes representing the variables $\boldsymbol{\alpha}$ appear at the first positions[1]. The value of each compiler optimization o_i is sampled in sequence by following the topological order such that all its parent nodes have been decided. Thus, the marginal probability $P(o_i = 0 \mid \mathcal{P})$ and $P(o_i = 1 \mid \mathcal{P})$ can be computed on the basis of the parent node vector value $\mathcal{P}$ (each parent being either an evidence α_j or a previously sampled compiler optimization o_j). Similarly, by using the maximum likelihood method, it is possible to compute the most probable vector from this biased probability distribution. When sampling from the application-specific probability distribution inferred through the *Bayesian Network*, we always consider to return the most probable optimization sequence as first sample.

[1]This is by construction due to the initialization of the MWST and the K2 algorithms used to discover the network topology.

4.5 Experimental Evaluation

The goal of this section is to assess the benefits of the proposed methodology. In this work, we run the experimental campaign on an ARMv7 Cortex-A9 architecture as part of a TI-OMAP 4430 processor [107] with *ArchLinux* and *GCC-ARM 4.6.3*.

4.5.1 Benchmark Suites

To assess the proposed methodology, we have used two major benchmark suites separately: i) *cBench* [86] and ii) *PolyBench* [91, 190]. Each consists of different classes of applications and kernels ranging from security and cryptography algorithms to office and image-processing applications. Readers can refer to Table 4.1 for the list of applications selected in the two benchmark suites.

Table 4.1: *Benchmark suites used in this work*

(a) *cBench applications selected for this work*

cBench list	Description
automotive_bitcount	Bit counter
automotive_qsort1	Quick sort
automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corner
automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Edge
automotive_susan_s	Smallest Univalue Segment Assimilating Nucleus S
security_blowfish_d	Symmetric-key block cipher Decoder
security_blowfish_e	Symmetric-key block cipher Encoder
security_rijndael_d	AES algorithm Rijndael Decoder
security_rijndael_e	AES algorithm Rijndael Encoder
security_sha	NIST Secure Hash Algorithm
telecom_adpcm_c	Intel/dvi adpcm coder/decoder Coder
telecom_adpcm_d	Intel/dvi adpcm coder/decoder Decoder
telecom_CRC32	32 BIT ANSI X3.66 crc checksum files
consumer_jpeg_c	JPEG kernel
consumer_jpeg_d	JPEG kernel
consumer_tiff2bw	convert a color TIFF image to grey scale
consumer_tiff2rgba	convert a TIFF image to RGBA color space
consumer_tiffdither	convert a TIFF image to dither noisespace
consumer_tiffmedian	convert a color TIFF image to create a TIFF palette file
network_dijkstra	Dijkstra's algorithm
network_patricia	Patricia Trie data structure
office_stringsearch1	Boyer-Moore-Horspool pattern match
bzip2d	Burrows Wheeler compression algorithm
bzip2e	Burrows Wheeler compression algorithm

(b) *Linear-algebra/applications of the PolyBench suite selected for this work*

PolyBench list	Description
2mm	2 Matrix Multiplications (D=A×B; E=C×D)
3mm	3 Matrix Multiplications (E=A×B; F=C×D; G=E×F)
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
doitgen	Correlation Computation
gemm	Matrix-multiply C = aA×B + bC
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
mvt	Matrix Vector Product and Transpose
symm	Symmetric matrix-multiply
syr2k	Symmetric rankâĹŠ2k operations
syrk	Symmetric rankâĹŠk operations
trisolv	Triangular solver
trmm	Triangular matrix-multiply

cBench

The *cBench* suite [86] is a collection of open-source programs with multiple data sets assembled by the community to enable realistic workload execution and targeted by many different compilers such as *GCC, LLVM*, etc.. The source code of individual programs is simplified to facilitate portability; therefore, it has been targeted in *autotuning* and *iterative compilation* research work. Of the available data sets for every individual kernel, we have selected five and sorted them in a way that dataset1 is always the smallest and dataset5 the largest. This ensures that for every kernel we have exposed enough of the input load to be able to measure fair runtime executions.

PolyBench

The *PolyBench* benchmark suite [91, 190] consists of benchmarks with static control parts. The purpose is to make the execution and monitoring of applications uniform. One of the main features of the *PolyBench* suite is that there is a single file per application, tunable at compile-time and used for kernel instrumentation. It performs extra operations such as cache flushing before the execution, and can set real-time scheduling to prevent OS interference. We have defined two different data sets for each individual application to expose the main function with different input loads. PolyBench has a variety of benchmarks, i.e. 2D and 3D matrix multiplication, vector decomposition, etc.. This suite is also suitable for parallel programming, which is beyond the focus of this work.

4.5.2 Compiler Transformations

The compiler transformations analyzed have been reported in Table 6.2. We based our design space on the work of [46]. The authors implemented sensitivity analysis over a vast majority of the compiler optimizations and defined with a list of promising passes. Building upon their work, we selected the compiler optimizations with a speedup factor greater than 1.10. They are applied to improve application performance beyond the standard optimization level -O3 and have not yet been included in any prior optimization level. The optimizations can be enabled/disabled by means of the respective compiler optimization flags. The standard optimization level -O3 has been also used to collect the dynamic software-features for each application on both *training* and *inference* phases.

The application execution time has been estimated by using the Linux-perf tool. The execution time is done by averaging five loop-wraps of the specific compiled binary with one second of sleep in between five different executions of those loop-wraps. Therefore, in total, each individual transformed binary has been executed 25 times as five packages of five loop-wraps to ensure better accuracy of estimations and fairness among the generation of executions. This technique is used both in the *training* and the *inference* phases.

4.5.3 Bayesian Network Results

In this work, Matlab environment [162, 196] has been used to train the *Bayesian Network*. We have used *Exploratory Factor Analysis* (EFA) of application features for the seven compiler optimization flags listed in Table 6.2. As stated in Section 4.4.2, one

Table 4.2: *Compiler optimizations under analysis (beyond -O3)*

Compiler Transformation	Abbreviation	Short Description
-funsafe-math-optimizations	*math-opt*	Allow optimizations for floating-point arithmetic that (a) assume valid arguments and results and (b) may violate IEEE or ANSI standards.
-fno-guess-branch-probability	*fn-gss-br*	Do not guess branch probabilities using heuristics.
-fno-ivopts	*fn-ivopt*	Disable induction variable optimizations on trees.
-fno-tree-loop-optimize	*fn-tree-br*	Disable loop optimizations on trees
-fno-inline-functions	*fn-inline*	Disable optimization that inline all simple functions.
-funroll-all-loops	*funroll-lo*	Unroll all loops, even if their number of iterations is uncertain
-O2	*O2*	Overwrite the -O3 optimization level by disabling some optimizations involving a space-speed tradeoff

Table 4.3: *Kaiser test results*

Application	Characterization Method	Original No. of Factors	Range of Selected Factors By Kaiser Test
cBench	MICA (Dynamic)	99	[7-11]
cBench	MILEPOST (Static)	53	[4-6]
cBench	Hybrid	143	[8-10]
polyBench	MICA (Dynamic)	99	[5-7]
polyBench	MILEPOST (Static)	53	[4-6]
polyBench	Hybrid	143	[4-5]

of the features of using EFA is that the factors are linear combinations that maximize the shared portion of the variance. Therefore, as prerequisite, the covariance matrix should be positive definite. This pre-processing helps purify the highly correlated application characterization columns that are linearly correlated. In theory, *PCA* accepts any matrix ignoring the aforementioned condition and that is why we think applying factor analysis as our dimension reduction technique tends to obtain the most important factors and correlate them with the compiler optimizations. Decision on the numbers of factors have been derived from the *Kaiser* test [113]. This test implies taking only the factors having greater than 1 in the covariance matrix. In other words, the Kaiser rule is to drop all components with eigenvalues under 1, this being the eigenvalue equal to the information accounted for by an average single item. Table 4.3 reports the factors derived for each individual benchmark and characterization method.

Table 4.3 represents the number of features that have been produced both originally by the different feature selection techniques and by the Kaiser test. The third column is the *original number of features* and the last one refers to *range of selected factors* in each specific benchmark suite/feature selection method. Note that the last column reports the range of selected factors rather than a number as we have used cross-validation approach in the experimental campaign, thus different applications/datasets/feature selection techniques can result in a different number of factors to be used in COBAYN's framework.

In this work, while training has been carried out using each application/dataset pair separately, the validation has been done through an application-level cross-validation

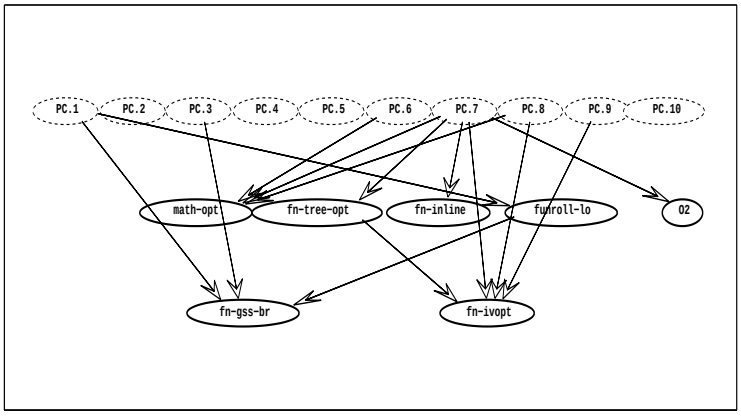


Figure 4.3: *Topology of the Bayesian Network if security_rijndael_e is left out of the training set*

(*Leave-One-Out* cross-validation, LOO). We train different BNs, each by excluding an applications (together with all its input dataset) from the training set.

Using *BN* enables us to investigate graphically the dependencies between the variables involved in the compiler optimization problem and to correlate them with the selected factors of the program characterization. We train a final *Bayesian Network* including all applications in the training set. The resulting network topology is a directed acyclic graph *DAG*, as shown in Figure 4.3. By removing *security_rijndael_e* application from the training set, the graph topology slightly changes, mainly in terms of the different edges connecting the *Principal Components (PC)/program factors (FA)* nodes to the compiler optimization nodes. This is due to the change in the program features and its factors, which are computed in a different way. For the sake of conciseness, we do not report all graph topologies derived by the LOO technique for each individual trained *Bayesian Network*.

The Nodes of the topology graph reported in Figure 4.3 are organized in layers. The first layer reports the FAs that are the observable variables (reported as dashed lines). The second layer contains the compiler optimizations whose parents are the PC nodes (or FA nodes depending weather PCA or EFA is used). Therefore the effects of these compiler optimizations depend only on the application characterization in terms of its features. In the third layer, the compiler optimization nodes whose parents include optimization nodes from the second layer are listed. Once a new application is characterized for a target application data set, the evidence related to the PCs (or FAs) of its features is fed to the network in the first layer. Then, the probability distributions of other nodes can be inferred in turn on the second and third layers. There are two nodes in the third layer of Figure 4.3. The first one is the *fn-gss-br* node that depends on *funroll-lo* because unrolling loops impacts the predictability of the branches implementing these loops. Moreover, *funroll-lo* impacts the effectiveness of the heuristic branch probability estimation, thus *fn-gss-br*. The second node in the third layer is the *fn-ivopt* node, which depends on *fn-tree-opt* as parent node in the second layer. Both these optimizations work on trees and therefore their effects are interdependent. While sampling compiler optimizations from the *Bayesian Network*, the decisions of whether to apply *fn-gss-br* and *fn-ivopt* are taken after deciding whether to apply *funroll-lo* and *fn-tree-opt*.

Table 4.4: *COBAYN timing breakdown for offline training and online inference for* Susan *application*

Phase	Tag & Category	Time
Offline training	(A) Offline data-collection	2 days
	(B) Construct BN	70 sec
Online inference	(C) SW Feature Collection	14.4 sec
	(D) BN Inference	0.85 sec
	(E) Susan compilation	4.5 sec
	(F) Susan execution	8.9 sec

Table 4.4 shows the fine-grain breakdown of the timing when we use COBAYN framework. We have reported the time spent for each phase of the proposed technique, both on the training phase (done offline) and on the inference phase (done online). Constructing COBAYN's network is a one-time process and depends on the number of applications in the training set. The time needed to collect the training data is on the other side, dependents not only on the number but also on the applications and data-set used for the training. The same happens for the time needed for compiling and executing the target application during the online compiler autotuning phase. To that end, Table 4.4 reports the numbers for each specific phase considering the cBench as training set and Susan as target application. During the offline training-phase, the time needed for data collection on the case of cBench, is around 2 days. It includes the time needed for each benchmark to compile and execute, considering all set of configuration and the feature collection phase. The time needed to post-process the data and to generate the Bayesian Network model is around 70 seconds.

During the online phase (inference phase), the time needed for extracting the software features from the target application is 14.4 seconds while, querying BN is less than 1 second. The compilation and execution-time on the target platform for Susan are 4.5 and 8.9 seconds, respectively. Those numbers show that the initial overhead in adopting the proposed methodology on the user-side (composed of the software feature extraction and BN inference) is less than 2 compilation/executions pairs, for this specific example.

4.5.4 Comparison Results

It is well known that *Random Iterative Compilation* (RIC) can improve application performance compared with static handcrafted compiler optimization sequences [7]. Additionally, given the complexity of the *iterative compilation* problem, it has been proved that drawing compiler optimization sequences at random is as good as applying other optimization algorithms such as genetic algorithms or simulated annealing [7, 17, 41, 46]. Accordingly, to evaluate the proposed approach, we compared our results with **i)** standard optimization levels -O2 and -O3 **ii)** the Random Iterative Compilation (RIC) methodology that samples compiler optimization sequences from the uniform distribution and **iii)** two advanced state-of-the-art methodologies, namely *a)* (Section 4.5.6) coupling machine learning with an iterative methodology, and *b)* (Section 4.5.6) a non-iterative methodology derived by *predictive modeling* based on different machine-learning algorithms to predict the final application speedup.

The proposed methodology samples different compiler optimization sequences from the BN. The performance achieved by the best application binary depends on the num-

Table 4.5: *COBAYN (BN using EFA) speedup w.r.t standard optimization levels (-O2 and -O3) and Random Iterative Compilation (RIC) and our previous approach of BN in [17] using PCA*

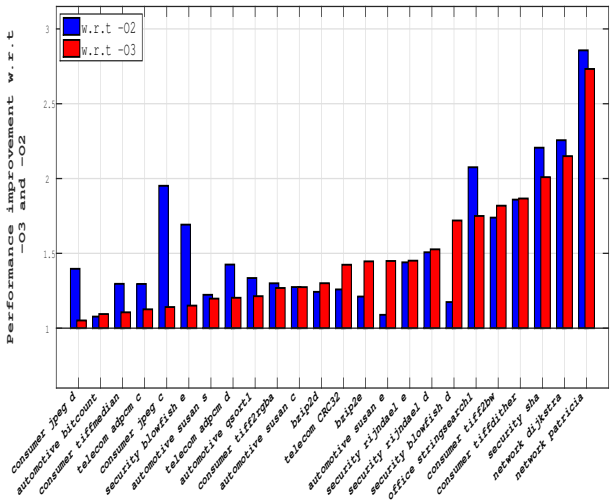
Benchmarks	Features	COBAYN Speedup w.r.t			
		-O2	-O3	RIC	BN w/ PCA
cBench	Dynamic	1.6093	1.528	1.2029	1.0744
cBench	Static	1.5447	1.478	1.1143	1.0543
cBench	Hybrid	1.5858	1.5066	1.2086	1.0654
cBench Average		1.5795	1.5035	1.1743	1.0617
PolyBench	Dynamic	1.9845	1.8387	1.3230	1.0921
PolyBench	Static	1.9353	1.8215	1.1518	1.0724
PolyBench	Hybrid	1.9441	1.7726	1.2333	1.1078
PolyBench Average		1.9541	1.8101	1.2350	1.0901
Overall (Harmonic Mean)		**1.7669**	**1.6571**	**1.2052**	**1.0771**

ber of sequences sampled from the model. In this section, the result of applying the proposed methodology using two benchmark suites with respect to standard optimization levels and the *random iterative compilation* have been reported. The performance speedup on the first comparison section is measured in reference to -O2 and -O3, which are the optimization levels available for GCC. In addition, we show the speedup of the proposed methodology with respect to our previous work [17].

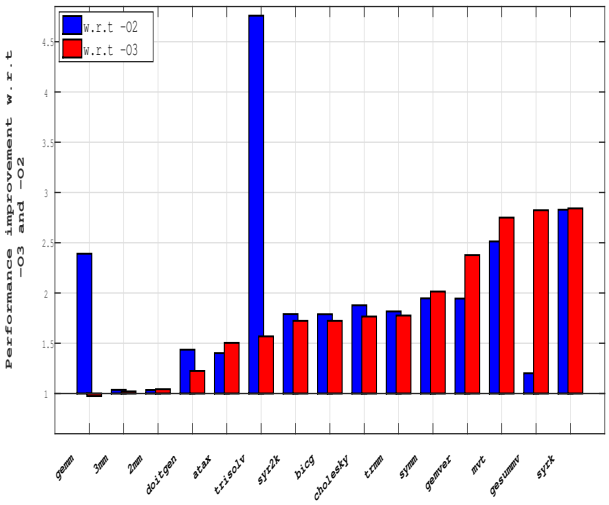
Bayesian Networks Performance Evaluation

Table 4.5 reports COBAYN's speedup achieved over the standard optimization levels of *-O2* and *-O3* and *Random Iterative Compilation (RIC)*. The last column represents the average speedup achieved by revising our *Bayesian Network* engine and using the *Explanatory Factor Analysis (EFA)* described in the Section 4.4.2 with respect to PCA in [17]. Note that all speedup values have been averaged using Harmonic mean. It is observed that in all categories, COBAYN outperforms standard optimization levels and the previous approach. The comparison with respect to the RIC has been reported by the Harmonic average over the speedup data derived by dividing the COBAYN's performance data by the RIC data in full space. It can be seen that dynamic feature selection brings best results followed by the hybrid and static method. However, in certain cases (cBench using hybrid SW features), it narrowly reaches the performance of dynamic feature selection.

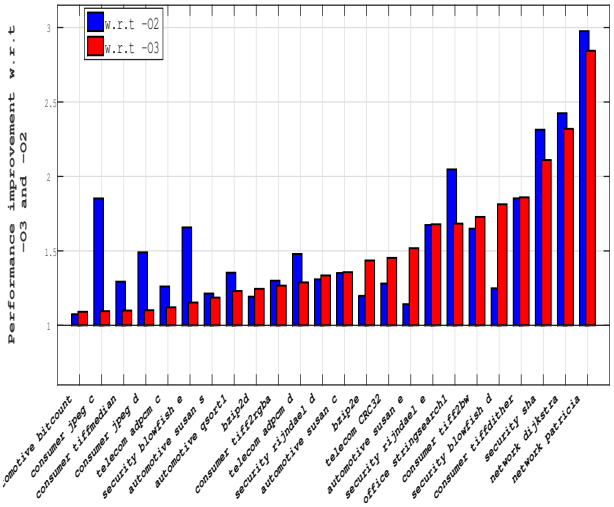
Using two major benchmarks and three different application characterization techniques, we report six different plots showcasing the benefits of the proposed methodology with respect to the GCC standard optimization levels. Figure 4.4, reports the speedups by considering a sample of eight different compiler optimization sequences. For each benchmark, the results have been averaged on the different data sets. All results have been sorted by the speedup values of -O3 and have been matched with their corresponding -O2 value. The bar plot is colored in blue and red, respectively, for the speedup achieved with respect to -O2 and -O3. All applications have achieved a speedup in reference to the performance of -O2 and -O3. This happens with the exception of *gemm* in reference to -O3 for static and hybrid feature-selection techniques and *consumer-jpeg-d* in reference to -O3 when using the dynamic method for feature selection. These applications reach their best performance using -O3 for two data sets



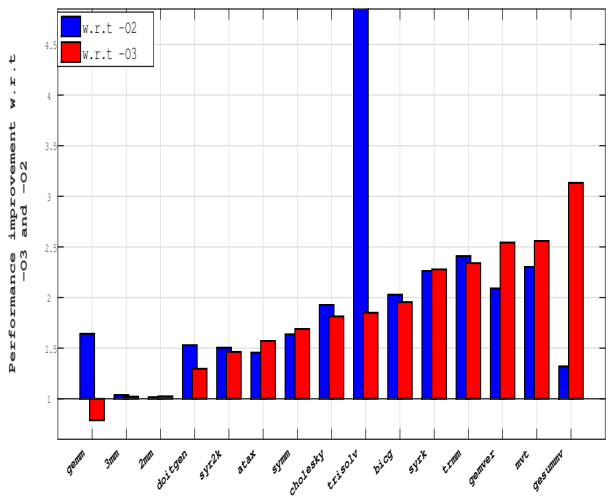
(a) *BN with dynamic features on cBench*



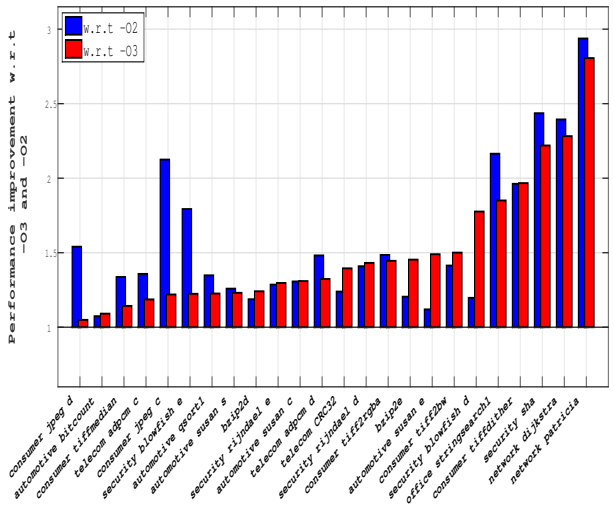
(b) *BN with dynamic features on PolyBench*



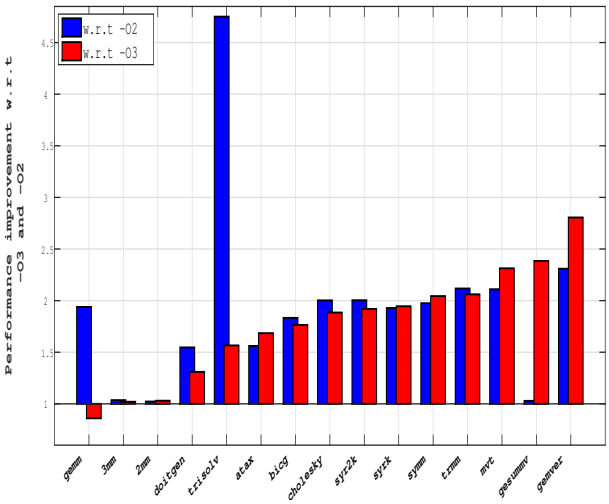
(c) *BN with static features on cBench*



(d) *BN with static features on PolyBench*



(e) *BN with dynamic+static features on cBench*



(f) *BN with dynamic+static features on PolyBench*

Figure 4.4: *Performance speedup w.r.t -O2 and -O3*

Table 4.6: *Evaluation of different BigSet formation in COBAYN Model Construction. Note that COBAYN's default refers to the version of COBAYN trained on a single benchmark set.*

BigSet Combination		Speedup w.r.t.
cBench	PolyBench	COBAYN's Default
24 (All)	15 (All)	1.1143
15	15	1.0743
10	10	1.0432
5	5	0.9896

out of five, and it was not possible to surpass this maximum by relying on the compiler transformations under consideration. On average for *cBench*, the speedups are of **1.57** and **1.5** in reference to -O2 and -O3, respectively, and **1.95** and **1.81** for *PolyBench*. The maximum speedup observed is **3.1**× and **4.7**×. Table 4.5 reports the speedup gained using COBAYN compared with the standard optimization levels, *Random Iterative Compilation (RIC)* and our previous approach exploiting *PCA* as dimension-reduction method.

Analysis of the Portability of COBAYN in Different Scenarios The results reported in this section are computed by means of LOO cross-validation on the two individual benchmark suites separately, one with 24 and the other with 15 applications. As the nature of these two benchmark suites is totally different, we believed it would be unfair to train on one and test on the other, so we analyzed the feasibility of mixing these applications in a fair heterogeneous set of *BigSet* so that COBAYN's engine gets evaluated. To this end, we tried 4 different scenarios, where the BigSet is obtained by: (i) including all 39 available applications, (ii) 15 applications of cBench and 15 applications of PolyBench, (iii) selecting 10 cBench and 10 PolyBench and finally (iv) 5 applications from each of those. Therefore, the BigSet was initialized with 39, 30, 20 and 10 different applications, and LOO cross-validation was carried-out. Table 4.6 reports the speedup gained in these scenarios. It is observed that COBAYN framework benefits from having *a)* more applications, and *b)* heterogeneous applications in the training set. The speedup listed in Table 4.6 is higher when BigSet accounts for more applications and, even just 10 applications per benchmark suite, it is higher than one (the default setting for the experimental results in this work refers to the COBAYN trained only on one of the two benchmark suites).

Performance Improvement

Let us define the *Normalized Performance Improvement* (NPI) as the ratio of the performance improvement achieved over the potential performance improvement:

$$NPI = \frac{E_{ref} - E}{E_{ref} - E_{best}} \tag{4.1}$$

where E is the execution time achieved by the methodology under consideration, E_{ref} is the execution time achieved with a reference compilation methodology and E_{best} is the best execution time computed through an exhaustive exploration of all possible compiler optimization sequences (in our case 128 different sequences). As the execution time E of the iterative compilation methodology under analysis gets closer to the

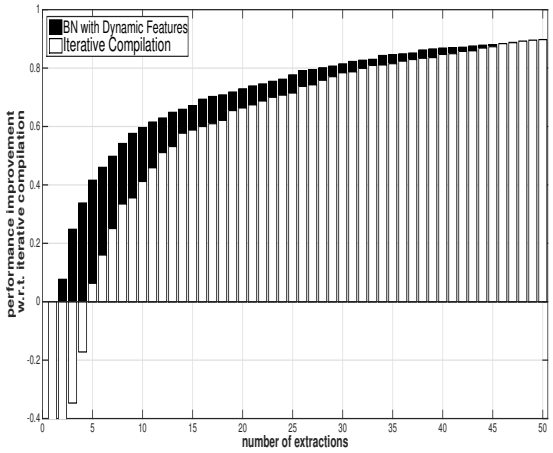
reference execution time E_{ref}, the NPI gets closer to 0, reporting that no improvement is returned. In the same way, as E gets closer to the best execution time E_{best}, while NPI gets close to 1, reporting that the entire potential performance improvement has been achieved.

Figure 4.5 reports for six different benchmark/feature selection methods; the NPI achieved by the *proposed optimization* technique and by the RIC technique in reference to the execution time obtained by -O3 (E_{ref}). It is noticeable that NPI has the upper-hand on performance on every number of extractions with respect to RIC. For readability purposes, we have only reported the first 50 extraction of the design space. The trend is continuously applied to the rest of the extractions until both get the maximum performance value of 1 at extraction no. 128, which accounts for the optimal compiler sequence given the specific application (also it is the optimal performance using exhaustive search). The comparisons reported in Figure 4.5 were carried out by considering the same number of compiler optimization sequences sampled for both the RIC and the *proposed* approach. We acknowledge the fact that there is still room for improvement in future work. However, NPI figures show that in all cases, the proposed method was superior in terms of performance and that 30 extractions, on the current scale, reach 80$\%$ of the optimality.

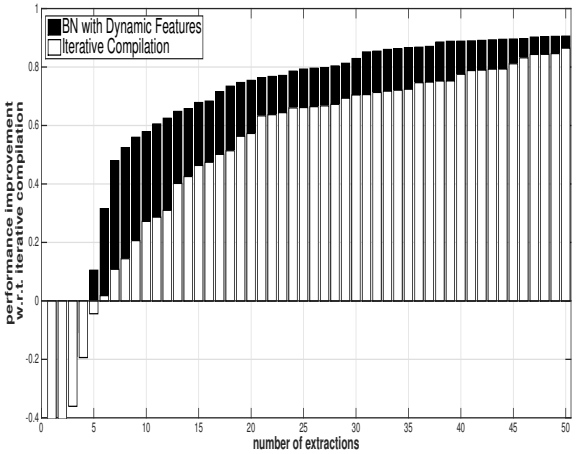
4.5.5 A Practical Usage Assessment

When using *iterative compilation* in realistic cases, we need to decide how much effort should be spent on the optimization itself. This effort can be measured in terms of optimization time, which is directly proportional to the number of compilations to be executed. Thus, in this section, we evaluate the proposed optimization approach in terms of the application performance reached after a fixed number of compilations. In particular, we fix this number to eight which represents 6.25% of the overall optimization space. Our model has been compared with RIC and in Figure 4.6, we report the *violin* plot for application speedup, while keeping the compilation effort of the proposed methodology to eight compilations (or extractions) and varying the compilation efforts of the RIC to explore more compiler space in the long run. Each individual distribution in Figure 4.6 represents the performance of the proposed work with respect to *RIC* across different extractions. The red cross marks the *mean* and the green square marks the *median* of each violin distribution. It can be seen that the proposed methodology with BN inference achieved an at least 3$\times$ reduction in exploration process effort compared with the same extraction of RIC. Here we define *exploration speedup* as the factor measuring the aforementioned metrics, enabling the researchers to traverse the compiler design space more efficiently.

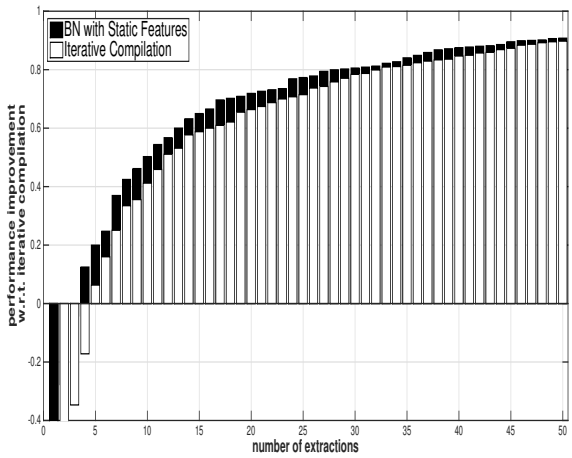
Accordingly, by increasing the compilation efforts on *RIC*, while keeping the exploration efforts of the proposed approach constant, the application speedup of COBAYN decreases. On average, RIC needs 24-32 extractions to achieve the application performance obtained with eight extractions by COBAYN. This means that COBAYN provides a speedup of 3-4$\times$ in terms of optimization efforts, that is only slightly impacted by the initial overhead (less than 2 evaluations) reported in Section 4.5.3. Furthermore, at the most extreme case, when RIC exhaustively enumerates and explores the full-space, 8 extractions of COBAYN, on average, still could gain up to 91% of the optimal solution. This is shown on the final distribution of each *violin* plot separated



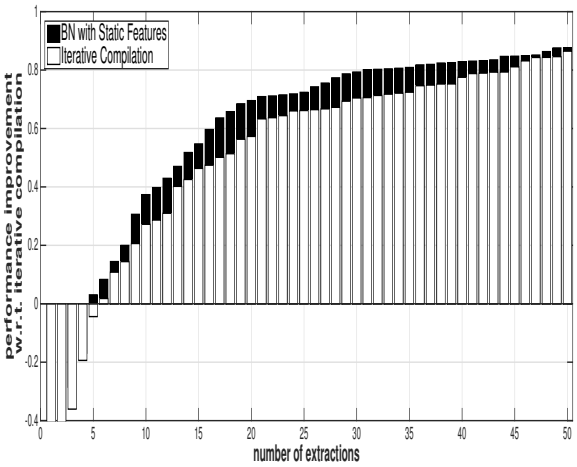
(a) *BN with dynamic features on cBench*



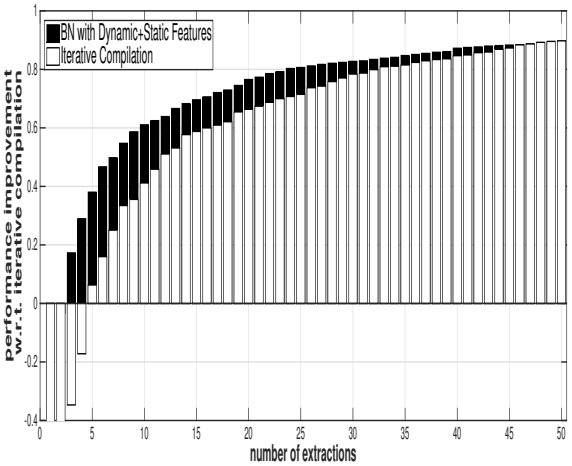
(b) *BN with dynamic features on PolyBench*



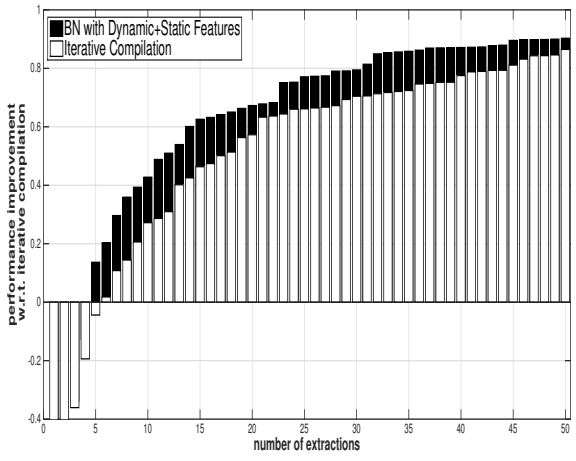
(c) *BN with static features on cBench*



(d) *BN with static features on PolyBench*

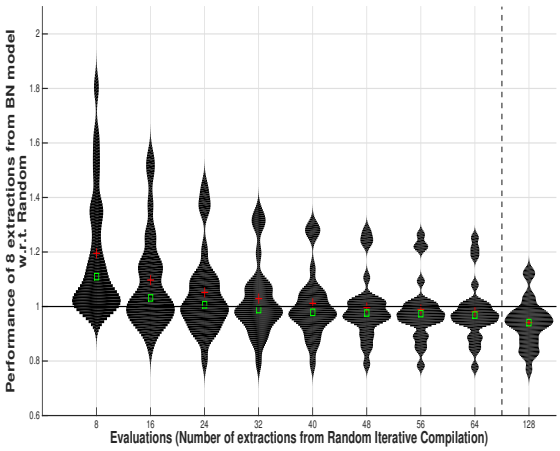


(e) *BN with hybrid features on cBench*

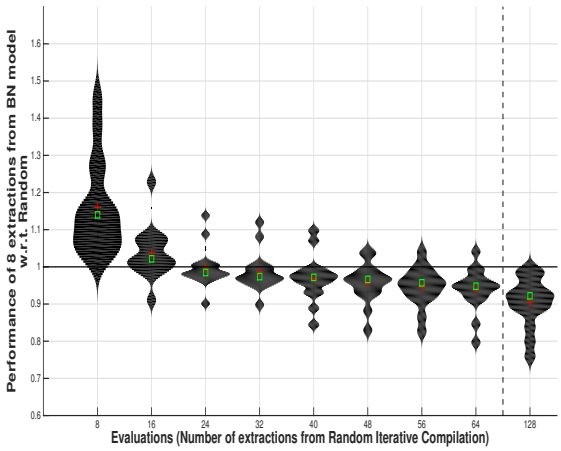


(f) *BN with hybrid features on PolyBench*

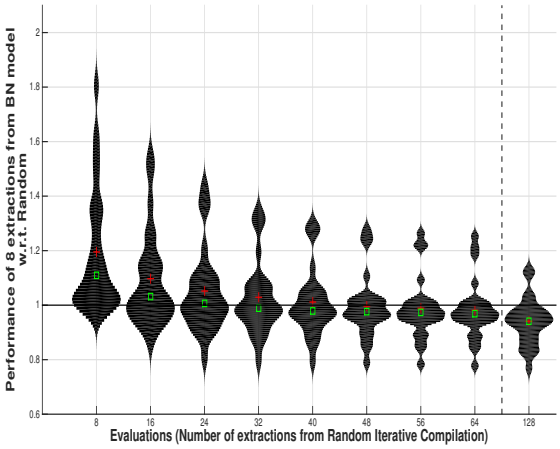
Figure 4.5: *Normalized performance improvement (NPI) w.r.t. RIC model*



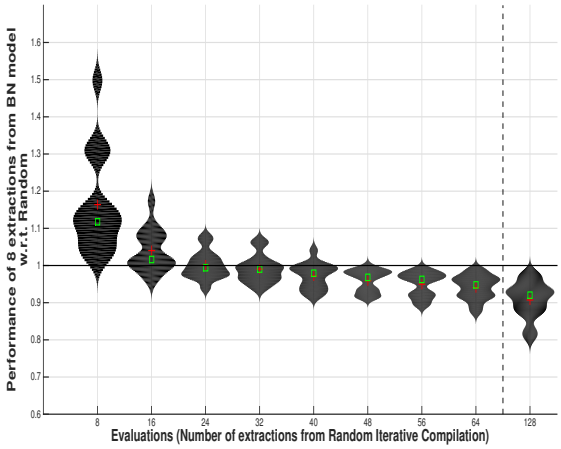
(a) *BN with dynamic features on cBench*



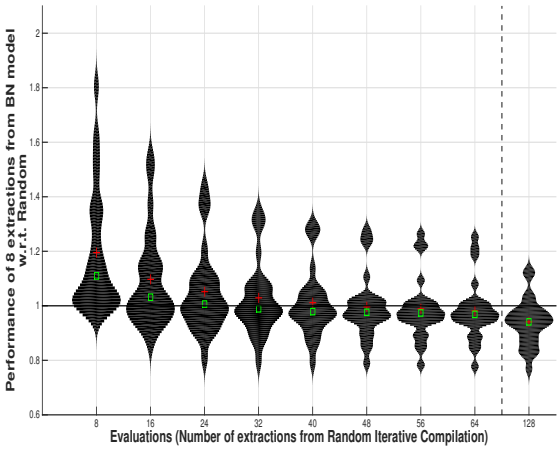
(b) *BN with dynamic features on PolyBench*



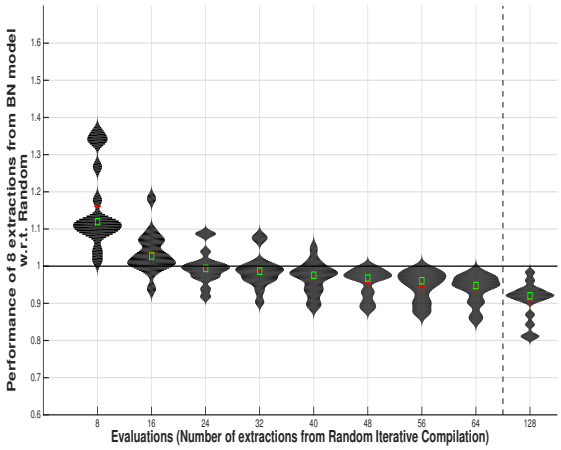
(c) *BN with static features on cBench*



(d) *BN with static features on PolyBench*



(e) *BN with dynamic+static features on cBench*



(f) *BN with dynamic+static features on PolyBench*

Figure 4.6: *Exploration speedup of 8 extractions w.r.t different evaluations of RIC*

by a vertical dashed-line.

4.5.6 Comparison with State-of-the-Art Techniques

In this section, we compare the quality of the COBAYN results with respect to approaches that derived from (A) an *iterative compilation* and (B) a *non-iterative compilation* methodology.

Comparison to a Iterative Compilation Methodology

[7] introduced machine-learning models to focus on the exploration of the compiler optimization for the most promising region. Their methodology exploits a Markov chain oracle and an independent identically distributed (IID) probability distribution oracle. These two offline-learnt models bias certain optimizations over others and replace the uniform probability distribution we applied earlier for the RIC reference methodology. [7] reports significant speedup by coupling these machine-learning models with a nearest-neighbor-classifier. When predicting the probability distribution of the best compiler optimizations for a new application, the classifier first selects the training application having the smallest Euclidean distance in the feature vector space (derived by PCA). Then it learns the probability distribution of the best compiler optimizations for this neighboring application either by means of the Markov chain model or by using an IID model. These probability distribution learnt is then used as the predicted optimal distribution for the new application. It has been reported that the Markov chain oracle outperforms the IID oracle, followed by the RIC methodology using a uniform probability distribution.

We constructs the $P(S)$ probability matrix reported in Section 4.2 and 4.3 of [7] as:

$$P(S_{IID}) = s_1, s_1, ..., s_L = \prod_{i=1}^{L} P(s_i) \tag{4.2}$$

$$P(S_{Markov}) = P(s_1) \prod_{i=2}^{L} P(s_i|s_{i-1}) \tag{4.3}$$

where $P(S_{IID})$ and $P(S_{Markov})$ define the probability of the specific sequence with IID and Markovian property for the optimization $t_1, t_1, ..., t_L$. Using LOO cross-validation, we find the closest neighbor for each cBench application trained by the two oracles, and we sample from their probability distributions. To comply with the original work in [7], we consider only the five most relevant principal components PCs and account only for the *static program features* (also when applying COBAYN). The results are depicted in Figure 4.7. It shows that COBAYN is faster in reaching higher speedup values. The results are scaled and normalized with respect to -O3 by using the NPI value (Equation 6.5). COBAYN is able to capture a more realistic probability matrix of the compiler optimization problem and achieves with faster convergence towards the optimal result. It brings 1.25× and 1.47× speedup with respect to IID and the Markov oracle, respectively.

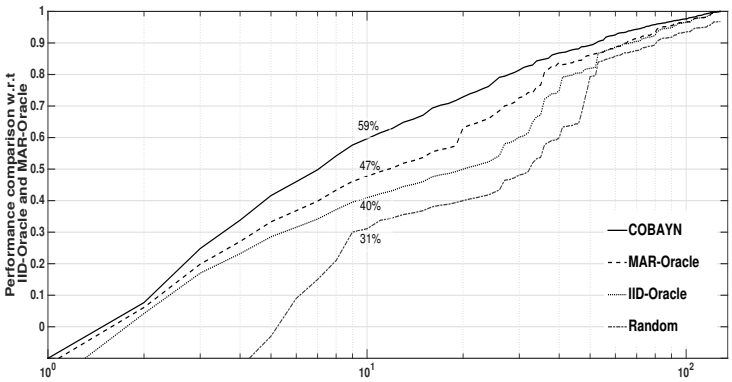


Figure 4.7: *NRI-scales speedup comparison of COBAYN with IID-oracle and MAR-oracle reported in [7]*

Comparison to a Non-iterative Compilation Methodology

[178], used a polyhedral compiler framework capable of predicting the speedup for an unseen application. They used certain loop-optimizations in their *design-space* and surfed the *full-search* of the space. They reported the average speedup gained with respect to standard optimization O3 by using different machine-learning models on *WEKA* [94] machine-learning environment.

i) Their predictive models are based on performance counters that are collected from the underlying architecture while running the applications. Therefore, the program features to be exposed to the model are architecture dependent, and the model loses its portability when it is used for a different architecture.

ii) We also explore a different compiler optimization space. They have explored polyhedral optimization space including loop transformations, whereas we focus on GCC optimization space including loop transformations and other optimizations such as *inlining, math optimizations*, etc..

iii) Furthermore, our model is based on a statistical analysis and BN, whereas they use a different set of machine-learning techniques, namely, predictive models. [178].

Nonetheless, we compare their methodology by applying it to the problem at hand. Table 4.7 reports the results obtained by using the machine-learning models in [178] on our compiler optimization space. We reproduced the data on both 1-shot and 8-shot scenarios to conform with the number of inference (predictions) COBAYN has in the current work. We use the Harmonic mean to average the speedup here. Note that Harmonic mean is always less than or equal to the arithmetic mean [99]. In all cases, COBAYN outperforms the reference methodology; specifically we have at least 1.3× and up to 2.04× speedup compared with the best achieved results reported in [178].

4.6 Conclusions

This chapter presents COBAYN, a methodology to infer by means of a *Bayesian framework* the best compiler optimizations to be applied for optimizing the performance of a target application. The methodology uses target independent software features to sample a statistical model built using Bayesian Networks to extract a set of suitable compiler configurations. Feature reduction techniques have been adopted to reduce the

Table 4.7: *COBAYN speedup w.r.t. the average speedup gained with predictive modeling in both 1-shot and 8-shot scenarios reported in [178].*

Algorithm and Parameter Configuration	COBAYN Speedup	
	w.r.t 1-shot	w.r.t 8-shot
LR -S 0	1.7551	1.6673
LR -S 1	1.7590	1.6710
LR -S 2	1.7191	1.6331
SVM NormalizedPolykernel -C 1.0 -E 8.0	1.5437	1.4665
SVM RBFKernel -C 2.0 -G 0.0	1.5206	1.4445
SVM RBFKernel -C 2.0 -G 25.0	1.5082	1.4327
SVM RBFKernel -C 2.0 -G 50.0	1.5045	1.4292
SVM RBFKernel -C 2.0 -G 75.0	1.5029	1.4277
SVM RBFKernel -C 2.0 -G 30.0	1.4927	1.4180
SVM RBFKernel -C 4.0 -G 30.0	1.5073	1.4319
SVM RBFKernel -C 0.01 -G 30.0	1.5073	1.4374
SVM RBFKernel -C 4.0 -G 50.0	1.5045	1.4292
IBk -K 1	1.4447	1.3724
IBk -K 2	1.4667	1.3933
IBk -K 5	1.4887	1.4142
M5P -M 1.0	1.4281	1.3566
M5P -M 2.0	1.4282	1.3567
M5P -M 4.0	1.4282	1.3568
M5P -M 10.0	1.4575	1.3846
M5P -M 50.0	1.4913	1.4167
K* -B 0 -M a	1.5192	1.4432
K* -B 20 -M a	1.5216	1.4455
K* -B 25 -M a	1.5258	1.4495
K* -B 50 -M a	1.4740	1.4003
K* -B 75 -M a	1.4737	1.4001
K* -B 100 -M a	1.5172	1.4413
K* -B 0 -M n	1.5208	1.4447
K* -B 20 -M n	1.5216	1.4455
K* -B 25 -M n	1.5258	1.4495
K* -B 50 -M n	1.4740	1.4003
MLP -L 0.3 -N 500 -H a	2.0435	1.9413
MLP -L 0.05 -N 500 -H a	1.6738	1.5901
MLP -L 0.1 -N 500 -H a	1.7246	1.6383
MLP -L 0.5 -N 500 -H a	1.8138	1.7231
MLP -L 0.9 -N 500 -H a	1.5250	1.4487
MLP -L 0.4 -N 500 -H a	1.9426	1.8454
MLP -L 0.5 -N 1000 -H a	1.8535	1.7608
MLP -L 0.5 -N 1500 -H a	1.7388	1.6518
MLP -L 0.5 -N 500 -H t	1.5579	1.4801
AVERAGE (Harmonic Mean)	**1.5622**	**1.4841**

complexity and training time of the Bayesian model while also eliminating possible noise in the data and improving the quality of the results. The proposed approach has been evaluated on an ARM-based platform, using GCC compiler. The experimental results demonstrated that the proposed technique outperforms both standard optimization levels and state-of-the-art iterative and not iterative compilation techniques while using the same number of evaluations.

4.7 Dissemination of The Chapter

The excerpt of this chapter has been published in ACM Transactions on Architecture and Code Optimization (TACO), June 2016, 13(2):21 under the title *COBAYN: Compiler Autotuning Framework Using Bayesian Networks*. Refer to the Chapter 7.2 for the list of publications of my PhD dissertation.

CHAPTER 5

The Phase-ordering Problem: An Intermediate Speedup Prediction Approach

5.1 Summary

Today's compilers offer a vast number of transformation options to choose among and this choice can significantly impact on the performance of the code being optimized. Not only the selection of compiler options represents a hard problem to be solved, but also the ordering of the phases is adding further complexity, making it a long standing problem in compilation research. This chapter presents an innovative approach for tackling the compiler phase-ordering problem by using predictive modeling. The proposed methodology enables *i)* to efficiently explore compiler exploration space including optimization permutations and repetitions and *ii)* to extract the application dynamic features to predict the next-best optimization to be applied to maximize the performance given the current status. Experimental results are done by assessing the proposed methodology with utilizing two different search heuristics on the compiler optimization space and it demonstrates the effectiveness of the methodology on the selected set of applications. Using the proposed methodology on average we observed up to 4% execution speedup with respect to LLVM standard baseline.

5.2 Introduction

Selecting the best ordering of compiler optimizations for an application has been an open problem in the field for many decades and the problem is known to be NP-complete. The unrealistic exhaustive search is the only solution that seems appealing to achieve the optimal solution. Compiler researchers rely on their insights on the compiler *backend* to come up with some predefined sequences and ordering. This process is usually done tentatively and the selected pass is constructed with little insight on the interaction between the selected compiler options. However, to come up with an optimal solution, researchers might have to spend several years to run different code variants and this is simply unfeasible, given the growing design space composed of different architectures and software models that rely on modern compiler frameworks. As an example, GCC compiler has more than 200 compiler passes and LLVM-OPT has more than 100, and these optimizations are working on different layers of application e.g. *analysis passes, loop-nest passes*, etc. Most of the passes are usually turned off by default and compiler developers rely on software developers to know which optimization can be beneficial for their code. The so-called *average case* has been defined as to get certain *standard optimization levels*, e.g. O1, O2, Os, etc. to introduce a fix sequence of compiler options, that on average can bring good results for most applications. Given the peculiarity of the problem, this certainly is not enough.

Exploiting compiler optimizations in application-specific *embedded domains*, where applications are compiled once and then deployed on the market on millions of devices is troublesome. The reason why is firstly because embedded systems are usually designed with tight extra-functional properties constraints. Secondly, the large variety of embedded platforms can not be faced with the average case provided by standard optimization levels, thus custom compiler optimization sequences might lead to substantial benefits in reference to several performance metrics (e.g. execution time, power consumption, memory footprint).

In the *High Performance Computing* (HPC) domain, parallel computer systems are increasingly more complex. Currently, HPC systems offering peak performance of sev-

eral Petaflops have hundreds of thousands of cores to be managed efficiently. Those machines have deep software stack, which has to be exploited by the programmer to tune the program. Moreover, to reduce the power consumption of those systems, advanced hardware and software techniques are applied, such as the usage of GPUs that are highly specialized for regular data parallel computations via simple processing cores and high bandwidth to the graphics memory. Numerous scientific and engineering compute-intensive applications spend most of their execution time in loop nests that are suitable for high-level optimizations. Typical examples include dense linear algebra codes and stencil-based iterative methods [231]. *Polyhedral compilation* is a recent attempt to bring mathematical representation focusing on the loop-nest of the *polyhedral model* including many different tools [26, 31, 116, 143].

In this chapter, we tackle the phase-ordering problem by using predictive modeling. Our predictive model is able to introduce the next-best compiler optimization to be applied given the current status of the application to maximize the performance. The status of the application is defined by a *vector of representative features* that has been collected dynamically and it is independent from the architecture the code is running on. The proposed predictive model has been trained off-line with different permutations of the compiler flags (allowing repetitions and dynamic sequence length). Therefore, the proposed method receives as input the *program features* and it generates the next-best compiler option to maximize the performance of the application. We selected a set of benchmark applications to assess the benefits of the proposed approach and to prove its feasibility.

In this chapter, we propose a predictive modeling methodology to mitigate the phase-ordering problem, In particular, the main contributions are:

- Predictive modeling methodology capable of capturing the correlation between the program features and the compiler optimization at each state.
- The integration of the predictive modeling within a compiler framework. The generated model is trained by means of Machine Learning to focus on the next-immediate best compiler optimization to be applied given the current status of the application for any new previously unobserved program.
- Tackling the phase-ordering problem on utilizing different relative positioning of the sequences of compiler options previously acquired as good sequences from LLVM standard optimization level and explore the design space by using a larger set of compiler flags rather than the individual options.
- Intermediate speedup predictive modeling, capable of iteratively predicting the next-best compiler option using two search heuristics namely to be applied given the current status of the application being optimized.

We apply prediction modeling techniques originally proposed in [60] for selecting the best compiler optimizations. However, the original work was mostly performing predictions on fixed optimization vectors length, while our proposed model is able to iteratively call the function and generate the next-best optimization to be applied, given the current status of the application. This feature is certainly vital for the phase-ordering problem because of: **i)** it opens up to complete the new states towards exploring more regions of interest in the design space and **ii)** it enables us to apply *repetitions* on the

application being optimized. Moreover, the original work was tackling the problem of *selection of best compiler optimization*, while the current work is targeting the substantially harder problem of *phase-ordering*.

The rest of the chapter has been organized as follows. Section 6.3 provides a brief discussion on the related work. In Section 6.4, we introduce the predictive modeling approach to tackle *phase-ordering*. Section 6.5 presents experimental evaluation of the proposed methodology on an Unix-based Intel platform. Finally, Section 5.6 summarizes the outcome of the work and some future paths.

5.3 Related Work

We have extensively presented the literature in the Chapter 2. However, for completeness we re-iterate on notable related work on the very subject here. *Phase-ordering* problem is closely tightened with the *selection of the best compiler options* problem. Therefore, study on the literature can be classified in two main classes: i) *autotuning and iterative compilation approaches* [7, 22, 41, 46] and ii) *applying machine learning to compilation* [20, 179]. Nevertheless, these two approaches have been amalgamated in many ways by exploiting different techniques and methodologies.

There are quite a few studies that have tackled the *phase-ordering* problem. Authors in [130] have applied *Neuro- Evolution for Augmenting Topologies* (NEAT) on Jikes dynamic compiler and come up with sets of good ordering of phases. Other works have approached the problem by exploiting compiler backend optimizations and using statistical tests to reduce code-size [95]. Authors in [129] exhaustively exploring the ordering space at functions' granularity level and evaluate their methodology with search tree algorithms and in [165] the authors have exploited iterative compilation with the information relying on relative passes in previously generated compiler options in the sequence in function level.

Our approach is rather different with respect to the literature, given that we propose a predictive modeling methodology utilizing an independent micro-architecture characterization features for all different *permutations with repetitions* of the compiler options and come up with the prediction of the *next-best compiler option to be applied* on the application given the current status. This is called an intermediate speedup predictor and is able to predict good set of compiler options even with *dynamic lengths* and it is not just limited to fixed vector length. We use previously acquired relative positioning of the promising sequences utilized on LLVM standard optimization levels and treat each of those acquired sequences as one whole. In this case, we could apply phase ordering feasibility on a larger set of compiler options, while generating less design space in the problem.

5.4 The Proposed Methodology

Main goal of the proposed methodology is to identify the feasibility of tackling the phase ordering problem using a predictive modeling methodology. Each application optimized with a unique compiler options sequence is passed through a characterization phase, that generates parametric representation of its dynamic features. A model based on predictive modeling correlates these features to the compiler optimizations applied

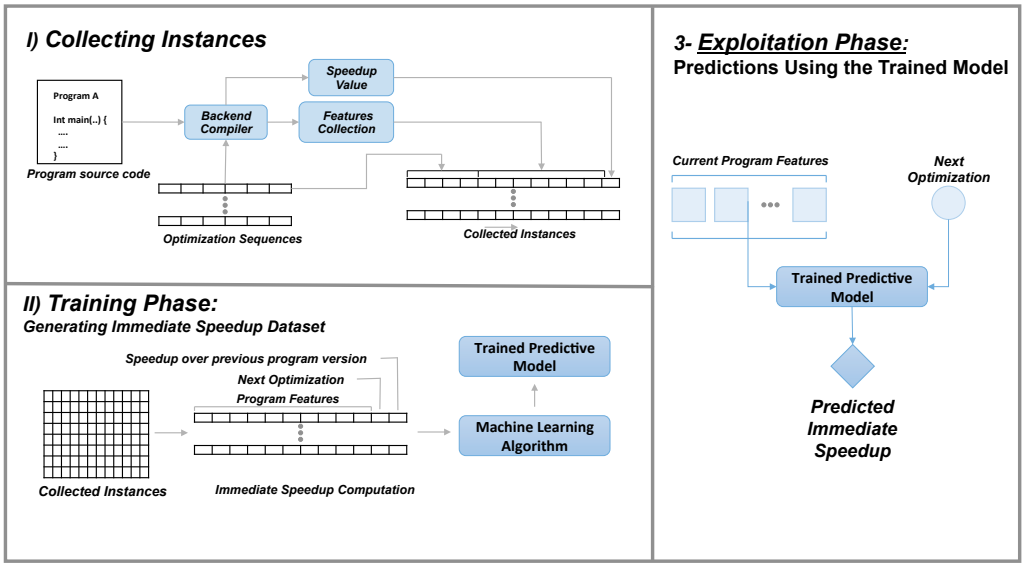


Figure 5.1: *Proposed Predictive Modeling Methodology*

such as to predict the application speedup by using the next-best compiler optimization at each level.

The optimization flow is represented in Figure 6.1. It consists of three main phases: i) *Data collection* where different instances of the application are executed and the application characterizations are fetched with the speedup achieved by utilizing the specific option, ii) *Training phase* where the predictive modeling is learned on the base of a set of training applications and iii) *Exploitation phase*, where new applications are optimized by exploiting the knowledge stored in the trained predictive model. The model is able to predict, given the current program characterization, the immediate speedups associated to each of the compiler optimization under analysis.

During the second and the third phases, an optimization process is necessary to identify the best compiler optimizations to be enabled to get the best performance. This is done for learning purposes during the *training phase* and for optimization purposes during the *exploitation phase*. To implement the optimization process, a Design Space Exploration (DSE) engine has been used. This DSE engine compiles, executes and measures application performance by enabling and disabling different permutations with repetitions of compiler optimizations.

In our approach the DoE is obtained by *exhaustive exploration* including all permutations with repetitions of compiler configurations (during the training phase as shown in Figure 6.1) either by means of the *whole sequence* at once or *the current compiler optimization* that has been applied to the previous state. On the other side, *exploitation phase*, they are generated by means of predicting the *whole sequence* at once or as *the next-best configuration* to be applied given the current status. These two different techniques will be elaborated more in Sections 5.4.3 and 5.4.3 respectively.

5.4.1 Compiler Phase-ordering Problem

To formulate the phase-ordering problem, first we come up with the *selection of the best compiler sequence* problem. Let us define a Boolean vector $\boldsymbol{o}$ whose elements o_i are the different compiler optimizations. Each optimization o_i can be either enabled $o_i = 1$ or disabled $o_i = 0$. A compiler optimization sequence to be *selected* is represented by the vector $\boldsymbol{o}$ belongs to the n dimensional Boolean space of:

$$\mathcal{O}selection = \{0, 1\}^n \tag{5.1}$$

For the application a_i being optimized and n represents the number of compiler optimizations under study. Therefore, the mentioned research problem consists of an exponential space as its upper-bound. Having n = 10, drive us to a total space (2^n) up to $|\mathcal{O}selection|$ = 1024 options to select among per interested target application a_i to be optimized and this number itself would be multiplied by different applications A = $a_0...a_N$ under study.

Coming back to the *phase-ordering* problem, let us define a Boolean vector $\boldsymbol{o}$ whose elements o_i are the different compiler optimizations. A *Phase-ordering* compiler optimization sequence represented by the vector $\boldsymbol{o}$ belongs to the n dimensional factorial space $|\mathcal{O}phases| = n!$, where n represents the number of compiler optimizations under study. However, the mentioned bound is for a simplified *phase-ordering* problem given fixed vector length without repetitions. Enabling repetitions and dynamic length will expand the design space size to:

$$|\mathcal{O}phases\_repetition| = \sum_{i=0}^{m} n^i \tag{5.2}$$

Where n is the number of interesting optimizations under study and m is the maximum desired length for the optimization sequence length. In this case, assuming the same n and m equal to 10, $|\mathcal{O}phases\_repetition|$ will drive up to more than 11 ***Billion*** different configurations to select per each application. [1]

The o_i in this work consists more than one single compiler optimizations. These set of optimizations are derived from the LLVM standard optimization level. Reader can refer to the specific o_i in Section 6.5 Table 6.2. We treat each of the whole sequence (which being referred to as *genes*) as a discrete variable, so that each optimization o_i can be either enabled $o_i = 1$ or disabled $o_i = 0$ and enabling the o_i will enable all its contained sub-optimizations respectively.

5.4.2 Application Characterization

In this work, we use PIN [148] based *dynamic profiling* framework to analyze the behavior of the different applications at execution time. In particular, the selected profiling framework provides a high level *Micro-architectural Independent Characterization of Applications* (MICA) [102] suitable for characterizing applications in a cross-platform manner. Furthermore, there is no static syntactic analysis, but the framework is based solely on MICA profiling.

[1] The problem of phase-ordering does not have a definite upper-bound in the case of having repetitions with unbounded *max_length($\mathcal{O}$)*.

In our experimental setup, an application is compiled and profiled on an x86 host processor, while the target architecture where the application executes (i.e. the architecture for which the application shall be optimized) could be a different platform thanks to the high level abstraction of the application characterization carried out with MICA, so as we can easily change the target architecture without the need of changing the profiling infrastructure.

The MICA framework reports data about *instruction type*, *memory and register access pattern*, potential *instruction level parallelism* and a dynamic control flow analysis in terms of *branch predictability*. Overall, the MICA framework characterizes an application in reference to 99 different metrics (or features). However, many of these 99 features are strongly correlated (e.g. the number of memory reads with stride smaller than 1K is bounded by the number of reads with stride smaller than 2K). Furthermore, generating a *predictive model* is a process whose time complexity grows up with the number of parameters in use. It is too expensive to include all the 99 features in the model. Given the goal of speeding up the construction, we applied *Principal Component Analysis* (PCA) to reduce the number of parameters to characterize the application. PCA is a technique to transform a set of correlated parameters (application features) into a set of orthogonal (i.e. uncorrelated) principal components. The PCA transformation aims at sorting the principal components by descending order based on their variance [111]. For instance, the first components include the most of the input data variability, i.e. they represent the most of the information contained in the input data. To reduce the number of input features, while keeping most of the information contained in the input data, it is simply needed to use the first k principal components as suggested in [102]. In particular, we set $k = 10$ to trade off the information stored in the application characterization and the time required to train the *predictive modeling*.

5.4.3 Speedup Prediction Modeling

The general formulation of the optimization problem is to construct a function that takes as input the features of the *current status* of a program being optimized to generate as output the *the next-best optimization* to be applied that maximize the *immediate predicted speedup*. We used the prediction model originally proposed in [60]. However, the original work was mostly performing predictions on fixed optimization vectors length, while our proposed model is able to iteratively call the function and generate the next-best optimization to be applied, given the current status of the application. This feature is certainly vital for the phase-ordering problem because of: **i)** it opens up to complete the new states towards exploring more regions of interest in the design space and **ii)** it enables us to apply *repetitions* on the application being optimized.

An application is parametrically represented by the vector $\boldsymbol{\rho}$, whose elements α_i are the first k principal components of its dynamic profiling features. Elements α_i in the vector $\boldsymbol{\rho}$ generally belong to the continuous domain. The optimal compiler optimization sequence $\bar{\boldsymbol{o}} \in \mathcal{O}$ that maximizes the performance of an application is generally unknown. However it is known that the effects of a compiler optimization o_i might depend on whether or not another optimization o_j is applied.

Our models predict optimizations to apply to unseen programs that were not used in training the model. To this purpose, we need to feed as input a characterization of the unseen program. The model is able to predict the speedup of each possible

optimization set $\mathcal{O}$ in our predictive optimization space, given the characteristics of the unseen program. We order the predicted speedups to determine which optimization set is predicted best, and we apply the predicted best optimization set(s) to the unseen program. In the experimental Section 6.5, we use a leave-one-benchmark-out cross-validation procedure for evaluating the models. The proposed predictive modeling is going to introduce two different heuristics on predictive modeling.

DFS Search Heuristic

Depth-First Search (DFS) and its optimized version Depth-First Iterative Deepening [123] are well-known tree traversing algorithms. DFS starts at the root and explores as far as possible along each branch before backtracking. Adapting the heuristic on the current problem, we propose to start from an empty optimization sequence $\boldsymbol{o}_o$, Considering sequence $\boldsymbol{o}_i$, for each of the possible optimizations we predict the immediate speedup δ_i derived from applying o_i after $\boldsymbol{o}_i$ in the compilation process. The *intermediate speedup* is computed as:

$$e_i = Exec\_time(\boldsymbol{o}_i)/Exec\_time(\boldsymbol{o}_j) \tag{5.3}$$

where e_i is the ratio between the execution time of the program compiled using $\boldsymbol{o}_i$ and the execution time of the version of the program generated using $\boldsymbol{o}_j$. We define $\boldsymbol{o}_j$ as $\boldsymbol{o}_i$ followed by o_i.

Then, we order the possible optimizations by the value of the predicted immediate speedup δ. If none of the optimizations o_i to be explored has an associated immediate speedup δ_i greater than 1, we choose $\boldsymbol{o}_i$ as the next sequence to test, and we use it to compile the program and measure corresponding execution time. Otherwise, we repeat the same exploration process starting from sequence $\boldsymbol{o}_j$, that is $\boldsymbol{o}_i$ followed by the still unexplored o_i maximizing predicted immediate speedup δ_i.

Once all the possible optimizations o have been explored, and the original sequence $\boldsymbol{o}_i$ tested, the algorithm backtracks to the previous considered sequence $\boldsymbol{o}_k$, that is, $\boldsymbol{o}_i$ without its last optimization. If a sequence $\boldsymbol{o}_i$ has reached the maximum optimization sequence length N to be considered, we stop applying further optimizations after it. We then evaluate $\boldsymbol{o}_i$ it and backtrack to the previous node. The algorithm explores the complete optimization space using this policy and terminates when reaching the backtracking point for the initial empty sequence $\boldsymbol{o}_o$.

Exhaustive Search Heuristic

The second iterative approach is tackling the exploration with exhaustive search. A model trained using machine learning techniques produces speedup predictions for all the configurations in the complete considered sequence space. Ordered by decreasing predicted speedup values, the sequences are then applied to the program, and their actual speedup is measured. This approach has been successfully used in selection of the best compiler sequences problem, but lacks of applicability in the phase-ordering problem, given the complexity increase of the configuration space.

In our specific case, we modify this methodology to adapt it to our application scenario. In particular, as described previously at Section 5.4.3 Equation 5.4, the model we trained is able to predict only immediate speedups δ (i.e. the speedup of $\boldsymbol{o}_j$ over

Table 5.1: *Applications used in this work*

Applications	Description
automotive_bitcount	Bit counter
automotive_qsort1	Quick sort
automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corner
automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Corner
network_dijkstra	Dijkstra's algorithm
network_patricia	Patricia Trie data structure

$\boldsymbol{o}_i$, where $\boldsymbol{o}_j$ is the optimization sequence obtained by applying o_i after $\boldsymbol{o}_i$. We are able to predict the actual speedup of optimization sequence $\boldsymbol{o}$ by *multiplying* the immediate speedups δ_i predicted at each optimization $o_i \in \boldsymbol{o}$:

$$\boldsymbol{o}_o : \prod_{o_i \in o} \delta(o_i) \tag{5.4}$$

where o_i is each individual optimization options to be explored. The proposed exhaustive exploration computes the immediate speedups starting from the initial empty sequence $\boldsymbol{o}_o$ to the complete optimization space. In this way, we are able to predict the speedup of every optimization sequence $\boldsymbol{o} \in \mathcal{O}$ and map our system to the classic exhaustive predictions methodology we described.

5.5 Experimental Evaluation

In this section, we assess the proposed methodology of the *immediate next-best predictive modeling* on quad-core Intel-Xeon E1607 running at 3.00 GHZ. We have used LLVM compilation tool v3.8 within our framework. A subset of cBench benchmark suite [86] consisting of six different applications has been integrated within the framework. The list of selected applications has been reported in Table 6.1. Table 6.2 presents the utilized sets of LLVM optimizations categories in 4 different genes. The utilized passes are part of the LLVM standard optimization levels and we exploited the phase-ordering scenario having them relatively fixed internally, while altering the whole sequence externally at each phase. In this mode, we speculated that we could explore more interesting regions of the design space and reaching higher potential speedups accordingly. The utilized 4 different genes, consists of 30 compiler optimizations in total and 13 unique optimizations. One of the features of the proposed methodology is that it supports repetitions in the compiler design space. Table 5.3 represents the maximum achievable speedup enabling repetition feature on every individual applications. We observe that excluding two applications that coincidentally had their gene having the best achievable speedup without repetitions (thus enabling repetition was converging to the very same result), the other four are gaining benefits from enabling this features. We used harmonic mean to better report the average of the speedups rather than arithmetic mean here [99].

The application execution time is estimated by using the *Linux-Perf* tool. Execution time required to process a given data set by a compiled application binary is estimated by averaging four different executions. In order to implement dimension reduction

Table 5.2: *Compiler optimizations under analysis: Derived from LLVM-Opt*

Gene	Abbreviation	Relative Positioning of the Optimizations
A	*domtreeRULE*	-domtree -memdep -dse -adce -instcombine -simplifycfg -domtree -loops -loop-simplify -lcssa -branch-prob
B	*simplifycfgRULE*	-simplifycfg -reassociate -domtree -loops -loop-simplify
C	*memdepRULE*	-memdep -domtree -memdep -gvn -memdep -memcpyopt -sccp
D	*loopsRule*	-loops -loop-simplify -lcssa -branch-prob -block-freq -scalar-evolution -loop-vectorize
Total Number of Optimizations Under Analysis		**30**
Unique Number of Optimizations Under Analysis		**13**

technique, we used PCA having set PCs to 10 in this work. The PCA components have been computed by using the MICA features collected from each application run, normalized by standard deviation across all data sets. Our exploration experiments have been generated by using the data previously collected offline.

WEKA Machine Learning tool [94] has been integrated in our framework to exploit predictive modeling algorithms. More in detail, in this work, we assessed the proposed methodology with *Linear Regression* (LR) Machine Learning algorithm activating the *M5* attribute selection method with default ridge parameter. Experimental results has been carried out by means of *leave-one-out cross-validation*. Given a new unseen application in the training set, the current program feature already obtained offline will impose a bias on the trained model and the predictive modeling will be able to predict the *next-best optimization* to be applied to maximize the immediate speedup in a greedy manner. As mentioned in Equation 6.1, given the complexity of the problem, we assessed the feasibility of our proposed approach with four different sequences of compiler options with a total of 30 compiler options (13 unique optimizations) to be used in the design space. Generating different permutations with repetitions and enabling dynamic sequence length led to 341 different variations and 2046 for all considered applications.

The results obtained by exploring the phase-ordering space with the two proposed heuristics is reported in Figure 5.2. It shows that the revised DFS search approach, namely an *Iterative Deepening First Search* policy (based on a greedy Depth First Search (DFS) heuristic) is doing slightly better with respect to the *exhaustive search heuristic* presented in Section 5.4.3. We define the *actual speedup* line as as the speedup observed from running the application using the compiler optimizatioj prediction of the machine learning model. Table 5.4 is presenting the quantitative values of utilizing the two predictive modeling algorithms within our framework. We evaluated our *itera-*

Table 5.3: *Maximum Achievable Performance Enabling Repetition*

Application	Best Opt Found W.Rep	Best Opt Found WOut.Rep	Speedup %
automotive-susan-c	CDD	CD	9.34
network-patricia	CDCD	AD	60.37
automotive-qsort1	CBA	CBA	-
automotive-bitcount	CCBB	CDB	2.41
network-dijkstra	ACAB	AD	36.59
automotive-susan-e	CD	CD	-
Harmonic Mean	-	-	7.68

Table 5.4: *Average speedup of the one-shot prediction for both approaches w.r.t LLVM baseline*

Application	Greedy DFS	Exhaustive Predictions
automotive-susan-c	0.9808	0.9658
network-patricia	1.0069	1.002
automotive-qsort1	1.1255	0.9670
automotive-bitcount	1.0848	1.1506
network-dijkstra	0.9988	0.9988
automotive-susan-e	1.0617	1.1015
Average	**1.0431**	**1.0242**

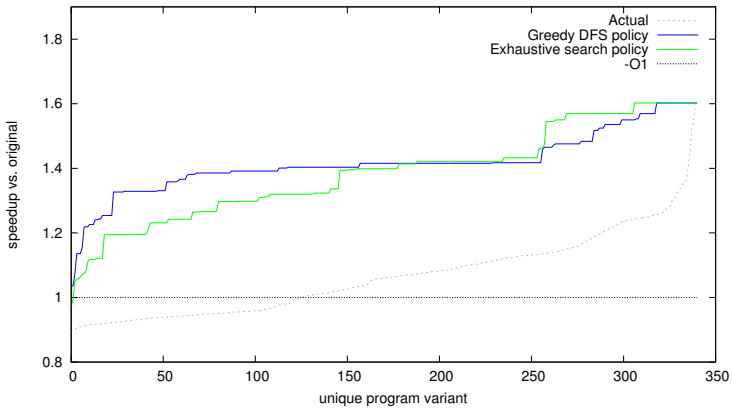


Figure 5.2: *Average Speedup of the proposed methodology among all the applications*

tive greedy approaches with the classic 1-shot predictive speedup approach mentioned in [40, 60, 178] and the results reported in Table 5.4 are the average output of the one-shot approach per application. One-shot approach is extracting the prediction by means of one-extraction only and observe its speedup gain. In average, these two search algorithms demonstrated respectively 4% and 2% performance speedup over LLVM default performance.

The graph in Figure 5.2 demonstrates that the performance of the greedy exploration policy in early generation of the prediction is better than the exhaustive search methodology for the selected benchmarks. The horizontal axis represents different variations of the applications. This is rather interesting because in the phase-ordering problem the cardinality of the optimization sequence space $\mathcal{O}$ is too huge for an exhaustive search

policy to be applied. On average, by traversing 15% of the compiler design space, we can reach up to 80% of the best found options in the design space.

5.6 Conclusions and Future Work

This chapter presents a method based on predictive modeling to select the next-best immediate compiler option to be applied to maximize the application performance. Experimental results exploiting two different search heuristics on the selected set of benchmarks demonstrated respectively 4% and 2% performance speedup with respect to the default LLVM compiler framework. Future work will focus on building more-accurate predictive models capable of capturing the intra/inter-correlation effects of different compiler options.

5.7 Dissemination of The Chapter

The excerpt of this chapter has been published in ACM PARMA-DITAM - Workshop of HiPEAC, 2016 held in Prague, Czech Republic, p7-12 under the title *Predictive Modeling Methodology for Compiler Phase-Ordering*. Refer to the Chapter 7.2 for the list of publications of my PhD dissertation.

CHAPTER 6

The Phase-ordering Problem: A Complete Sequence Prediction Approach

6.1 Summary

Recent compilers offer a vast number of multilayered optimizations, capable of targeting different code segments of an application. Choosing among these optimizations can significantly impact the performance of the code being optimized. The selection of the right set of compiler optimizations for a particular code segment is a very hard problem, but finding the best ordering of these optimizations adds further complexity. In fact, finding the best ordering is a long standing problem in compilation research called the phase-ordering problem. The traditional approach of constructing compiler heuristics to solve this problem simply can not cope with the enormous complexity of choosing the right ordering of optimizations for every code segment in an application.

This chapter proposes MiCOMP: Mitigating the Compiler Phase-ordering problem using optimization sub-sequences and machine learning, an autotuning framework to effectively mitigate the compiler phase-ordering problem based on machine-learning techniques. The idea is to cluster the optimization passes of the LLVM O3 setting into different clusters to predict the speedup of the complete-sequence of all the optimization clusters. The predictive model uses (i) a platform-independent dynamic features, (ii) an encoded version of the compiler sequence and (iii) an exploration heuristic to tackle the problem.

Experimental results using the LLVM compiler framework and the cBench suite show the effectiveness of the encoding technique to application-based reordering of passes while using a number of predictive models. We perform statistical analysis on the prediction space and compared against (i) standard optimization levels O2 and O3, (ii) random iterative compilation, and (iii) two recent non-iterative approaches. We demonstrate that our proposed methodology outperforms the performance of -O1, -O2, and -O3 optimization levels in just a few iterations, reaching an average performance speedup of 1.26 (up to 1.51) on the cBench benchmark suite.

6.2 Introduction

Compiler developers typically design optimization passes in order to transform each code segment of a program to produce an optimized version of an application. The optimizations can be applied at different stages of the compilation process. Optimizing source code by hand is a tedious task and therefore compiler optimizations are provided to automatically transform code. However, these code optimizations are programming language, application, and architecture dependent. Additionally, the word optimization is a misnomer and there is no guarantee the transformed code will perform better than the original version. In fact, aggressive optimizations can degrade the performance of the code they are applied to. Understanding the behavior of the optimizations and the actual effect on the source-code and the interaction of the optimizations with each other are complex modeling problems. The problem is particularly difficult because compiler developers have to deal with hundreds of different optimizations that can be applied during the different compilation phases and this creates the phase-ordering problem. The phase-ordering problem has been an open-problem in the field of compiler research for many decades. The inability of researchers to solve the phase-ordering problem has led to advances in the more simple problem of selecting the right set of optimizations, but even this problem has yet to be solved [28, 46].

This process of selecting the right optimizations for each code segment is typically done manually and the sequence of optimizations is constructed with little insight into the interaction between the preceding compiler optimizations in the sequence. The task of constructing heuristics to select the right sequence of compiler optimizations is infeasible given the ever growing number of compiler optimizations being integrated into compiler frameworks. As an example, GCC has more than 200 compiler passes, and LLVM-clang and LLVM-opt both have more than 100 transformations each. Additionally, these optimizations are applied at very different phases of the compilation, including analysis passes and loop-nest passes. Most optimization flags are turned off by default and compiler developers rely on software developers to know, which optimizations will be beneficial for their code. Compiler developers provide standard optimization levels, e.g. -O1, -O2, -Os, etc. to introduce a fixed-sequence of compiler optimizations that on average bring good performance on a set of benchmarks the compiler developers tested these optimization levels on.

Finding the best ordering of compiler optimizations can have substantial benefits for performance metrics such as execution time, power consumption and code-size. To this end, using predefined optimizations usually is not good enough to bring the best achievable application-specific performance. In this chapter, we propose a framework in order to mitigate the complexity of the phase-ordering problem. So far, there are two potential techniques we could use to predict good optimization orders for code being optimized:

- (i) *Intermediate Sequence Prediction*: This technique uses a model to predict the current best optimization (from a given set of optimizations) that should be applied based on the characteristics of code in its present state. [19, 130].
- (ii) *Complete Sequence Prediction*: This technique uses a model to predict the complete sequence of optimizations that needs to be applied to the code just by looking at characteristics of the [41, 177–179].

The framework proposed in this chapter, MiCOMP, falls under the second category. It uses predictive models on complete optimization sequences, rather than individual optimizations. We characterize applications as a vector of dynamic features that are independent from the target architecture. Predicting the complete optimization sequence to apply to a piece of code, i.e., complete sequence prediction, has the benefit of only requiring a single-round of feature collection of the code before any optimizations are applied to it. In order to use classic machine learning algorithms with the phase-ordering problem, we adapt an encoding scheme to transform variable-length vectors of optimizations into fixed-length vectors. Our prediction models are trained offline and program features and different compiler configurations are fed as inputs. As outputs, a prediction model predicts the speedup without the need to actually run the code on the target architecture. The dynamic characterization is independent from the architecture the code is running, thus it brings portability among different architectures. Additionally, we define exploration heuristics to find the best models in the shortest time. By time, we refer to the minimum number of predictions from the model to obtain the best version of the code being optimized. The heuristic is based on Adjusted Cosine Similarity [200] to correlate different configurations of optimizations with their corresponding predicted speedups across all the training data. A recommendation algorithm enables

us to explore only a fraction of the configuration space to reach the best speedups rather than a simple sorting/ranking [41, 177–179]. In our experimental results, we show that our technique can outperform LLVM's highest optimization level of `-O3` by just a few predictions. We also show competitive and quantitative comparisons with respect to state-of-the-art iterative and non-iterative models. We selected a variety of applications from the Ctuning Cbench benchmark [86] to assess and evaluate the benefits of the proposed approach and to prove its feasibility. The main contributions of the proposed approach are as follows:

- An independent predictive-modeling framework, capable of capturing the correlation between different compiler optimizations and their predicted speedup without having to run optimized code variants on the target platform. Our autotuning framework can be paired with any desired predictive models.
- Dynamically reordering the optimizations within the LLVM optimization level `-O3`. We have clustered different compiler optimizations, all taken from LLVM's `O3` into 5 different groups. The order of optimizations within a group is internally fixed but the ordering of the groups can be altered. In this work, these groups are called sub-sequences and we exploit the phase-ordering by using these sub-sequences rather than the individual optimizations. By starting from no optimizations (as the baseline) and exploring different orderings of the sub-sequences using the same optimizations available to `-O3`, we outperformed `-O3`.
- Adapting a simple mapping technique to encode an optimization sequence into a bit string. The proposed technique allows us to apply traditional machine learning algorithms as they are mostly designed to cope with both fixed-length feature vectors.
- Adapting a Recommender System (RS) approach on the prediction space to use dynamic information. We show this can boost the exploration and help to obtain better speedups.

The rest of the chapter organized as follows: Section 6.3 presents related work. Section 6.4 introduces our proposed methodology including all its components. In Section 6.5, we present our experimental results and evaluate the results by means of several comparisons in the Section 6.6. We conclude this chapter with future work and the conclusion.

6.3 Related Work

As we discussed in the previous chapter, literature on the phase-ordering problem is closely related to the problem of selecting the best set of compiler optimizations in a fixed ordering. Recent literature can be classified into two main classes: (i) autotuning and iterative compilation approaches and (ii) applying machine learning to the problem of optimization selection. (Readers can refer to Chapter 2 for the complete survey on the literature.)

Autotuning addresses automatic code-generation and optimization by using different scenarios and architectures. It involves building techniques for automatic optimization

of different parameters in order to maximize or minimize the satisfaction of an objective function. One strategy in autotuning consists of coupling the approach with random generation of code-variants at each run. This technique can generally improve application performance in reference to static-handcrafted compiler optimization sequences [7]. Given the complexity of the iterative compilation problem [28], it has been shown that applying compiler optimization sequences at random can be as good as using other algorithms such as Genetic algorithms or Simulated Annealing to choose which optimizations to apply [7, 41, 46]. Other authors [18, 22] explored compiler Design Space Exploration (DSE) techniques jointly with architectural DSE for VLIW architectures.

Applying *machine learning* to the problem of selecting the best compiler optimizations has been extensively investigated by many researchers in the past. Proposed methodologies [36, 52, 119, 159, 160, 213] were among the first notable works introducing the use of machine learning to solve compilation problems. Recent related work [17,20,177,178] also tackled the problem of selecting the best compiler optimizations to apply by utilizing Bayesian Networks with an application-independent characterization technique, predictive modeling with dynamic characterization, and predictive modeling with compiler representations (Intermediate Representation (IR)) . There have been different objective functions used with machine learning on the problem: i) A *speedup predictor* takes as input both the characterization of the program being compiled and an optimization sequence, and it predicts as output the speedup when applying that optimization sequence relative to a default optimization setting. [41, 177, 178] ii) A *sequence predictor* characterizes a program being compiled and uses it as input to a model, and the model predicts a probability distribution of optimizations to apply to that program. [7, 20, 191]. iii) A *tournament predictor* [179]) takes as input a triple corresponding to the characterization of the program and two optimization sequences. This model predicts whether the speedup after applying the the first optimization sequence will be more or less than speedup if applying the second optimization sequence.

The phase-ordering problem has not yet been studied in-depth. However, there are a few notable published research studies that attempted to solve the problem. Kulkarni and Cavazos [130] have applied *Neuro-Evolution for Augmenting Topologies* (NEAT) in the Java JikesRVM compiler to phase-ordering by using intermediate sequence prediction. They build prediction models that use as input features of the current state of the transformed source-code and define certain stop-condition rules to complete the final predicted sequence at each iteration. They used source-code features and the Java JikesRVM JIT compiler to experimentally evaluate their approach. In contrast, we tackle the problem using predictive modeling and dynamic independent-characterization of the applications and our proposed methodology enables us to predict the full-sequence in one-shot.

Matrins et al., tackled the problem of phase-ordering by a DSE approach that uses a clustering-based selection method for grouping functions with similarities and exploration of a reduced search space resulting from the combination of optimizations previously suggested for the functions in each group [151]. Authors used DNA encoding where program elements (e.g., operators and loops in function granularity) are encoded in a sequence of symbols, and followed by calculating the distance matrix and a tree construction of the optimization set. Consequently, they applied the compiler

optimization passes already included in the DSE to measure the reduction in the total exploration time of the search space such as Genetic algorithm. Our proposed approach on the other hand is mainly different, as we mitigate the phase-ordering problem by inducing a prediction model rather than a design space exploration scheme. Once our model is trained, it can be further used for any number of applications under analysis to induce a prediction inexpensively and we believe it will bring scalability in autotuning compilers.

Other related work has approached the problem by exhaustively exploring the optimization ordering space at the granularity of functions [129]. The exhaustive enumeration these authors proposed, constructs probabilities of enabling/disabling interactions between different optimization sequences, but these probabilities are not specific to any program. Ashouri et al. introduced an approach that uses predictive modeling to construct an intermediate sequence of optimizations for code being compiled [19]. Other related work used iterative Design Space Exploration (DSE) and clustering-based approaches to down-sample and cluster the available optimizations targeting performance gain and power reduction [164–166].

Our approach, MiCOMP, is significantly different compared with those mentioned in the literature. Our work mostly resembles the approach of Park et al. [178, 179]. However, our techniques tackle the significantly harder problem of the phase-ordering. We introduce a mapping function that encodes an optimization sequence into a bit string. It preserves the ordering and the repetition of the optimizations. At the same time, the proposed work is able to predict the complete optimization sequence to apply to the unoptimized code, rather than predicting the best optimization to apply to the current state of the optimized code [19, 130]. An intermediate sequence approach needs multiple profiling of the application being optimized (based on the characteristics of code in its present state) while we need to profile just once, both in the offline-training and the online-prediction. We use dynamic architecture independent features to feed into our model. Moreover, we used clustering over all passes in LLVM's `-O3`, that tended to perform well, to significantly outperform the single optimization sequence performed by `-O3` itself. We do that by re-ordering these sub-sequences automatically based on the type of the application under optimization. To summarize, the proposed work is the first approach that uses machine-learning based techniques on the phase-ordering problem to predict the complete sequence of optimizations. In Section 6.5, we improve the machine-learning model through Recommender System techniques and assess the experimental results we obtain against the state-of-the-art phase-ordering approaches.

6.4 The Proposed Methodology

Compilers typically ship with standard optimization levels (e.g. `-O2`, `-O3` and `-Ofast`) each tuned at the compiler factory to obtain a certain level of performance on a standard set of benchmarks. These optimization levels do not always translate to good performance on other applications. The main objective of the proposed methodology is to introduce a compiler autotuning framework, which is able to dynamically reorder the compiler passes within LLVM's optimization level `-O3` [1], to achieve the

[1] Additionally, LLVM's `Clang` has an optimization level `-O4`, but after inspecting the passes in this level and the `-O3`, we believe both optimization levels are equivalent.

maximum speedup for the applications being optimized. We found that if we could reorder sub-sequences of optimizations that tended to perform well, we could significantly outperform the single optimization sequence performed by `-O3`. This process should be customized based on the features of the application under analysis. To mitigate the phase-ordering problem, a model has to be constructed in such a way that it can correlate the effect of using different compiler sequences and the corresponding achievable speedup. MiCOMP uses such a model, and it can (i) recommend good sequences of optimizations that maximize an application's performance and (ii) these good-sequences are recommended with very few predictions.

There are certain limitations facing the ever increasing complexity of the problem. The phase-ordering problem is also complicated by allowing the possibility of variable-length compiler sequences. State-of-the-art approaches for selecting the right set of optimizations used fixed length feature-vectors [41, 178, 179]. Therefore, in order to tackle the phase-ordering problem, we propose to encode the phase-ordering space into a conventional fixed-length feature vector space to be able to apply traditional machine learning algorithms, specifically Regression models that tends to fit the data to relate Y to a function of X and β, where X and β are the vector of the application features and the compiler sequence, respectively. We provide a simple encoding-scheme with mathematical deduction in Section 6.4.3. During the prediction phase, MiCOMP proposes an iterative process in which different solutions are explored by evaluating different optimization sequences with the potential of leading to higher speedups. We predict optimization sequences that will perform well against using state-of-the-art *ranking* [178, 179] techniques.

Figure 6.1 illustrates the two main phases of MiCOMP: (i) *offline training* and (ii) *online prediction*.

The *offline training phase* is used to learn about the effects of compiler optimizations when compiling an application. In particular, this phase is used to induce a prediction model considering application features and applied optimizations (including order and repetitions). This phases is performed once for each compiler and the model is built on a set of representative application. In this phase, each application is passed through a single round of feature collection to to extract an application's characteristics. A dynamic profiler is used to generate a representation of the program in terms of its features. Since a very large set of features is extracted for each application, we apply a dimension-reduction technique to reduce the number of features that is fed as input to the prediction model (e.g. Principal Component Analysis (PCA) [111]). This speeds up the learning during the model construction process. Application-profiling and dimension-reduction techniques are extensively described in Section 6.4.1. Next, an application is compiled with different configurations of compiler optimizations, executed and profiled in terms of speedup with respect to LLVM 's `-O3`. The speedup values together with the reduced program features and an encoded version of the used compiler optimizations (characterized by a fixed-length binary output, see Section 6.4.3) are fed to a machine learning algorithm to induce the speedup predictor (see Section 6.4.4). This model can then used during the online phase.

The *online prediction phase*, is used every time a new application is optimized. To this end, we use the same feature extraction and dimension reduction techniques described in the offline training phase. The collected features are used to query the

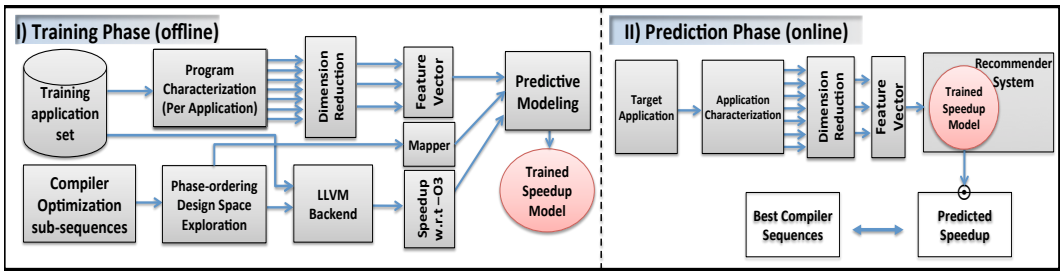


Figure 6.1: *Proposed framework. (i) offline-training phase which is done once and (ii) online-prediction phase for optimizing new unseen applications*

speedup prediction model to predict the best set of compiler sequences to apply to an application. The goal of our method is to discover the fewest number of predictions that will be needed to obtain the optimization sequence that gives the best speedup possible. Thus, MiCOMP has been coupled with a heuristic derived from the field of Recommender Systems (see Section 6.4.5). This technique is used to obtain a predicted set of optimization sequences where each sequence is as diverse as possible to the other sequences in the set, thus guaranteeing coverage of a large part of the optimization configuration space, consequently obtaining a set of optimization sequences that are robust to model inaccuracies.

6.4.1 Application Characterization

In this work, we used a PIN-based [148] dynamic instrumentation framework to analyze and characterize the behavior of applications at execution-time. In particular, our framework provides a high level Micro-architectural Independent Characterization of Applications (MICA) [102] suitable for characterizing applications in a target architecture agnostic manner. There is no static syntactic analysis, but the framework is solely based on dynamic MICA profiling.

In our experimental setup, an application is compiled and profiled on an Intel XEON machine, while the target architecture where the application will eventually execute (i.e. the architecture for which the application is being optimized) will be a different platform. That is, the machine learning model will be fed a high level abstraction of the application characterization carried out with MICA. This allows us to easily change the target architecture without the need of replicating the profiling infrastructure.

The MICA framework reports information about instruction types, memory and register access pattern, potential instruction level parallelism and a dynamic control flow analysis in terms of branch predictability. Overall, the MICA framework characterizes an application in reference to 99 different metrics (or *features*). Many of these 99 features are strongly correlated (e.g. the number of memory reads with stride smaller than 1K is bounded by the number of reads with stride smaller than 2K). is a process whose time complexity grows up with the number of parameters in use and at the same time, including all feature might lead to higher noise. To significantly improve the speed of model construction, we applied a dimension reduction by using Principal Component Analysis (PCA) [111] to reduce the number of features used to characterize an application. PCA is a technique to transform a set of correlated features into a set of orthogonal, i.e., uncorrelated principal components. The PCA transformation sorts the

principal components by descending order based on their variance [111]. For instance, the first principal component includes the most input data variability, i.e., this component represents most of the information contained in the input data. To reduce the number of input features, while keeping most of the information contained in the input data, one simply needs to use the first k principal components as suggested in in previous work [102]. In particular, we set $k = 5$, which captures more than 98% of the overall variance across all training sets.

6.4.2 Constructing Compiler Sub-sequences

In this section, we briefly explain our novel idea behind clustering certain compiler optimizations as *sub-sequences*. A phase-ordering optimization sequence represented by the vector $\boldsymbol{o}$ belongs to the n dimensional factorial space $|\Omega_{phases}| = n!$, where n represents the number of compiler optimizations under study. However, the mentioned bound is for a simplified phase-ordering problem having a fixed length optimization sequence length and no repetitive application of optimizations. Allowing optimizations to be repeatedly applied and a variable length sequence of optimizations will expand the problem space to:

$$|\Omega_{phases\_repetition}| = \sum_{i=0}^{m} n^i \tag{6.1}$$

Where n is the number of optimizations under study and m is the maximum desired length for the optimization sequence. Even for reasonable values for n and m, the entire search space is enormous. For example, assuming n and m are both equal to 10, this leads to an optimization search space of more than 11 billion different optimization sequences to select from for each piece of code being optimized [19] [2].

The Optimization Dependence Graph

Mitigating the phase-ordering problem with previous approaches is not practical due to the large number of different possible optimization sequences to select from each piece of code being optimized. MiCOMP, proposes to group optimizations into clusters of sub-sequences that are known to perform well, which reduces the size of the search space to explore and thus introduces scalability. There are 157 compiler passes in LLVM optimization level `-O3` (more than 60 unique compiler passes) and selecting the most promising sub-sequences from these optimizations can positively affect the autotuning process. Among all these 157 compiler passes, some are analysis passes (i.e. basicaa, memdep, etc) which do not transform the code directly, but instead provide analysis information to other compiler passes that follow them. The rest are transformation passes, i.e., Aggressive Dead Code Elimination (adce), Loop Invariant Code Motion (licm), loop-rotate, etc., which perform optimizations on the code [3].

In this chapter, we introduce the idea of clustering sub-sequences of all the passes available to the optimization level `-O3` and adapt prediction models to order these subsequences in ways that improve the performance of a particular application. We show that this technique can improve the performance of an application over using `-O3` by evaluating a few predicted orderings of the sub-sequences of optimizations.

[2] The problem of phase-ordering does not have deterministic upper-bound in the case of unbounded length.
[3] http://llvm.org/docs/Passes.html

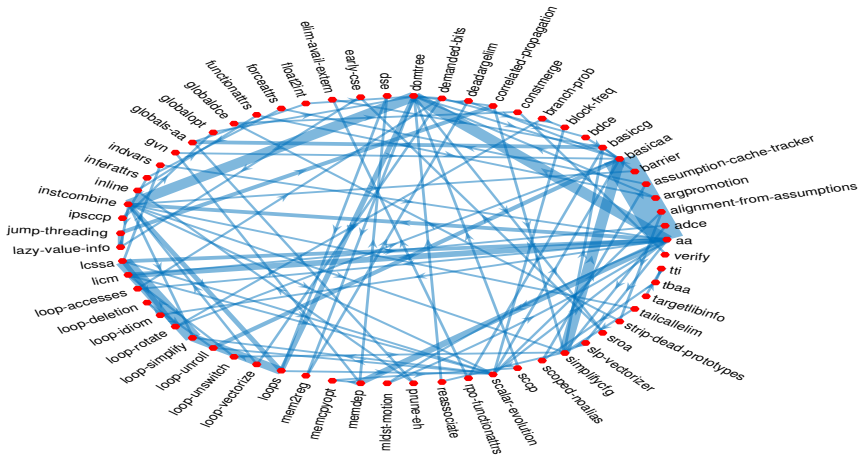


Figure 6.2: *Generated directed graph for LLVM's -O3. Each node in the graph represents an optimization pass. The edge thickness depicts the strengths in the connection between two nodes.*

Let $\boldsymbol{o} = \{o_1, \ldots, o_N\}$ be the set of all transformation passes from LLVM optimization level `-O3`. We can represent the optimizations in optimization level `-O3`, with a directed graph of $G = (V, E)$ where V is the set of nodes representing the optimizations and E is the set of edges, where we add an edge between two optimization nodes if that pair of optimizations appears in the optimization sequence in `-O3`. Further, we can annotate an edge with the number of times that each pair of optimizations consecutively appears in the sequence. The outcome of this is a graph, we term the optimization dependence graph. This graph can be represented by a *Weighted Adjacency Matrix* that has the size of N×N. This matrix can be used for clustering optimizations into different sub-sequences. Figure 6.2 shows the constructed graph on `-O3`.

Graph and Sub-sequence Clustering

For our clustering of optimizations into sub-sequences, we could have used any number of the numerous clustering methods proposed in the literature related to Pattern Recognition (e.g. iterative, hierarchical, divisive, etc,) [201]. We selected *agglomerative clustering* [244] which is an iterative clustering technique that merges smaller clusters and improves the complexity of k-mean clustering on graphs [201]. A key insight of this method is that it treats clusters as a dynamical system and its samples as states. The algorithm works as follows: Agglomerative clustering, receives as input the matrix of the graph G and the number of desired clusters (n_T) and builds (i) the graph G with k-nearest-neighbors upon computing its Weighted Adjacency Matrix (W). (ii) The algorithm then calculates the transition probabilities and (iii) forms sample clusters $C = \{c_1, ..., c_{nc}\}$. (iv) It enters a loop to iteratively try to add more sub-clusters to the already available clusters in C as long as the conditional sum of the all-path integrals within the new sub-clusters maximizes some objective function (*argmax*) [244]. A path integral is a metric to measure the stability of a dynamical system and is computed by summing the paths within the cluster on the directed graph weighted by transition probabilities. We used the algorithm and tentatively increased the number of max de-

sired clusters until no clusters could be added. The final five clusters, namely, the best optimization sub-sequences the algorithm could find are reported in Section 6.5 Table 6.2.

Benefits of Sub-sequences

Clustering optimizations into sub-sequences makes sense. Certain analysis algorithms typically should be done before an optimization in order for the optimization to have any significant impact. For example, we may want to run analysis that performs basic block counts and predicts branch instruction outcomes before applying an optimization that reorders the code blocks in an application. Additionally, it is likely that `-O3` will contain optimizations that should follow other optimizations in order to obtain the best performance. Thus, forming a cluster of optimizations that should be applied together makes a lot of sense.

6.4.3 The Proposed Mapper

Constructing prediction models for the problem of selecting the right compiler optimizations with fixed-length feature vectors has been extensively studied [41, 177–179]. However, prediction models fall short when correlating program characterizations with the right compiler optimizations to apply when it comes to a variable optimization sequence length [8]. Therefore, we adapt a simple encoding technique which allows us to map the representation of the compiler phase-ordering sub-sequences to a fixed-length vector of optimizations and at the same time preserves the order of optimizations in the sequence. Our proposed mapping function encodes an optimization sequence into a bit string.

Let $\mathcal{A} = \{\alpha_1, \ldots, \alpha_N\}$ be the set of all variables, which can be thought of as an *alphabet*. Every α_i is a *letter*. A finite string of not necessarily distinct letters is called a *word*. Thus, each word is a concatenation of the form $\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}$, where $i_1, i_2, \ldots, i_k \in \{1, \ldots, M\}$. The integer k is the *length* of the word. We will also allow the empty word which by definition has length zero.

There is a simple way of encoding the space $\mathcal{W}$ of all words of length at most M using the space described by $\{0,1\}^{N\times M}$ consisting of all binary strings of the fixed length $N \times M$. To see this, consider the mapping function $f : \mathcal{A} \to \{0,1\}^N$ which maps each letter α_i to the binary string $f(\alpha_i) = b_1 \cdots b_M$, where

$$b_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \neq i. \end{cases}$$

Now we define the mapping function $F : \mathcal{W} \to \{0,1\}^{N\times M}$ by mapping each word $\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}$ to the binary string

$$F(\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}) = f(\alpha_{i_1})f(\alpha_{i_2})\cdots f(\alpha_{i_k})\ \underbrace{\mathbf{0}\cdots\mathbf{0}}_{N-k \text{ times}},$$

where $\mathbf{0} = 0\cdots0$ is the zero string of length N. Evidently the map F is one-to-one. The image $F(\mathcal{W})$ is much smaller than the target space $\{0,1\}^{N\times M}$, as these sets have $\sum_{k=0}^{M} N^k$ and $2^{N\times M}$ elements, respectively.

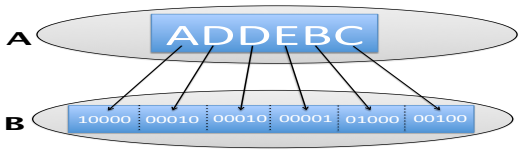


Figure 6.3: *An example of the proposed mapping function on the example where we have repetitions and* $\{N = 5, M = 6\}$*: Each letter represents a compiler sub-sequences containing different compiler optimizations.*

If we identify each element of $\{0,1\}^{N\times M}$ with a concatenation $s_1 \cdots s_N$ of N elements of $\{0,1\}^N$, the image $F(\mathcal{W})$ can be simply characterized by the following two requirements:

1. Each s_i has at most one non-zero binary digit.
2. If $s_i = \mathbf{0}$ and $s_j \neq \mathbf{0}$, then $i > j$.

Given the proposed mapping, there exists a one-to-one (1:1) mapping F for every instance of $\mathcal{A} = \{\alpha_1, \ldots, \alpha_N\}$ with the binary size of $N \times M$ that has the same characteristics of the original presentation with the benefit of having a fixed $N \times M$ length. An example of the proposed mapping function is shown on the Figure 6.3. Our adapted mapping function uses a *one-hot encoding* approach [169] for N=5 and M=6 to assign a single high (1) at each segment of the transformed binary while other bits are turned off (0). This technique can inexpensively preserve the order and the repetitions of optimizations in a sequence, at the same time it assures the transformed feature vector has fixed-length size.

6.4.4 Predictive Modeling

The proposed methodology in Figure 6.1 illustrates the use of predictive modeling in both the offline (training) and online (testing) phases of the of the process. We used the predictive modeling in the offline training phase to (i) construct the model and in (ii) the online prediction phase we exploit the constructed model on the target application to predict the speedup of a complete optimization sequence without the need to actually apply the sequence of optimizations to the code.

Constructing the Prediction Model Predictive modeling is the process of constructing, testing and validating a model to predict an unobserved outcome based on characterization of a state from which to predict the outcome. In this chapter, the state being characterized is the code being optimized and the predicted outcome corresponds to the speedup metric calculated by normalizing the execution time of the current optimization sequence by the execution time of the baseline optimization sequence. The general formulation of the optimization problem is to construct a function that takes as input the features of the unoptimized program being compiled. In other words, this model takes as an input a tuple (F, T) where F is the feature vector of the collected instrumentation of the program being optimized; and T is one of the several possible compiler sequences predicted to perform well on this program. Its output is a prediction of the speedup T should achieve when applied to the original code.

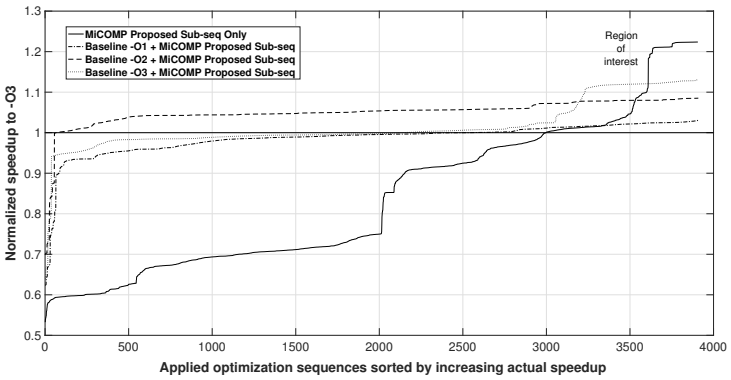


Figure 6.4: *Empirical analysis of having different compilation baseline across all CBench applications (Harmonic mean). Region of interest is depicted where MiCOMP sub-sequences outperformed other compiler sequences having a fixed standard compilation baseline.*

Analysis of Selecting the Compilation Baseline

As explained in Section 6.4.2, we do not use any of the default compilation optimization levels as a baseline to start from since we used all compiler optimizations passes that are used in `-O3` for our clustering purposes (see Section 6.4.2). Additionally, we found that using a baseline compiler optimization level to start from ultimately reduces the speedup achievable from the sequence we construct with predictive modeling. We empirically justify this argument by running a set of experiments, one without using a compiler optimization level as a baseline and a set of experiments where we use a sequence of optimizations to apply on top of different `-OX` baselines. Figure 6.4 illustrates the mean of MiCOMP's proposed optimization sequences using different compilation baselines. All four speedup lines have an upper-bound of sequences of length five. Results suggests that using MiCOMP optimization sequence without an optimization level as a baseline can lead to substantial benefits compared with using any of `-OX` optimization levels as a baseline. MiCOMP targets the region of interest where the highest achievable speedup values are located and it drives the prediction model to reach that part of the optimization sequence search space least number of predictions. Note that using a baseline of `-O1`, `-O2`, or `-O3` all converge to a sub-optimal speedup. Thus, applying certain sequences causes a degradation in performance as can be seen by using these standard optimization levels as a baseline. The better option is to not use a baseline sequence at all and to allow MICOMP to predict the best sequence to apply on its own.

The insights of this experiment are threefold: (**i**) The clustering technique is beneficial; first, to gain better speedup values and second, to reduce the number of compiler optimization sequences needed to achieve the best results from around 50 to 5 so that our iterative compilation method is both scalable and practical. (**ii**) The sub-sequences can be coupled with machine-learning techniques so they can be reordered based on the applications being optimized while outperforming the highest standard optimizations levels. (**iii**) Phase-ordering indeed does matter in the field of compilers; i.e., using the same set of optimization flags available to `-O3`, MiCOMP can significantly outperform `-O3` itself.

Application-specific Prediction

Our machine-learning constructed models can be used for unseen target applications to predict the speedup when applying compiler sequences to them. The predicted speedup values correspond to the optimization sequence applied to the program. For a given input program, first a feature vector containing dynamic instrumentation is collected. Then, our prediction model is fed the features of the program being compiled to predict the expected speedup if a optimization sequence T was applied to it. By predicting the performance of each possible optimization sequence that can be applied, it is possible to rank the optimization sequences according to their expected speedup and only select the sequences to actually apply that are predicted to give the highest speedups.

A state-of-the-art ranking approach [177, 179] was used to rank optimization sequences in descending order, and we only select the top N optimization sequences to evaluate their actual optimization quality. In this work, we propose an iterative process in which different solutions are explored to find those leading to higher speedups. In other words, our proposed exploration technique, uses the output of our prediction model to generate an initial exploration strategy, and the exploration strategy dynamically updates itself in order to reach the highest speedup values in the least number of predictions.

6.4.5 Recommender System Heuristic

Mitigating the phase-ordering problem imposes a proper exploration strategy. In the initial steps taken by [19, 130], the authors defined iterative exploration heuristics, based on the current optimized state of the target application being compiled, to select the next best optimization to apply, which will bring the eventual best speedup. As the current state of the optimized application depends on the optimizations that were already applied, this previous approach required several rounds of feature collection. In this chapter, we propose a predictive approach that generates the complete optimization sequence for a program that has not been optimized, thus it needs to collect features only once before any optimizations are applied.

Adjusted Cosine Similarity

Many of the aforementioned state-of-the-art approaches, tackling both the selection and the phase-ordering problem, define exploration strategies on the optimizations design space. Yet, to the best of the authors' knowledge, none of them make use of information in order to dynamically improve the strategy itself. Dynamic information, in our particular case, is the predicted speedup on the sequences already explored and evaluated. The knowledge can be effectively used to improve the initial exploration. The technique we propose, leverages the similarity between the unexplored and the explored optimization sequences. In particular, our proposed technique prioritizes the evaluation of solutions less similar to the ones already explored. is especially important for the phase-ordering problem where there are a plethora of optimization sequences that need to be explored. The similarity measure is based on how close the achieved speedup is for predicted solutions across all the training sets. As an example, let $S_{p,i}$ and $S_{p,j}$ be the predicted speedups of the sequence i and j when applied to program p in the set of programs P. We define an iterative process to look for predicted similarities in i and j.

In recommender system (RS), an algorithm called Basic Cosine Similarity [200] is used to correlate users and items. However, computing the similarity using this algorithm has one important drawback; the difference in rating scale are not taken into account. The Adjusted Cosine Similarity offsets this drawback by subtracting the corresponding user-average from each co-rated pair. Adapting this technique, we can compute the Adjusted Cosine Similarity between optimization sequence i and j as:

$$sim(i,j) = \frac{\sum_{p \in P}(S_{p,i} - \bar{S_p})(S_{p,j} - \bar{S_p})}{\sqrt{\sum_{p \in P}(S_{p,i} - \bar{S_p})^2}\sqrt{\sum_{p \in P}(S_{p,j} - \bar{S_p})^2}} \tag{6.2}$$

where $S_{p,i}$ is the speedup achieved by sequence i when applied to program p of all set of programs P, and $\bar{S_p}$ is the average speedup on program p. We use the computed measure to evaluate the correlation between a pair of optimization sequences to boost our exploration strategy.

Algorithm 1 Proposed heuristic using Adjusted Cosine Similarity

```
tmpTestedSet = EmptySet
while phases Not Tested Yet do
  tmpTestedSet = EmptySet
  for phases in prediction space do
    if new phases exist then
      if similar solution exists then
        Skip Phases
      else {Test phases: sim(i, j)}
        Add phases
      end if
    end if
  end for
end while
```

The pseudocode of the algorithm is shown in Algorithm 1. The exploration strategy is defined as follows:

1. Sort predicted speedup solutions in decreasing order in a list.
2. Test solutions in order. If the solution to test is too similar to one already tested in the current list iteration, skip it.
3. If the end of the list has been reached and there are still optimization sequences to test, go to 2., starting from the head of the list and excluding already tested solutions.

High values of Adjusted Cosine Similarity (ACS) for a pair of optimization sequences are the consequence of achieving pairwise similar speedups across all training data. We exploit this measure to give exploration priority to the solutions that are less similar to the ones already tested. This allows the our ACS algorithm to boost exploration to cover different areas of the optimization sequence space quicker than it would have achieved by predictive modeling alone, thus achieving better speedups with less exploration steps.

6.5 Experimental Results

In this section we evaluate our proposed methodology on an Intel Xeon architecture. We adapted our instrumentation and architecture-independent tool (Section 6.4.1) to extract characteristics from a large set of benchmarks from the Ctuning CBench suite [86].

We have used LLVM compilation framework v3.8 (`Clang` for the frontend/backend and `Opt` for the optimization passes). The training set consists of different applications ranging from automotive, security, office, and telecom. The list of applications we evaluated is reported in Table 6.1. Table 6.2 illustrates the list of different compiler optimizations that are clustered into 5 different *sub-sequences* (refer to Section 6.4.2) that are derived from LLVM's `-O3`. We used the sub-sequences with no baseline in MiCOMP and generate the design space enabling orderings and repetitions of these sub-sequences. The optimizations are fixed within a sub-sequence, but sub-sequences are allowed to appear in any order in the full optimization sequence.

An application's execution time is measured by using the Linux *Perf* tool. Execution time of a compiled application binary is measured by averaging the execution times of three different executions. In order to implement a dimension reduction technique we applied PCA. This analysis reveals that by using a 5-D vector of features we can capture 98% of the variance available in the training set. The Principal Components (PCs) have been computed using the MICA features collected from application executions (it is required only once), normalized by standard deviation across all data sets. An application characterization phase takes between 15 to 50 seconds depending on the type of the application. We noticed a small factor of slowdown when we perform the feature collection phase versus measuring the pure application's execution time. The overhead is negligible first, as it is required once and second, the speedup gained by using MiCOMP is far higher. The proposed methodology is prediction model independent, and we report the results using three different models described in Table 6.3. Machine learning algorithms we used include **(i)** a Linear Regression (LR) classifier using the M5 attribute selection method with default ridge parameter, **(ii)** a *Multilayer Perception* that back-propagates to classify instances and using the default configuration, and the **(iii)** K* algorithm using default settings.

In this work, the WEKA machine learning tool [94] has been integrated in our frameworkWe trained different speedup predictors, each one by excluding from the training application set, one of the applications. This technique is called Leave-One-Out-Cross Validation (LOOCV) and ensures a fair evaluation of our trained models. Validation data is used on the application excluded from the training set for prediction purpose. In MiCOMP, cross validation is done in a few minutes for each application under analysis [4].

6.5.1 Analysis of Longer Sequence Length

As described in Section 6.4, MiCOMP requires having upper bound on the sequence length for using the encoding scheme. To this end, we evaluate MiCOMP by having different max values for the sequence length. A speedup prediction model requires a one time expensive training be done in order to construct an accurate model. We believe that the longer the sequence length, the better the chance of finding higher speedup values. To that end, we have tested our proposed sub-sequences with different maximum sequence lengths to empirically find the most effective length across all the training applications. This is done also with the goal of scalability and speeding up the training phase.

[4] Model construction is heavily correlated with the type of machine learning algorithm we use. We observed LR to be the fastest and MLP to be slowest for our data.

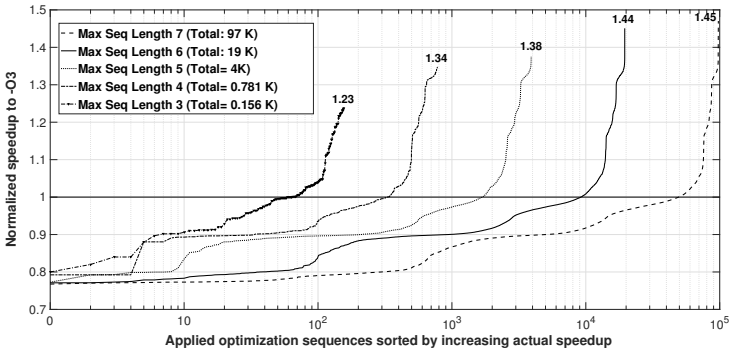


Figure 6.5: *Empirical analysis of having different compiler sequence lengths on 5 candidate applications:* telecom_adpcm_d, jpeg_d, bzipd, network_dijkstra, automotive_bitcount. *Note that X axis is in logarithmic scale.*

Figure 6.5 gives the Harmonic mean (as suggested by [99] [5].) values of the actual speedups using five selected applications each having different upper bound sequence lengths. We randomly selected an application from each of CBench categories (automotive, compression, telecom, consumer and network) since it was impractical to do this analysis with all applications. Having the upper bounds set to 3, 4, 5, 6 and 7 respectively, gives search spaces of 156, 781, 3909, 19k and 97k distinct permutations of sub-sequences with repetitions enabled (refer to Equation 6.1 for the optimization space). The five speedup lines show the trend of reaching a higher speedup value by iteratively exploring more fraction of optimization space. The maximum speedup found against `-O3` using sequence lengths of 3, 4, 5, 6, and 7, respectively, are 1.23, 1.34, 1.38, 1.44 and 1.45. These results suggest to set the maximum length to 6 as this ensures achieving good speedups while avoiding a potential exploration of 100K sequences per each application in the training set. For our optimization sub-sequences, this empirically found upper bound value is the right trade-off between choosing a good optimization sequence space and the possibility to explore efficiently.

6.5.2 MiCOMP Prediction Accuracy

Unlike sequence prediction models [7, 20, 179] in speedup prediction approaches, prediction quality is measured by means of prediction error. This metric demonstrates how close the prediction values were to the actual speedups given the same sequence. We use the following different error measurement techniques.

Mean Absolute Error In statistics, the Mean Absolute Error (MAE) [105] is a quantity used to measure how close predictions are to the eventual outcomes. The mean absolute error is given by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|f_i - y_i| \tag{6.3}$$

where we define e_i as $|f_i - y_i|$ given f_i as the prediction values and y_i the actual

[5] We provide harmonic mean rather than arithmetic mean as we are dealing with averaging speedups. Note that harmonic-mean is always less than or equal to arithmetic-mean

Table 6.1: *Applications under analysis (CTuning CBench suite [86])*

cBench list	Description
automotive_bitcount	Bit counter
automotive_qsort1	Quick sort
automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corners
automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Edges
automotive_susan_s	Smallest Univalue Segment Assimilating Nucleus Smoothing
security_blowfish_d	Symmetric-key block cipher Decoder
security_blowfish_e	Symmetric-key block cipher Encoder
security_rijndael_d	AES algorithm Rijndael Decoder
security_rijndael_e	AES algorithm Rijndael Encoder
security_sha	NIST Secure Hash Algorithm
telecom_adpcm_c	Intel/dvi adpcm coder/decoder Coder
telecom_adpcm_d	Intel/dvi adpcm coder/decoder Decoder
telecom_gsm	gsm encoder/decoder
consumer_jpeg_c	JPEG kernel
consumer_jpeg_d	JPEG kernel
consumer_tiff2bw	convert a color TIFF image to grey scale
consumer_tiff2rgba	convert a TIFF image to RGBA color space
consumer_tiffdither	convert a TIFF image to dither noisespace
consumer_tiffmedian	convert a color TIFF image to create a TIFF palette file
network_dijkstra	Dijkstra's algorithm
network_patricia	Patricia Trie data structure
office_stringsearch1	Boyer-Moore-Horspool pattern match
bzip2d	Burrows Wheeler compression algorithm
bzip2e	Burrows Wheeler compression algorithm

Table 6.2: *Candidate clusters of compiler optimizations into sub-sequences (all derived from `LLVM -O3`)*

sub-seq	Compiler Passes
A	-alignment-from-assumptions -argpromotion -barrier -bdce -block-freq -branch-prob -constmerge -deadargelim -demanded-bits -dse float2int -forceattrs -functionattrs -globaldce -globalopt -globals-aa -gvn -indvars -inferattrs -inline -ipsccp -jump-threading -lcssa -loop-accesses -loop-deletion -loop-idiom -loop-unroll -loop-unswitch -loop-vectorize -mldst-motion -prune-eh -reassociate -rpo-functionattrs -sccp -simplyfycfg -sroa -strip-dead-prototypes
B	-licm -mem2reg
C	-instcombine -loop-rotate -loop-simplyfy
D	-memcpyopt
E	-adce -loop-unswitch -slp-vectorize -tailcallelim

values. Consequently, the value e_i is inverse proportional to the accuracy of the prediction.

Approximation Error Complementary to MAE, Approximation Error (AE) [208] is a common error measurement whereas in some data there is some discrepancy between an exact value and the approximation. An approximation error can occur because (i) certain measurements of the data are not precise (which we consider it can be the case for any computer scientific measurement) and (ii) approximated values are used instead of the real values (the iterative prediction way keeps using the predicted values). It is calculated as:

Table 6.3: *List of the predictive models used in our experiments. Note that the proposed methodology is independent from any specific machine-learning algorithm (classifier) and it can be paired with any algorithm desired.*

Predictive Model	Description
MultilayerPerceptron (MLP)	A feedforward artificial neural network model that maps sets of input data onto a set of appropriate outputs. A MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one
LinearRegression (LR)	An approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X. In linear regression, the relationships are modeled using linear predictor functions whose unknown model parameters are estimated from the data.
KStar	It is an instance-based classifier, that is the class of a test instance is based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy-based distance function.

Table 6.4: *Average error rate for the proposed mapping function versus an arbitrary mapping*

M.L	MiCOMP Mapping function		Arbitrary Mapping		Improvement Factor	
	MAE	AE	MAE	AE	MAE	AE
MLP	0.06798	0.05479	0.10826	0.11838	1.59×	2.16×
LR	0.07525	0.07795	0.12879	0.13974	1.71×	1.79×
KStar	0.05179	0.05078	0.09188	0.10866	1.77×	2.13×

$$\delta = \frac{|\epsilon|}{|v|} = \frac{|v - v_{approx}|}{|v|} \tag{6.4}$$

where the absolute error is the magnitude of the difference between the exact value and the approximation. These definitions can be extended to the case when v and $v_{approximate}$ are n-dimensional vectors, then by replacing the absolute error with an n-norm error.

Prediction Accuracy

We provide the prediction's error rate in Table 6.4. We observe that the arbitrary mapping leads to higher error rates in the prediction values. Exploiting the adapted encoding scheme reduces the noise on the prediction and stabilizes the output trend with lower error-rate. Table 6.4 shows that the KStar model does slightly better in terms of accuracy compared with other models, it achieves around 5% error rate on average. In general, having a smaller error rate does not always guarantee higher performance gain but rather showcases the accuracy of the prediction model to capture the correlation between different compiler sub-sequences and the speedup values. [6]

Iterative Compilation Max Speedups

Iterative compilation is known to be able to achieve good performance results when compiling applications [28]. However, the approach is expensive and should be combined with more intelligent search algorithms [7, 20]. Table 6.5 reports the maximum speedups found by an iterative compilation approach using our proposed clustering while exploring the full optimization space. This experiment empirically confirms that the proposed clustering is useful on the phase-ordering space since we show that we can achieve on average a 26% speedup versus `-O3`. Figure 6.4 illustrates the trend

[6] We are aware of the many other encoding possibilities that are more efficient (currently having $N \times M$ length). However, we believe that extending the current encoding scheme to a more sophisticated version is out of scope of the work. Moreover, the proposed clustering technique can effectively reduce the number of N, thus the encoding scheme is scalable for higher orders.

Table 6.5: *Best compiler optimization sub-sequences found using an iterative compilation and their related speedups*

Application	Best sub-sequence	Speedup w.r.t. -O3
telecom_adpcm_c.csv	ECDDCC	1.35
security_sha.csv	ACCACE	1.06
security_blowfish_e.csv	BCCEEA	1.03
automotive_susan_e.csv	AABACA	1.15
consumer_tiffdither.csv	DCEDCD	1.20
security_rijndael_e.csv	CAEEC	1.10
consumer_tiff2bw.csv	CCDCD	1.30
bzip2e.csv	CBADCA	1.30
automotive_susan_s.csv	ECCCDE	1.22
office_stringsearch1.csv	ABCBAC	1.07
telecom_adpcm_d.csv	DCAACA	1.13
consumer_jpeg_c.csv	DDC	1.14
network_patricia.csv	CECBAA	1.08
automotive_susan_c.csv	BDBCCB	1.23
consumer_tiff2rgba.csv	DEDDC	1.32
automotive_qsort1.csv	CBAAAC	1.04
security_blowfish_d.csv	DACECA	1.05
network_dijkstra.csv	EECBBE	1.51
security_rijndael_d.csv	ECEACD	1.05
bzip2d.csv	CBDACA	1.29
automotive_bitcount.csv	BEACCA	1.07
consumer_jpeg_d.csv	CCED	1.18
consumer_tiffmedian.csv	BCBACB	1.15
telecom_gsm.csv	BACBAC	1.07
Harmonic mean		**1.26**

when using MiCOMP sub-sequences with no baseline compared with having a baseline (e.g.: `-O1`, `-O2` or `-O3`). The best optimization sequence for each of applications under-analysis and its speedup value are reported in the second and the third columns of Table 6.5. Readers can refer to Table 6.2 to find the exact set of compiler optimizations clustered in each sub-sequence.

6.5.3 Performance Gain of the MiCOMP Technique against the Ranking Approach

Our proposed approach can improve the exploration to find the best optimization sequences in an optimization search space and to find the best speedups using a fewer number of predictions. Table 6.6 reports the comparison between the best speedup found by our approach and a state-of-the-art N-shot approach [177, 179]. The results, averaged using a Harmonic mean across all applications, show that using the same number of predictions from both models, our exploration technique can outperform the ranking approach on every number of predicted optimization sequences used (1, 5, 10, 15 and 20). This shows that our proposed methodology can effectively predicts the best compiler sequences to use and converges faster to better solutions in the space.

Table 6.6: *Prediction improvement of MiCOMP based on Adjusted Cosine Similarities against the Ranking (N-shot approach)*

Exploration Techniques	Top-1	Top-5	Top-10	Top-15	Top-20
MiCOMP	1.01	1.06	1.09	1.10	1.12
Ranking	0.93	1.02	1.06	1.07	1.08

6.6 Comparative Results

In this section we evaluate the results of our model against three different techniques: **(i)** Standard optimization levels, **(ii)** Random Iterative Compilation (RIC) and, **(iii)** Non-iterative Models. We use our MiCOMP exploration policy and compare the performance of predictions to a previously published ranking approach. For each application under analysis, we tested the speedup gained using 1, 5 and 10 predictions and provide the Harmonic mean values. Table 6.7, reports the results. For each application and number of predictions, we provide two: i) achieved speedup compared with `-O3` and, ii) achieved speedup compared with the optimal solution for the specific application. For example, one can see that for the network_dijkstra application we can gain a higher speedup values using MiCOMP and, on average even better than `-O3` from just the first prediction. Moreover, we can achieve a 4% performance improvement over `-O3` when we use 5 predicted optimization sequences from our model. Over all our benchmarks, using our model we can achieve 1%, 4%, and 9% speedups over -O3 using 1, 5, and 10 predicted optimization sequences, respectively. Consequently, our technique allows MiCOMP to outperform `-O3` by high margins. Thus combining prediction models with iterative compilation achieves much better performance than using pure iterative compilation alone. For example, note in Figure 6.5 that it takes a pure iterative compilation technique 100-5000 iterations to surpass the performance of `-O3`.

6.6.1 Comparison with Standard Optimization Levels

Standard optimization levels have been introduced to achieve good performance on average. However, they are coming short of the customized auto-tuning framework per architecture/application/dataset. As we showed in Table 6.7, MiCOMP can surpass the performance of `-O3` with a few predictions on application bases. Here we provide Table 6.8 which reports more fine-grained speedup over all standard optimization levels. This demonstrates how fast (first number in the tuple) and in what percentage of the explored space (the second number), the framework is reaching a sequence which can outperform the specific standard optimization level. Each column is reporting two values: (i) in how many predictions and (ii) in what percentage of the whole configuration space the propped methodology can outperform `OX` levels.

6.6.2 Comparison with Non-iterative Model

In this section, we compare MiCOMP with two state-of-the-art intermediate-sequence prediction approaches proposed in [19, 130].

Table 6.7: *MLP's speedup table against LLVM's -O3. Reported numbers are A (B%):* ***(A)*** *speedup and* ***(B)*** *achieved speedup w.r.t. the optimal speedup value*

Application	1 prediction	5 predictions	10 predictions
automotive_bitcount	1.04 (95.38%)	1.07 (98.12%)	1.08 (98.92%)
automotive_qsort1	1.01 (95.32%)	1.03 (96.93%)	1.03 (97.55%)
automotive_susan_c	1.04 (96.61%)	1.06 (98.53%)	1.06 (99.07%)
automotive_susan_e	1.04 (96.47%)	1.03 (98.41%)	1.04 (99.00%)
automotive_susan_s	0.99 (96.26%)	1.01 (98.42%)	1.02 (98.98%)
bzip2d	0.93 (92.77%)	0.96 (94.02%)	1.00 (94.37%)
bzip2e	1.09 (83.77%)	1.10 (86.02%)	1.12 (90.37%)
consumer_jpeg_c	1.01 (85.18%)	1.07 (90.35%)	1.10 (94.51%)
consumer_jpeg_d	1.09 (84.70%)	1.14 (88.97%)	1.17 (97.85%)
consumer_tiff2bw	0.96 (75.54%)	0.99 (80.59%)	1.02 (82.46%)
consumer_tiff2rgba	0.91 (80.61%)	0.95 (86.19%)	1.07 (88.08%)
consumer_tiffdither	1.02 (80.14%)	1.09 (85.86%)	1.11 (87.68%)
consumer_tiffmedian	0.94 (79.21%)	1.02 (85.72%)	1.06 (89.31%)
network_dijkstra	1.13 (60.00%)	1.29 (68.46%)	1.38 (73.00%)
network_patricia	0.91 (64.99%)	0.93 (70.79%)	0.97 (73.91%)
security_sha	0.91 (64.99%)	1.01 (70.79%)	1.03 (73.91%)
security_blowfish_e	0.92 (64.99%)	1.03 (70.79%)	1.03 (73.91%)
security_blowfish_d	0.91 (64.99%)	0.99 (70.79%)	1.02 (73.91%)
security_rijndael_e	0.92 (64.99%)	1.02 (70.79%)	1.01 (73.91%)
security_rijndael_d	0.89 (64.99%)	1.01 (70.79%)	1.04 (73.91%)
telecom_adpcm_c	0.91 (64.99%)	1.01 (70.79%)	1.02 (73.91%)
telecom_adpcm_d	0.92 (64.99%)	1.02 (70.79%)	1.01 (73.91%)
telecom_gsm_d	0.89 (64.99%)	1.03 (70.79%)	1.04 (73.91%)
Harmonic mean	**1.01 (82.74%)**	**1.04 (87.51%)**	**1.09 (91.52%)**

Table 6.8: *Average Speedup w.r.t LLVM -O3. Numbers are A (B%):* ***(A)*** *How fast (in terms of number of predictions) in average the proposed methodology outperforms LLVM standard Optimizations.* ***(B)*** *The percentage of the optimization space explored to satisfy the goal.*

Predictive Modeling	-O1	-O2	-O3
MultilayerPerceptron	1 (0.01%)	1 (0.01%)	3 (0.02%)
LinearRegression	1 (0.01%)	1 (0.01%)	3 (0.02%)
KStar	1 (0.01%)	1 (0.01%)	2 (0.016%)

Intermediate Speedup Comparison Case. (A)

Kulkarni et al., [130] used *Neuro-Evolution for Augmenting Topologies* (NEAT) to predict the best compiler optimization to apply given the state of source-code being optimized by the dynamic JIT Jikes RVM compiler. They used static source-code features to characterize each state of application under optimization, as opposed to the technique we propose in this chapter where we obtain features of the code only once before it is optimized. Kulkarni et al., used NEAT, a machine-learning framework based on genetic evolution, to generate many neural-networks where each network was evaluated on the task of using static source code features to predict the next compiler optimization to apply. NEAT can make optimization predictions to any given maximum-length to predict the most beneficial sequence of optimizations for the target application being compiled. In NEAT training time was reported around 10 days while the current approach requires a few hours to construct the model. Another advantage of the current work is the fact that is supporting multiple predictions from the prediction-space while

Table 6.9: *Performance comparison of the single prediction by MiCOMP against the intermediate speedup approach reported in in previous work [130]. All values are normalized by* `-O3`.

Application	NEAT		MiCOMP
	Best NN Size	1 prediction	1 prediction
automotive_qsort1	1105	1.0336	1.0385
automotive_bitcount	1536	1.0923	1.0498
automotive_susan_c	607	1	1.0491
automotive_susan_e	613	1	1.0481
automotive_susan_s	1295	1.0135	1.0195
bzip2d	1159	1	0.9698
consumer_jpeg_c	1327	1.0205	1.1882
consumer_jpeg_e	596	1	1.0981
consumer_tiff2bw	1038	0.9522	0.9491
consumer_tiff2rgba	1147	0.9905	0.9295
consumer_tiffdither	612	1	1.0288
consumer_tiffmedian	1356	0.9097	0.9497
network_dijkstra	1343	1.0353	1.1382
network_patricia	622	0.7971	0.8585
Harmonic Mean		**0.9742**	**1.0275**

the NEAT approach can produce one-shot result based on the stop condition for each application and neural network configuration. We reproduced the work by Kulkarni et al. [130] by using 100 chromosomes and 500 generations on 12 cores Xeon(R) CPU E5-1650 v2 @ 3.50GHz with 12GB running on Ubuntu and we report the result in Table 6.9. We ran NEAT in parallel with average running time of 1.75 hours per model (the longest took 4 hours).For this comparison, we used the same training data of up to the length of 4 for both MiCOMP and Kulkarni et al. to be uniform on both comparisons as NEAT needs feature collection on each iteration and collecting 19k feature vectors for the number of applications in our training set was impractical. The training/prediction is done with leave-one-out cross-validation to be uniform to the reported experimental results in this chapter. We used Harmonic-mean to average the speedup gains on both models and observed 1.0295$\times$ speedup (5$\%$ performance improvement) against the mentioned work.

Intermediate Speedup Comparison Case. (B)

Ashouri et al. [19] demonstrated a predictive methodology in order to predict the intermediate speedup obtained by an optimization from the configuration space, given the current state of the application. The fitness function for the intermediate speedup is the ratio between the execution times of the program before and after the optimization process. They exploited predictive models to correlate the current state of the dynamic features of the application under study with the current state of the compiler optimization to come up with a speedup value and utilize heuristics to search that space. Ashouri et al. used dynamic feature selections, however as mentioned in Section 6.6.2, a major downside in this work is that application feature should be collected on every state by means dynamic feature extraction and this makes the system impractical on large-scale data. There is an extension to the aforementioned work. Their comparison baseline was LLVM's default optimization, while here we provide a comparison against LLVM's `-O3` (we show MiCOMP can outperform an aggressive optimization

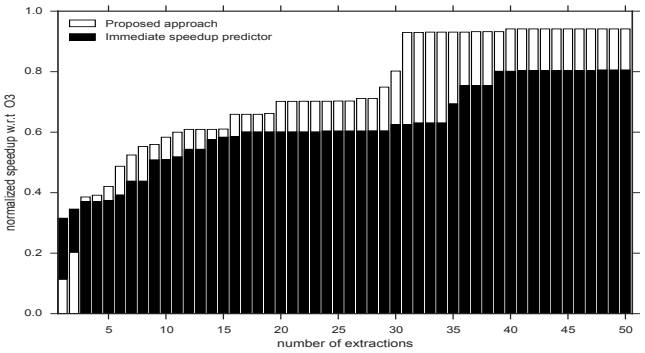


Figure 6.6: *Performance of MiCOMP w.r.t the performance of intermediate speedup predictor approach [19]*

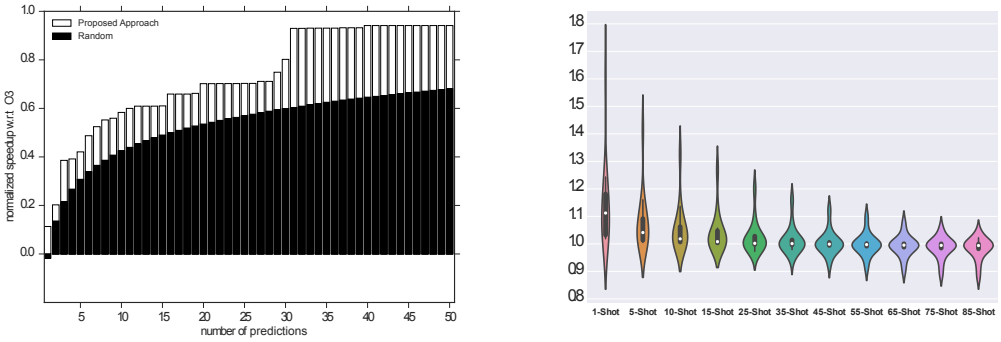
setting in LLVM, that is, `-O3`, in only a few predictions.). Figure 6.6 demonstrates the comparison. For this comparison, we used the same training data of up to the length of 4 for both MiCOMP and Ashouri et al. to be uniform on both comparisons. We observe that except the first two predictions, the proposed approach outperforms the intermediate speedup methodology reported in this work and on average MiCOMP brings 11% speedup gain.

6.6.3 Comparison with Random Iterative Optimization

As we illustrated in Section 6.5.2, iterative compilation can improve application performance over standard compiler optimization sequences [7, 28]. Additionally, several published works have shown that drawing compiler optimization sequences at random can often be as good as using other more complicated search algorithms, such as genetic algorithms or simulated annealing [7, 41, 46]. In this section, we compare the effectiveness of MiCOMP to a Random Iterative Compilation (RIC) method that samples sequences from a uniform distribution. We randomized the distribution of predictions 10000 times to make sure the obtained model is totally uniform. Our results are presented in Figure 6.7. To present our results, we define *Normalized Performance Improvement* (NPI) as the ratio of the performance improvement achieved over the potential performance improvement:

$$NPI = \frac{E_{ref} - E}{E_{ref} - E_{best}} \tag{6.5}$$

where E is the execution time achieved by the methodology under consideration, E_{ref} is the execution time achieved with a reference compilation methodology and E_{best} is the best execution time that can be obtained through an exhaustive exploration of all possible compiler optimization sequences in the optimization space we are exploring. As the execution time E of the iterative compilation methodology under analysis gets closer to the reference execution time E_{ref}, the value of NPI gets closer to 0, where 0 indicates no improvement was obtained. As E approaches the best execution time, E_{best}, the value of NPI approaches 1. An NPI value of 1 indicates that the optimal performance available was achieved. The goal of the evaluation in this section is to show how effective MiCOMP is at exploring the optimization sequence space compared to RIC. Figure 6.7a shows results for both MiCOMP and RIC with the same.



(a) *Both MiCOMP and Random exploring the opt. space*

(b) *Only Random iterative compilation exploring the opt. space (MiCOMP is fixed at 5 predictions)*

Figure 6.7: *MiCOMP performance comparison versus Random Iteration Compilation*

The X axis pertains to the number of predicted optimization sequences used and the Y axis shows their corresponding speedup values. We used NPI (scaled within $[-\infty, 1]$) and the speedups are all normalized by `-O3` performance. Thus, $Y = 0$ is the speedup line corresponding to `-O3`. We observe that the performance of MiCOMP outperforms Random Iterative Compilation for each number of predicted optimization sequences used. Note the larger the number of predicted sequences used, the more significant the performance difference between MiCOMP and RIC. Table 6.5 gives the the absolute speedup values.

Figure 6.7b displays another result where we compare a fixed number of predicted optimization sequences for MiCOMP, that is 5 predicted sequences, versus different number of predicted sequences from RIC. That is we observe the prediction quality of MiCOMP compared to different numbers of predicted optimization sequences drawn from a random distribution. Figure 6.7b depicts this scenario using a violin plot where the Y axis pertains to the speedup with respect to the RIC and the X axis corresponds to the different predicted optimization sequences obtained from RIC. Statistically, we observe that the quality of the 5-prediction of MiCOMP is as good as using 25 prediction optimization sequences from RIC. In our experiments, we observed that the predicted optimization sequences derived by MiCOMP can give up to $5\times$ exploration speedup versus the RIC method.

6.7 Conclusion

This work presents MiCOMP, a framework to exploit predictive modeling to solve the compiler phase-ordering problem. We proposed a clustering technique for all the compiler optimizations in LLVM's `-O3` and clustered them in five different optimization sub-sequences to speedup the training and exploration phase. This method helps us outperform LLVM's `-O3` optimization sequence. Moreover, MiCOMP has a simple mapping function that encodes an optimization sequence into a bit string, allowing us to apply standard machine learning techniques that require fixed length feature vectors. We incorporated analogies between the analyzed problem and the context of Recommender Systems, and integrate similarity measures to boost exploration efficiency.

We show that MiCOMP can outperform LLVM's standard optimization levels with just a few predicted of optimizations sequences and achieves top 80% of the available speedup by traversing less than 5% of the optimization sequence space. This is rather crucial when the optimization space can consist of enormous number of different optimization sequences.

6.8 Dissemination of The Chapter

The excerpt of this chapter has been submitted to ACM Transaction on Architecture and Code Optimization (TACO) under the title *MiCOMP: Mitigating Compiler Phase-ordering using Machine Learning and Optimization Sub-sequences* and is currently under review. Refer to the Chapter 7.2 for the list of publications of my PhD dissertation.

CHAPTER 7

Conclusion and Future Work

In this PhD dissertation, we have tackled the major problems of compiler autotuning. We have used machine learning, DSE, and meta-heuristic techniques to construct efficient and accurate models to induce prediction models.

This chapter unfolds in two main sections. Section 7.1 summarizes the main contribution of the thesis and 7.2 provides a list of open issues and our future direction towards tackling those challenges.

7.1 Main Contributions

The main conclusions of the research presented in this dissertation can be summarized as follow.

1. We provides an extensive survey on the literature by elaborating all the research papers of the past 25 years. We classified them by their type of contribution, approaches taken and many sub-features. We hope that the survey ca be useful for a wide range of researchers and industry professionals.

2. We provided a co-exploration framework in order to statistically analyzing the compiler level parameters over customized VLIW architecture. We show that not all the available compiler parameters are beneficial to use on the very embedded domain. Moreover, we applied several statistical tests e.g. ANOVA, Kruskal-Wallis and clustering techniques to customize a VLIW architecture and select a set of promising compiler parameters specifically for the derived architecture.

3. The COBAYN is a framework for autotuning compiler passes using Bayesian networks. It uses application's characteristics to predict the best set of optimization passes to be used. We experimentally showed that this framework can outperform GCC's standard optimization levels e.g. `-O2 and -O3`.

4. The intermediate speedup predictor approach, known presented in the Chapter 5 aims to provide a first contribution towards tackling the phase-ordering problem. Systematically choosing the best local alternative as the next-best optimization to select might lead to a local minimum, whereas other paths, with less steep initials might end up in a better global point. Providing this chapter is to familiarize with the difficult problem of the phases involved with the compiler optimization field and an intuitive approach for tackling such.

5. The MiCOMP is a framework for mitigating the phase-ordering problem. It uses predictive modeling to form the best ordering of phases for an application under analysis. It has the advantage of predicting the full optimization sequence at once and approach towards global minimum versus the previous intermediate approach presented in Chapter 5. We experimentally showed that the framework can outperform state-o-the-art techniques and standard optimization levels.

7.2 Open Issues and Future Direction

From the research standpoint presented in this dissertation, some open issues have been identified. In this section, we list these issues, together with the possible future research directions.

1. Using COBAYN and MiCOMP framework, many research questions in the field of compiler autotuning can be answered. However, these open problems can be further tackled and the approaches can be faster and more accurate. Inducing prediction models are kind of approximating the problem's results, therefore there wouldn't be a definite optimal result for the given problems. Introducing new machine learning paradigms e.g. deep learning, convolutional networks, etc. can be used to further improve the achievable results on the same optimization space.

2. Parallel and heterogeneous computing brought both new challenges and applications of autotuning for the compiler optimization field. Now, it is up to the compiler researchers to adapt and introduce novel heterogeneous tuning both on CPU and GPU applications. This direction is indeed both challenging and interesting for us.

3. Multi-objective optimization strategies need to be further addresses. Power-aware and energy efficient optimizations are still open challenges in the field to be addresses.

We hope that the dissertation will pave the way to give more fine-grain knowledge and understanding of the discussed problems and light up more ideas for young researchers to carry on the path towards tackling the existing open issues.

Publications

Articles Published/Under Review in International Journals

1. Amir Hossein, Ashouri; William, Killian; John, Cavazos; Gianluca, Palermo and Cristina, Silvano, under peer-review 2016, "A Survey on Compiler Autotuning using Machine Learning", under peer-review at ACM Transactions on Computing Surveys (CSUR)

2. Amir Hossein, Ashouri; Andrea, Bignoli; Gianluca, Palermo; Cristina, Silvano; Sameer, Kulkarni and John, Cavazos, 2016, "MiCOMP: Mitigating Compiler Phase-ordering using Machine Learning and Optimization Sub-sequences", under peer-review at ACM Transactions on Architecture and Code Optimization (TACO)

3. Amir Hossein, Ashouri; Giovanni, Mariani; Gianluca, Palermo; Eunjung, Park; John, Cavazos and Cristina, Silvano, June 2016, "COBAYN: Compiler Autotuning Framework Using Bayesian Networks", 13(2):21, ACM Transactions on Architecture and Code Optimization (TACO)

Articles Published in proceedings of International Conferences

1. Amir Hossein, Ashouri; Giovanni, Mariani; Gianluca, Palermo and Cristina, Silvano; 2014, "A Bayesian Network Approach for Compiler Auto-tuning for Embedded Processors", IEEE- ESTIMedia; New Delhi, INDIA, 90-97

2. Amir Hossein, Ashouri; Vittorio, Zaccaria; Sotirios, Xydis; Gianluca, Palermo and Cristina, Silvano, 2013, "A Framework for Compiler Level Statistical Analysis over Customized VLIW Architecture", IEEE- VLSI-SoC; Istanbul, TURKEY, 124-129

Articles Published in Proceedings of International Workshops

1. Amir Hossein, Ashouri; Gianluca, Palermo and Cristina, Silvano, 2016, "An Evaluation of Autotuning Techniques for the Compiler Optimization Problems",

RES4ANT Workshop Co-located with ACM DATE; Dresden, Germany, CEURS-WS 1543: 23-27

2. Amir Hossein, Ashouri; Andrea, Bignoli; Gianluca, Palermo and Cristina, Silvano, 2016, "Predictive Modeling Methodology for Compiler Phase-Ordering", ACM PARMA-DITAM- Workshop of HiPEAC, ACM; Prague, Czech Republic, 7-12

Posters Published/Presented in Poster Sessions co-located with International Conference

1. Amir Hossein, Ashouri; Andrea, Bignoli; Gianluca, Palermo and Cristina, Silvano, 2016, "A Predictive Modeling Framework For Compiler Phase-ordering Problem", Student Research Competition (SRC) Poster Session co-located with IEEE/ACM CGO, Barcelona, Spain

2. Amir Hossein, Ashouri; Gianluca, Palermo and Cristina, Silvano, 2016, "Auto-tuning Techniques for Compiler Optimization", PhD Forum co-located with ACM DATE, Dresden, Germany

3. Amir Hossein, Ashouri; Vittorio, Zaccaria; Sotirios, Xydis; Gianluca, Palermo and Cristina, Silvano, 2013, "Design Space Exploration and Analysis Of Compiler Transformation in VLIW Processors", Friday Workshop (DEPCP) co-located with ACM DATE; Grenoble, FRANCE

Reports

- Amir Hossein Ashouri, HiPEAC short-term collaboration report. HiPEAC newsletter No. 44. Online: http://home.deib.polimi.it/ashouri/hipeacinfo44.pdf#page=11

Bibliography

[1] Hewlett-packard laboratories. vex toolchain. [online], available:. http://www.hpl.hp.com/downloads/vex/.

[2] Multicube explorer. http://m3explorer.sourceforge.net/.

[3] Core architecture, 2012. http://www.kalray.eu/technology/.

[4] *Opentuner: An extensible framework for program autotuning*, 2014.

[5] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[6] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M Parikh, and James M Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *ACM SIGPlAN Notices*, volume 33, pages 280–290. ACM, 1998.

[7] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. Using machine learning to focus iterative optimization, 2006.

[8] Alan Agresti, I Liu, et al. Modeling a categorical variable allowing arbitrarily many category choices. *Biometrics*, 55(3):936–943, 1999.

[9] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[10] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[11] L Almagor and KD Cooper. Finding Effective Compilation Sequences. 2004.

[12] George Almasi and David A Padua. Majic: A matlab just-in-time compiler. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 68–81. Springer, 2000.

[13] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[14] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[15] Karl-Erik Årzén and Anton Cervin. Control and embedded computing: Survey of research directions. *IFAC Proceedings Volumes*, 38(1):191–202, 2005.

[16] G. Ascia, V. Catania, M. Palesi, and D. Patti. A system-level framework for evaluating area/performance/power trade-offs of vliw-based embedded systems. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 2, pages 940–943 Vol. 2, Jan.

[17] A.H. Ashouri, G. Mariani, G. Palermo, and C. Silvano. A Bayesian network approach for compiler autotuning for embedded processors. In *2014 IEEE 12th Symposium on Embedded Systems for Real-Time Multimedia, ESTIMedia 2014*, pages 90–97, 2014.

[18] Amir Hossein Ashouri. Design space exploration methodology for compiler parameters in vliw processors. Master's thesis, Politecnico Di Milano, ITALY, 2012. `http://hdl.handle.net/10589/72083`.

[19] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. Predictive modeling methodology for compiler phase-ordering. In *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, PARMA-DITAM '16, pages 7–12, New York, NY, USA, 2016. ACM.

[20] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim. (TACO)*, 13(2):21:1–21:25, June 2016.

[21] Amir Hossein Ashouri, Gianluca Palermo, and Cristina Silvano. An Evaluation of Autotuning Techniques for the Compiler Optimization Problems. In *Res4ant*, pages 23–27, 2016.

[22] Amir Hossein Ashouri, Vittorio Zaccaria, Sotirios Xydis, Gianluca Palermo, and Cristina Silvano. A framework for Compiler Level statistical analysis over customized VLIW architecture. In *VLSI-SoC*, pages 124–129, 2013.

[23] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[24] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, dec 1994.

[25] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[26] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[27] Rajendra Bhatia. *Positive definite matrices*. Princeton University Press, 2009.

[28] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[29] U Bondhugula and M Baskaran. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146, 2008.

[30] U Bondhugula and A Hartono. A practical automatic polyhedral parallelizer and locality optimizer, 2008.

[31] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[32] Karsten M Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 8–pp. IEEE, 2005.

[33] Norman Breslow. A generalized kruskal-wallis test for comparing k samples subject to unequal patterns of censorship. *Biometrika*, 57(3):579–594, 1970.

[34] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*, pages 5–13. Ieee, 2008.

[35] Gustavo Camps-Valls, Tatyana V Bandos Marsheva, and Dengyong Zhou. Semi-supervised graph-based hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 45(10):3044–3054, 2007.

[36] J Cavazos and JEB Moss. Inducing heuristics to decide whether to schedule. *ACM SIGPLAN Notices*, 2004.

[37] J Cavazos, JEB Moss, and MFP O'Boyle. Hybrid optimizations: Which optimization algorithm to use? *Compiler Construction*, 2006.

[38] J Cavazos and MFP O'Boyle. Automatic tuning of inlining heuristics. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 14–14, 2005.

[39] J Cavazos and MFP O'boyle. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices*, 2006.

[40] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, Grigori Fursin, and Olivier Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 24–34, 2006.

[41] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. *International Symposium on Code Generation and Optimization (CGO'07)*, 2007.

[42] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[43] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009.

[44] C Chen, J Chame, and M Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. *International Symposium on Code Generation and Optimization*, pages 111–122, 2005.

[45] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[46] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):21, 2012.

[47] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 448–459, New York, NY, USA, 2010. ACM.

[48] B.R. Childers and M.L. Soffa. A Model-Based Framework: An Approach for Profit-Driven Optimization. In *International Symposium on Code Generation and Optimization*, pages 317–327. IEEE, 2005.

[49] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[50] Katherine E Coons, Behnam Robatmili, Matthew E Taylor, Bertrand A Maher, Doug Burger, and Kathryn S McKinley. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 32–42. ACM, 2008.

[51] KD Cooper, A Grosul, and TJ Harvey. ACME: adaptive compilation made efficient. 40(7):69–77, 2005.

[52] KD Cooper, PJ Schielke, and D Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 1999.

[53] KD Cooper, D Subramanian, and L Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 2002.

[54] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[55] Howard Demuth and Mark Beale. Neural network toolbox for use with matlab. 1993.

[56] Open64 Developers. Open64 compiler and tools, 2001.

[57] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[58] Y Ding, J Ansel, and K Veeramachaneni. Autotuning algorithmic choice for input sensitivity. *ACM SIGPLAN Notices*, 50(6):379–390, 2015.

[59] Kaivalya M Dixit. The spec benchmarks. *Parallel computing*, 17(10):1195–1209, 1991.

[60] C Dubach, J Cavazos, and B Franke. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th international conference on Computing frontiers*, pages 131–142, 2007.

[61] Christophe Dubach, Timothy M Jones, Edwin V Bonilla, Grigori Fursin, and Michael FP O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. pages 78–88, 2009.

[62] Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, ISBN-13: 978-1-59327-289-0, 2011.

[63] S Fang, W Xu, Y Chen, and L Eeckhout. Practical iterative optimization for the data center. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(2):15, 2015.

[64] Paolo Faraboschi, Geoffrey Brown, Joseph A Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable vliw embedded processing. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 203–213. ACM, 2000.

[65] Paolo Faraboschi and Fred Homewood. St200: A vliw architecture for media-oriented applications. In *Microprocessor Forum 2000. San Jose, CA*, 2000.

[66] Paul Feautrier. Parametric integer programming. *RAIRO Recherche opérationnelle*, 22(3):243–268, 1988.

[67] Damon Fenacci, Björn Franke, and John Thomson. Workload characterization supporting the development of domain-specific compiler optimizations using decision trees for data mining. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, page 5. ACM, 2010.

[68] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[69] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[70] JA Fisher, P Faraboschi, and C Young. Vliw processors: Once blue sky, now commonplace. *Solid-State Circuits Magazine, IEEE*, 1(2):10–17, 2009.

[71] Joseph A Fisher. Microcode compaction. *IEEE Transactions on Computers*, 30(7), 1981.

[72] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann, 2004.

[73] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[74] B Franke, M O'Boyle, J Thomson, and G Fursin. Probabilistic source-level optimisation of embedded programs. *ACM SIGPLAN Notices*, 2005.

[75] Christopher W. Fraser. Automatic inference of models for statistical code compression. *ACM SIGPLAN Notices*, 34(5):242–246, may 1999.

[76] Stefan M Freudenberger and John C Ruttenberg. Phase ordering of register allocation and instruction scheduling. In *Code Generation Concepts, Tools, Techniques*, pages 146–170. Springer, 1992.

[77] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine learning*, 29(2-3):131–163, 1997.

[78] G Fursin, J Cavazos, M O'Boyle, and O Temam. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. *International Conference on High-Performance Embedded Architectures and Compilers*, pages 245–260, 2007.

[79] G Fursin and A Cohen. Building a practical iterative interactive compiler. *Workshop Proceedings*, 2007.

[80] G Fursin, A Cohen, M O'Boyle, and O Temam. A practical method for quickly evaluating program optimizations. *International Conference on High-Performance Embedded Architectures and Compilers*, pages 29–46, 2005.

[81] G Fursin, Y Kashnikov, and AW Memon. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[82] G Fursin, C Miranda, and O Temam. MILEPOST GCC: machine learning based research compiler. *GCC Summit*, 2008.

[83] G Fursin and O Temam. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):20, 2010.

[84] GG Fursin. Iterative Compilation and Performance Prediction for Numerical Applications. 2004.

[85] GG Fursin, MFP O'Boyle, and PMW Knijnenburg. Evaluating iterative compilation. *International Workshop on Languages and Compilers for Parallel Computing*, pages 362–376, 2002.

[86] Grigori Fursin. Collective benchmark (cbench), a collection of open-source programs with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization, 2010.

[87] Grigori Fursin and Olivier Temam. Collective optimization. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 34–49. Springer, 2009.

[88] Unai Garciarena and Roberto Santana. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, GECCO '16 Companion, pages 1159–1166, New York, NY, USA, 2016. ACM.

[89] Lise Getoor. *Introduction to statistical relational learning*. MIT press, 2007.

[90] Richard L Gorsuch. Exploratory factor analysis. In *Handbook of multivariate experimental psychology*, pages 231–258. Springer, 1988.

[91] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[92] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[93] M Hall, D Padua, and K Pingali. Compiler research: the next 50 years. *Communications of the ACM*, 2009.

[94] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[95] M Haneda. Optimizing general purpose compiler optimization. *Proceedings of the 2nd conference on Computing frontiers*, pages 180–188, 2005.

[96] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Unsupervised learning. In *The elements of statistical learning*, pages 485–585. Springer, 2009.

[97] David Heckerman and David M. Chickering. Learning bayesian networks: The combination of knowledge and statistical data. In *Machine Learning*, pages 20–197, 1995.

[98] Jeffrey Hightower and Gaetano Borriello. A survey and taxonomy of location systems for ubiquitous computing. *IEEE computer*, 34(8):57–66, 2001.

[99] Torsten Hoefler and Roberto Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 73. ACM, 2015.

[100] K Hoste and L Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, 2008.

[101] K Hoste, A Georges, and L Eeckhout. Automated just-in-time compiler tuning. *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 62–72, 2010.

[102] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[103] Ronald A Howard. Dynamic programming and markov processes.. 1960.

[104] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. The superblock: an effective technique for vliw and superscalar compilation. *the Journal of Supercomputing*, 7(1-2):229–248, 1993.

[105] Rob J Hyndman and Anne B Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, 22(4):679–688, 2006.

[106] Texas Instruments. Pandaboard. *OMAP4430 SoC dev. board, revision A*, 2:2012, 2012.

[107] Texas Instruments. Pandaboard. *OMAP4430 SoC dev. board, revision A*, 2, 2012.

[108] Sverre Jarp. A methodology for using the itanium 2 performance counters for bottleneck analysis. Technical report, Technical report, HP Labs, 2002.

[109] Brian Jeff. Big. little system architecture from arm: saving power through heterogeneous multiprocessing and task context migration. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1143–1146. ACM, 2012.

[110] Zhanpeng Jin and A.C. Cheng. Improve simulation efficiency using statistical benchmark subsetting - an implantbench case study. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 970–973, June 2008.

[111] Richard Arnold Johnson and Dean W Wichern. *Applied multivariate statistical analysis*, volume 5 of *8*. Prentice hall Upper Saddle River, NJ, 2002.

[112] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. Java (tm) language specification. *Addisson-Wesley, June*, 2000.

[113] Henry F Kaiser. The varimax criterion for analytic rotation in factor analysis. *Psychometrika*, 23(3):187–200, 1958.

[114] Agnieszka Kamińska and Włodzimierz Bielecki. Statistical models to accelerate software development by means of iterative compilation. *Computer Science*, 17(3):407, 2016.

[115] Tapas Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, and A.Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):881–892, 2002.

[116] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

[117] Christos Kartsaklis, Oscar Hernandez, Chung-Hsing Hsu, Thomas Ilsche, Wayne Joubert, and Richard L Graham. Hercules: A pattern driven code transformation system. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 574–583. IEEE, 2012.

[118] William Killian, Renato Miceli, Eunjung Park, Marco Alvarez, and John Cavazos. Performance Improvement in Kernels by Guiding Compiler Auto-Vectorization Heuristics. *PRACE-RI.EU*, 2014.

[119] Toru Kisuki, P Knijnenburg, M O'Boyle, and H Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44. Citeseer, 2000.

[120] Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00), Philadelphia, Pennsylvania, USA, October 15-19, 2000*, pages 237–248, 2000.

[121] Toru Kisuki, Peter MW Knijnenburg, Mike FP O'Boyle, François Bodin, and Harry AG Wijshoff. A feasibility study in iterative compilation. In *High Performance Computing*, pages 121–132. Springer, 1999.

[122] P M W Knijnenburg, T Kisuki, and M F P O 'boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. *The Journal of Supercomputing*, 24:43–67, 2003.

[123] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[124] A Koseki. A method for estimating optimal unrolling times for nested loops. In *Parallel Architectures, Algorithms, and Networks, 1997.(I-SPAN'97) Proceedings., Third International Symposium on*, pages 376–382, 1997.

[125] DH Kulkarni, S Tandri, L Martin, N Copty, R Silvera, XM Tian, X Xue, and J Wang. Xl fortran compiler for ibm smp systems. *AIXpert Magazine*, 1997.

[126] P Kulkarni, S Hines, and J Hiser. Fast searches for effective optimization phase sequences. *ACM SIGPLAN Notices*, 39(6):171–182, 2004.

[127] Prasad A Kulkarni, Michael R Jantz, and David B Whalley. Improving both the performance benefits and speed of optimization phase sequence searches. In *ACM Sigplan Notices*, volume 45, pages 95–104. ACM, 2010.

[128] Prasad A. Kulkarni, David B. Whalley, and Gary S. Tyson. Evaluating Heuristic Optimization Phase Order Search Algorithms. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 157–169. IEEE, mar 2007.

[129] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Trans. Archit. Code Optim.*, 6(1):1:1–1:36, April 2009.

[130] S Kulkarni and J Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices*, 2012.

[131] S Kulkarni and J Cavazos. Automatic construction of inlining heuristics using machine learning. *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–12, 2013.

[132] J Larmouth. Fortran 77 portability. *Software: Practice and Experience*, 11(10):1071–1117, 1981.

[133] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[134] H Leather, E Bonilla, and M O'Boyle. Automatic feature generation for machine learning based optimizing compilation. *International Symposium on Code Generation and Optimization, CGO'09*, pages 81–91, 2009.

[135] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[136] Corinna Lee and Mark Stoodley. Utdsp benchmark suite, 1998.

[137] Junghsi Lee and V John Mathews. A stability condition for certain bilinear systems. *IEEE transactions on signal processing*, 42(7):1871–1873, 1994.

[138] Bruce W Leverett, Roderic Geoffrey Galton Cattell, Steven O Hobbs, Joseph M Newcomer, Andrew H Reiner, Bruce R Schatz, and William A Wulf. *An overview of the production quality compiler-compiler project*. Carnegie Mellon University, Department of Computer Science, 1979.

[139] Fengqian Li, Feilong Tang, and Yao Shen. Feature mining for machine learning based compilation optimization. *Proceedings - 2014 8th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2014*, pages 207–214, 2014.

[140] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[141] Y Li, J Dongarra, and S Tomov. A note on auto-tuning GEMM for GPUs. *Computational Science (ICCS) 2009*, 2009.

[142] The LLVM website, 2013. http://www.llvm.org/.

[143] Vincent Loechner. Polylib: A library for manipulating parameterized polyhedra, 1999.

[144] P Lokuciejewski and F Gedikli. Automatic WCET reduction by machine learning based heuristics for function inlining. *3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, pages 1–15, 2009.

[145] P Lokuciejewski and S Plazar. Approximating Pareto optimal compiler optimization sequences a trade off between WCET, ACET and code size. *Software: Practice and Experience*, 41(12):1437–1458, 2011.

[146] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Multi-objective exploration of compiler optimizations for real-time systems. *ISORC 2010 - 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 1:115–122, 2010.

[147] David B Loveman. Program improvement by source-to-source transformation. *Journal of the ACM (JACM)*, 24(1):121–145, 1977.

[148] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[149] L Luo, Y Chen, C Wu, S Long, and G Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. *arXiv preprint arXiv:1407.4075*, 2014.

[150] J Mars and R Hundt. Scenario based optimization: A framework for statically enabling online optimizations. *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

[151] LGA Martins and R Nobre. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1), 2016.

[152] Luiz GA Martins, Ricardo Nobre, Alexandre CB Delbem, Eduardo Marques, and João MP Cardoso. Exploration of compiler optimization sequences using clustering-based selection. In *ACM SIGPLAN Notices*, volume 49, pages 63–72. ACM, 2014.

[153] Amy McGovern, Eliot Moss, and Andrew G Barto. Scheduling straight-line code using reinforcement learning and rollouts. *Tech report No-99-23*, 1999.

[154] Amy McGovern, Eliot Moss, and Andrew G Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine learning*, 49(2-3):141–160, 2002.

[155] Xiangrui Meng, Joseph Bradley, B Yuvaz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[156] R Miceli, G Civario, A Sikora, and E César. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. *International Workshop on Applied Parallel Computing*, pages 328–342, 2012.

[157] Minimal ir space, 2011. http://www.assembla.com/wiki/show/minir-dev.

[158] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[159] A Monsifrot, F Bodin, and R Quiniou. A machine learning approach to automatic production of compiler heuristics. *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50, 2002.

[160] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, D Stefanovic, Carla Brodley, and David Scheeff. Learning to schedule straight-line code. *Advances in Neural Information Processing Systems*, 10:929–935, 1998.

[161] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, pages 7–10, 1999.

[162] Kevin P. Murphy. The bayes net toolbox for matlab. *Computing Science and Statistics*, 33:2001, 2001.

[163] M Namolaru, A Cohen, and G Fursin. Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, 2010.

[164] Ricardo Nobre, Luiz G A Martins, and João M P Cardoso. A Graph-Based Iterative Compiler Pass Selection and Phase Ordering Approach. pages 21–30, 2016.

[165] Ricardo Nobre, Luiz GA Martins, and Joao MP Cardoso. Use of previously acquired positioning of optimizations for phase ordering exploration. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 58–67. ACM, 2015.

[166] Ricardo Nobre, Luıs Reis, and Joao MP Cardoso. Compiler phase ordering as an orthogonal approach for reducing energy consumption. In *CPC*, 2016.

[167] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. Openjit: An open-ended, reflective jit compiler framework for java. In *European Conference on Object-Oriented Programming*, pages 362–387. Springer, 2000.

[168] William Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Minimizing the cost of iterative compilation with active learning. In *CGO*. IEEE, 2016.

[169] Holger Orup. On-the-fly one-hot encoding of leading zero count, October 26 1999. US Patent 5,974,432.

[170] Karl Joseph Ottenstein. Data-flow graphs as an intermediate program form. 1978.

[171] David A Padua and Michael J Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[172] G Palermo, C Silvano, S Valsecchi, and V Zaccaria. A system-level methodology for fast multi-objective design space exploration. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 92–95. ACM, 2003.

[173] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Multi-objective design space exploration of embedded systems. *Journal of Embedded Computing*, 1(3):305–316, 2005.

[174] Z Pan and R Eigenmann. Rating compiler optimizations for automatic performance tuning. *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 14, 2004.

[175] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[176] Su-lin PANG and Ji-zhang GONG. C5. 0 classification algorithm and application on individual credit evaluation of banks. *Systems Engineering-Theory & Practice*, 29(12):94–104, 2009.

[177] E Park, J Cavazos, and MA Alvarez. Using graph-based program characterization for predictive modeling. *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, 2012.

[178] E Park, J Cavazos, and LN Pouchet. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming*, pages 704–750, 2013.

[179] E Park, S Kulkarni, and J Cavazos. An evaluation of different modeling techniques for iterative compilation. pages 65–74, 2011.

[180] Eun Jung Park. Automatic selection of compiler optimizations using program characterization and machine learning title. 2015.

[181] Eunjung Park, Christos Kartsaklis, and John Cavazos. HERCULES: Strong Patterns towards More Intelligent Predictive Modeling. *2014 43rd International Conference on Parallel Processing*, pages 172–181, 2014.

[182] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[183] Judea Pearl. Bayesian netwcrks: A model of self-activated memory for evidential reasoning. *UCLA Technical Report CSD-850017). Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine, CA.*, (3):329–334, 1985.

[184] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[185] R. P J Pinkers, P. M W Knijnenburg, M. Haneda, and H. A G Wijshoff. Statistical selection of compiler options. *Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*, pages 494–501, 2004.

[186] L. L. Pollock and M. L. Soffa. Incremental global optimization for faster recompilations. In *Computer Languages, 1990., International Conference on*, pages 281–290, Mar 1990.

[187] LN Pouchet and C Bastoul. Iterative optimization in the polyhedral model: Part I, one-dimensional time. *International Symposium on Code Generation and Optimization (CGO'07)*, pages 144–156, 2007.

[188] LN Pouchet, C Bastoul, A Cohen, and J Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. *ACM SIGPLAN Notices*, 2008.

[189] LN Pouchet and U Bondhugula. Combined iterative and model-driven optimization in an automatic parallelization framework. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[190] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.

[191] S Purini and L Jain. Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):56, 2013.

[192] Matthieu Stéphane Benoit Queva. Phase-ordering in optimizing compilers, 2007.

[193] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[194] Ranjit K Roy. *Design of experiments using the Taguchi approach: 16 steps to product and process improvement*. Wiley-Interscience, 2001.

[195] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. Using machines to learn method-specific compilation strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 257–266, 2011.

[196] Roberto Santana, Concha Bielza, Pedro Larranaga, Jose A Lozano, Carlos Echegoyen, Alexander Mendiburu, Rubén Armananzas, and Siddartha Shakya. Mateda-2.0: Estimation of distribution algorithms in matlab. *Journal of Statistical Software*, 35(7):1–30, 2010.

[197] Debyo Saptono, Vincent Brost, Fan Yang, and Eri Prasetyo. Design space exploration for a custom vliw architecture: Direct photo printer hardware setting using vex compiler. In *Proceedings of the 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*, SITIS '08, pages 416–421, Washington, DC, USA, 2008. IEEE Computer Society.

[198] V Sarkar. Optimized unrolling of nested loops. *Proceedings of the 14th international conference on Supercomputing*, pages 153–166, 2000.

[199] Vivek Sarkar. Automatic selection of high-order transformations in the ibm xl fortran compilers. *IBM Journal of Research and Development*, 41(3):233–264, 1997.

[200] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[201] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[202] Robert R Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[203] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Ruckstieb, and Jurgen Schmidhuber. Pybrain. *Journal of Machine Learning Research*, 11(Feb):743–746, 2010.

[204] E Schkufza, R Sharma, and A Aiken. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices*, 2014.

[205] Paul B Schneck. A survey of compiler optimization techniques. In *Proceedings of the ACM annual conference*, pages 106–113. ACM, 1973.

[206] Michael J Schulte, Vitaly Zelov, Ahmet Akkas, and James Craig Burley. The interval-enhanced gnu fortran compiler. *Reliable Computing*, 5(3):311–322, 1999.

[207] Bernard W Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

[208] Steve Smale and Ding-Xuan Zhou. Estimating the approximation error in learning theory. *Analysis and Applications*, 1(01):17–41, 2003.

[209] Donald F Specht. Probabilistic neural networks. *Neural networks*, 3(1):109–118, 1990.

[210] Richard Stallman. Using and porting the gnu compiler collection. In *MIT Artificial Intelligence Laboratory*. Citeseer, 2001.

[211] Richard M Stallman et al. *Using GCC: the GNU compiler collection reference manual*. Gnu Press, 2003.

[212] Kenneth O Stanley. Efficient reinforcement learning through evolving neural network topologies. In *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002*. Citeseer, 2002.

[213] M Stephenson and S Amarasinghe. Meta optimization: improving compiler heuristics with machine learning. 38(5):77–90, 2003.

[214] M Stephenson and S Amarasinghe. Predicting unroll factors using supervised classification. In *International symposium on code generation and optimization*, 2005.

[215] M Stephenson and UM O'Reilly. Genetic programming applied to compiler heuristic optimization. *European Conference on Genetic Programming*, 2003.

[216] MW Stephenson. Automating the construction of compiler heuristics using machine learning. 2006.

[217] Ralph E Steuer. *Multiple criteria optimization: theory, computation, and applications*. Wiley, 1986.

[218] K Stock, LN Pouchet, and P Sadayappan. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):50, 2012.

[219] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the ibm java just-in-time compiler. *IBM systems Journal*, 39(1):175–193, 2000.

[220] Mirai Tanaka and Kazuhide Nakata. Positive definite matrix approximation with condition number constraint. *Optimization Letters*, 8(3):939–947, 2014.

[221] R Core Team et al. R: A language and environment for statistical computing. 2013.

[222] Gerald Tesauro and Gregory R Galperin. On-line policy improvement using monte-carlo search. In *NIPS*, volume 96, pages 1068–1074, 1996.

[223] Bruce Thompson. "statistical,""practical," and "clinical": How many kinds of significance do counselors need to consider? *Journal of Counseling & Development*, 80(1):64–71, 2002.

[224] J Thomson, M O'Boyle, G Fursin, and B Franke. Reducing training time in a one-shot machine learning-based compiler. *International Workshop on Languages and Compilers for Parallel Computing*, pages 399–407, 2009.

[225] A Tiwari, C Chen, and J Chame. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, 2009.

[226] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP MFP O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan Notices*, pages 177–187, 2009.

[227] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215. IEEE Comput. Soc, 2003.

[228] Eben Upton and Gareth Halfacree. *Raspberry Pi user guide*. John Wiley & Sons, 2014.

[229] K Vaswani. Microarchitecture sensitive empirical models for compiler optimizations. *International Symposium on Code Generation and Optimization (CGO'07)*, pages 131–143, 2007.

[230] Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. *ACM SIGMICRO Newsletter*, 13(4):125–133, 1982.

[231] Richard Vuduc, James W Demmel, and Jeff A Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[232] Z Wang and MFP O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. *ACM Sigplan notices*, 2009.

[233] D. Whitfield, M. L. Soffa, D. Whitfield, and M. L. Soffa. An approach to ordering optimizing transformations. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming - PPOPP '90*, volume 25, pages 137–146, New York, New York, USA, 1990. ACM Press.

[234] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *ACM SIGPLAN Notices*, volume 26, pages 120–129. ACM, 1991.

[235] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, nov 1997.

[236] Doran K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, December 1993.

[237] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[238] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, et al. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.

[239] MI Wolczko and DM Ungar. Method and apparatus for improving compiler performance during subsequent compilations of a source program. *US Patent 6,078,744*, 2000.

[240] Stephan Wong, Thijs Van As, and Geoffrey Brown. ρ-vex: A reconfigurable and extensible softcore vliw processor. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 369–372. IEEE, 2008.

[241] T Yuki, V Basupalli, G Gupta, G Iooss, and D Kim. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. 2012.

[242] T Yuki, G Gupta, DG Kim, T Pathan, and S Rajopadhye. AlphaZ: A System for Design Space Exploration in the Polyhedral ModelâĂŤ. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 17–31, 2012.

[243] Vittorio Zaccaria, Gianluca Palermo, Fabrizio Castro, Cristina Silvano, and Giovanni Mariani. Multicube explorer: An open source framework for design space exploration of chip multi-processors. In *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*, pages 1–7. VDE, 2010.

[244] Wei Zhang, Deli Zhao, and Xiaogang Wang. Agglomerative clustering via maximum incremental path integral. *Pattern Recognition*, 46(11):3056–3065, 2013.

[245] Min Zhao, Bruce Childers, Mary Lou Soffa, Min Zhao, Bruce Childers, and Mary Lou Soffa. Predicting the impact of optimizations for embedded systems. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems LCTES '03*, volume 38, page 1, New York, New York, USA, 2003. ACM Press.