SPRINGER BRIEFS IN APPLIED SCIENCES AND TECHNOLOGY · POLIMI SPRINGER BRIEFS

Amir H. Ashouri
Gianluca Palermo
John Cavazos
Cristina Silvano

Automatic Tuning of Compilers Using Machine Learning

POLITECNICO
MILANO 1863

Springer

SpringerBriefs in Applied Sciences and Technology

PoliMI SpringerBriefs

More information about this series at http://www.springer.com/series/11159
http://www.polimi.it

Amir H. Ashouri · Gianluca Palermo
John Cavazos · Cristina Silvano

Automatic Tuning of Compilers Using Machine Learning

Springer

Amir H. Ashouri
Edward S. Rogers Sr. Department
of Electrical and Computer
Engineering (ECE)
University of Toronto
Toronto, ON
Canada

Gianluca Palermo
Department of Electronics, Information
and Bioengineering (DEIB)
Politecnico di Milano
Milan
Italy

John Cavazos
Computer and Information Sciences (CIS)
University of Delaware
Newark, DE
USA

Cristina Silvano
Department of Electronics, Information
and Bioengineering (DEIB)
Politecnico di Milano
Milan
Italy

ISSN 2191-530X ISSN 2191-5318 (electronic)
SpringerBriefs in Applied Sciences and Technology
ISSN 2282-2577 ISSN 2282-2585 (electronic)
PoliMI SpringerBriefs
ISBN 978-3-319-71488-2 ISBN 978-3-319-71489-9 (eBook)
https://doi.org/10.1007/978-3-319-71489-9

Library of Congress Control Number: 2017959897

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Learning never exhausts the mind
Imparare non stanca mai la mente
—Leonardo da Vinci

Foreword

Compilers have two jobs: translating programs into a form understandable by machines and making the translated code run efficiently. This second role, compiler optimization is a long-standing research problem. It has led to a large number of compiler heuristics or optimizations, each of which is designed to improve system performance. While each of these optimizations may deliver good performance individually, when combined they may degrade performance. Determining what optimizations to use and in what order depends on the program and the target platform. The different combinations and orderings quickly create a massive optimization space greater than the number of atoms in the known universe. The complexity of this problem prevents innovation in compiler research and leads to a loss of performance. In recent years, researchers have looked to search and machine learning-based approaches to navigate this complex space and select the best combination and sequence of optimizations.

This book tackles the difficult problem of determining the best set of compiler optimizations for a range of platforms. It addresses this problem using innovative machine learning-based solutions that exploit prior knowledge. This knowledge is used to build models that predict the right optimizations for unseen programs. It succinctly describes the fundamental research problem and extensively surveys the large body of prior work. This survey provides an excellent background to the topic.

This book makes four specific technical contributions. The first considers how to co-design VLIW micro-architecture and compiler optimizations using a performance/area Pareto curve. The second contribution is the use of a novel Bayesian network to predict the best optimizations using a method that explains how program features correlate with output. The third contribution is the use of a performance predictor to guide and select compiler optimizations without running the code. The final contribution is the most ambitious chapter, tackling phase order based on a unique optimization clustering approach.

This book provides an excellent state-of-the-art survey of compiler optimization, develops innovative solutions to long-standing problems, and most importantly of all opens up new lines of research in compiler optimization.

Michael O'Boyle
University of Edinburgh,
Edinburgh UK

October 2017

Preface

The diversity of today's architectures has forced programmers to spend additional efforts to port and tune their application source code across different platforms. In this process, compilers need additional tuning to generate better code. Recent compilers offer a vast number of multilayered optimizations, capable of targeting different code segments of an application. Choosing the right set of optimizations can significantly impact the performance of the code. This is a challenge made more complicated by the need to find the best order in which they should be applied given an application. Finding the best ordering is a long-standing problem in compilation research called the phase-ordering problem. The traditional approach for constructing compiler heuristics to solve it simply cannot cope with the enormous complexity of choosing the right ordering of optimizations for each code segment in an application. The current research focuses on exploring, studying, and developing an innovative approach to the problem of identifying the compiler optimizations that maximize the performance of a target application.

Overview of this Book

This book addresses two fundamental problems involved in compilation research: the problem of *selecting* the best compiler optimizations and the *phase-ordering* problem. Statistical analyses were extensively used to relate the performance of an application to the applied optimizations. More precisely, machine learning models were adapted to predict an outcome. Here, an outcome is described either in terms of performance metrics, or the use of a certain compiler optimization. Similar to other machine learning approaches, we use a set of training applications to learn the statistical relationships between application features and compiler optimizations. For instance, Bayesian networks are used to learn the statistical model to which an

application can be represented with. We call these representations an application feature. Thus, given a new application not included in the training set, its features are fed to the Bayesian network as evidence. This evidence generates a bias on the distribution, as compiler optimizations are correlated with software features. The obtained probability distribution is application-specific and effectively focuses on the prediction of the most promising compiler optimizations. This will be discussed in detail in Chap. 3.

Who is this Book for?

This is a textbook that aims to showcase the very recent developments of research approaches in the compilation research, specifically autotuning. Therefore, all researchers in the compiler community, computer architecture, parallel computing, and machine learning can benefit from reading it. Additionally, given the potential industrial impact of the provided approaches, it is recommended to read to other technical professionals as well.

Summary and Organization

This book tackles the major problems of compiler autotuning. We use machine learning, DSE, and meta-heuristic techniques to construct efficient and accurate models to induce prediction models.

It is organized as follows: First, we provide an extensive review of the state of the art in Chap. 1. We survey more than hundred recent papers of the past twenty-five years since when the applications of machine learning have been introduced to compiler optimization field. Following the literature review, in Chap. 2, we provide a co-exploration approach using design space exploration technique for an embedded domain, namely VLIW. We show that this technique can speed up the performance of an application by using certain optimizations pass over our proposed VLIW micro-architecture. In Chap. 3, we present a novel machine learning approach to 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 selects the most promising compiler passes. Chapters 4 and 5 are presenting our novel machine learning predictive models on how to tackle the phase-ordering problem. The former presents an intermediate approach, and the latter showcases a complete sequence speedup predictor on the very problem.

Finally, we present some concluding remarks and future works. Note that in this book, the bibliography is chapter-wise.

We hope this book brings the latest research done to a wide range of readers and promotes the use of machine learning on the field of compilation.

Toronto, ON, Canada — Amir H. Ashouri
Milan, Italy — Gianluca Palermo
Newark, DE, USA — John Cavazos
Milan, Italy — Cristina Silvano

Acknowledgements

The majority of the research related to this book has been carried out in the Department of Electronics, Information and Bioengineering (DEIB) at Politecnico di Milano. Additionally, I had the chance to be a Visiting Scholar in the Department of Electrical Engineering and Computer Science at the University of Delaware, USA. The collaboration allowed me to carry out further elaborations and analyses. The work described in this book was partially supported by the European Commission Call H2020-FET-HPC program under the grant ANTAREX-671623.

I would like to thank all my colleagues and advisers including postdoctoral and Ph.D. fellows with whom I had the opportunity to collaborate, specially Cristina Silvano, Gianluca Palermo, John Cavazos, Giovanni Mariani, Sotiris Xydis, Marco Alvarez, Eunjung Park, Sameer Kulkarni, William Kilian, Andrea Bignoli, and Robert Searles. The teamwork was truly fun and challenging at the same time, and I was grateful to participate in numerous constructive discussions. Many thanks to Michael O'Boyle and Erven Rohou for their valuable comments and provided reviews. Their insight on the compiler optimization field is truly inspiring.

Last but not least, I would like to appreciate the lifetime support of my lovely family: mother, father, and the younger brother who always have been my backbone during the hard times and the good times. Thank you for giving me the positive energy to carry on and for urging me to choose this path for my life.

Amir H. Ashouri
University of Toronto
Toronto, Canada

October 2017

Contents

Chapter 1
Background

Abstract Since the mid-1990s, 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 use) 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. The chapter highlights the approaches taken, the obtained results, the holistic comparisons among different approaches and finally, the visionary path towards the near future.

1.1 Introduction

Recent developments in silicon production and fabrication led to the creation of much faster computational units such as CPUs, GPUs, FPGAs, and similar devices with different 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, the growing complexity of modern compilers and the concern over security are among the most important problems that compilers should answer. Moore's law [1] 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 [2].

A. H. Ashouri et al., *Automatic Tuning of Compilers Using Machine Learning*,
PoliMI SpringerBriefs, https://doi.org/10.1007/978-3-319-71489-9_1

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 [2, 3] 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 method 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 represents a challenging problem [4].

Autotuning addresses automatic code-generation and optimization by using different scenarios and architectures. It constructs techniques for automatic optimization of different parameters to maximize or minimize the satisfaction of an objective function. Historically, optimizations were mostly in the backend where *scheduling*, *resource-allocation*, and *code-generation* are done [5, 6]. The constraints form a linear problem (ILP), thus one should solve an ILP to find an optimized design. Recently, researchers have shown increased effort in introducing frontend and IR optimizations. The reason is twofold: (i) the complexity of a backend compiler requires strict and exclusive knowledge of compiler designers, and (ii) frontend optimizations have lower overheads compared with the optimizations performed in the back-end. Nonetheless, each has its benefits and drawbacks and is subject to further analyses.

The major challenge in choosing the right set of compiler optimizations is the fact that the optimizations are programming language, application, and architecture dependent. Additionally, the word optimization is a misnomer; there is no guarantee that the transformed code will perform better than the original version. Aggressive optimizations can degrade the performance of the code to which they are applied [7]. 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 optimizations that can be applied during the different compilation phases. This optimization ordering dilemma creates the phase-ordering problem.

There were a number of long-standing compiler optimization problems have not yet been adequately addressed. These problems comprise of knowing *what* optimizations (by which configuration (i.e., the tile size in loop tiling)), and in *which order* the optimizations should be used to obtain 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. The *phase-ordering* problem

has been an open problem for many decades [8–11]. The inability of researchers to solve the phase-ordering problem has led to advances in the less complex problem of selecting the right set of optimizations, but even this has yet to be fully resolved [12–14].

In order to understand the difficulty of the problems, one should consider that the 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, i.e., `-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 used recently in literature to find the best optimizations to apply to an application is inducing prediction models using different classes of machine learning [15]. Approaches which leverage machine learning to find the best optimizations to apply will be the center focus of this book.

Contribution. In this chapter, we provide an overview of techniques and approaches appeared so far to tackle the aforementioned problems. We elaborate many 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 autotuning problem was done by [16–18]. However, there were other original works which tackled the problem of compiler autotuning without the use of machine learning technique [10, 19–22], and we believe them to be the driving force of using machine learning to 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 [23, 24] on the compiler optimization field.

We first discuss the motivation and challenges involved in the compiler optimization research in Sect. 1.2, followed by an analysis of the optimization space for the two major optimization problems. Then, we discuss the machine learning models used in Sect. 1.3 and provide a full classification of different techniques used in the recent research. Finally, we conclude the chapter with discussion and future trends.

[1] https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html.

[2] http://llvm.org/docs/Passes.html.

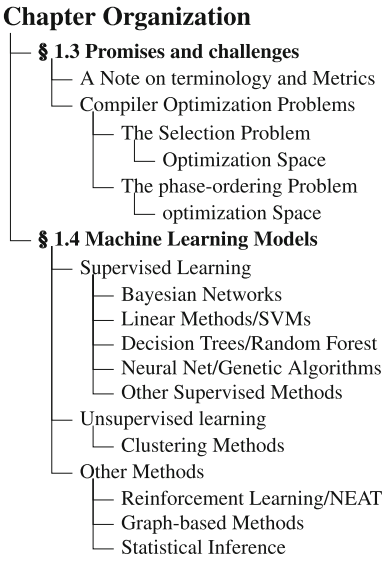


Fig. 1.1 Organization of the chapter in different sections

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

Organization of the chapter. In these sections, we organize the state-of-the-art works in different categories to underscore their similarities and differences. Note that the works presented in these sections are deeply intertwined. While we study a certain work under a single group, several of these works belong to multiple groups. We organize the chapter in a way that all research papers corresponding to a specific type of classification are cited. However, we selectively focused on the most notable works under each section and we provide more elaboration on their contribution. Fig. 1.1 represents our organization of the chapter.

1.2 Compiler Optimizations

1.2.1 A Note on Terminology and Metrics

Since publications mentioned in this chapter originated from varying authors, terminologies were locally defined and might not be strictly defined. We clarify terms used in this chapter 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 have been used widely and our reference subsequently.

1.2.2 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 [20] allow a user to affect the generated code without changing the original high-level source code. When these optimizations are applied may result in a program running better on a target architecture. Since a user is not able to manual tune a large code, automatic methods need to be 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. We discuss the challenges and benefits at two levels:

Research Level

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.

User's Level

Compiler writers expose general-purpose transformations to end users. Ultimately, a subset of these transformations leads to better architecture fitting given source code. Over time we have seen an 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 [25] and OpenACC [26], automatically transform users' code to exploit parallelism. One of the most 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.

1.2.3 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

Table 1.1 A classification based on the type of the problem

Classification	References
The selection problem	[16, 29–80]
The phase-ordering problem	[7–11, 22, 33, 81–88]

traditional optimizing compilers between constant folding and flow analysis, and between register allocation and code generation [10, 27].

In optimization theory [28], 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. A feasible set 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 1.1 classifies the existing literature based on the type of the problem. As we mentioned earlier, the inability of researchers to completely solve the phase-ordering problem has led to some advances in the problem of selecting the right set of compiler optimizations. Thus, we see more research focusing on the selection problem recently.

1.2.3.1 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 the performance of a running code even by ignoring the order of phases [12, 29].

Optimizations Space

Let us define a Boolean vector 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 o belongs to the n dimensional Boolean space of:

$$|\Omega_{Selection}| = \{0, 1\}^n \tag{1.1}$$

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 $A = a_0 \ldots a_N$ under study.

Extended version of the current definition in Eq. 1.1 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 turn the previous equation as:

$$|\Omega_{Selection\_Extended}| = \{0, 1, \ldots, m\}^n \tag{1.2}$$

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

1.2.3.2 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 both the selection and the ordering are of importance, the phase-ordering problem is formed. It is one 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 [10, 27].

Optimizations Space

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

$$|\Omega_{Phases}| = n! \tag{1.3}$$

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. Enabling optimizations to be repeatedly applied and a variable length sequence of optimizations will expand the problem space to:

$$|\Omega_{Phases\_Repetition\_variableLength}| = \sum_{i=0}^{m} n^i \tag{1.4}$$

where n is the number of optimizations under study and m is the maximum desired length for the optimization sequence. Even for reasonable values of 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 [82].[3]

1.3 Machine Learning Models

Machine learning explores the study and construction of algorithms that can learn from and make predictions on data [89]. 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 1.2. 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.

1.3.1 Supervised Learning

Supervised learning is the machine learning task of inferring a function from labeled training data [89, 112]. 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.

1.3.1.1 Bayesian Networks

Bayesian Networks (BN) [113, 114] 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

[3]The phase-ordering problem does not have a deterministic upper-bound when we have an unbounded optimization length.

Table 1.2 A classification based on machine learning models

Classification		References
Supervised learning	Bayesian networks	[12, 90]
	Linear models/SVMs	[12, 68, 72, 82, 86, 91]
	Decision trees/Random forests	[45, 46, 57, 63, 77, 92–94, 94–97]
	Neural networks/Genetic algorithm	[8, 9, 16, 29, 30, 40, 52, 54, 55, 62, 66, 71, 73, 78, 83–85, 87, 95, 96, 98–107]
	Others	[12, 35, 48, 53, 59, 71, 75–78, 86, 91, 104, 106–109]
Unsupervised	Clustering methods	[75, 84, 87, 110, 111]
Others methods	Reinforcement learning/NEAT	[9, 95, 107]
	Graph-based methods	[62, 84, 111]
	Statistical methods	[9, 12, 29, 31, 32, 34, 35, 40–48, 50, 51, 56, 59, 61, 64, 68, 70, 71, 74, 77, 78, 90, 93, 95, 96, 98, 101, 107, 108, 110]

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. [12, 90] proposed a Bayesian network approach to address the problem of selecting the best compiler optimizations suitable for an 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 [115] and Polybench [116, 117] 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%.

1.3.1.2 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 very stable [112]. 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, namely the maximum margin hyperplane, and shows the largest separation from the set of hyperplanes [91].

1.3.1.3 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 [89]. Random forests or random decision forests are an ensemble learning methods 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 [112].

Fraser et al. [93] 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 a stream that compress better. They evaluated their approach with GCC and used Opcodes which can also help predict elements of the operand stream.

Monsifrot [97] 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 [94] 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. [96, 118] 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. [57] 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 a decision tree namely, rule induction based machine learning techniques to rapidly select best code versions at run-time based on dataset features and minimize selection overhead.

1.3.1.4 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 almost all in some way determine pattern statistics from a set of training samples and then classify new patterns on the basis of these statistics [119]. 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 [120].

Cooper et al. [16, 101] 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. [54] 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. [71] 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 [121] applications to evaluate their approach.

Cavazos and O'Boyle [99] developed a genetic algorithm 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 as candidates and the geometric mean of the performance of the `SPECjvm98` benchmarks was used as their fitness function.

Agakov et al. [29] introduced machine-learning models to focus on the exploration of the compiler optimizations for the most promising region. Their methodology exploits a Markov chain oracle and an independent identically distributed (IID) probability distribution oracle. These two models learned offline bias using only a certain optimizations rather than using 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. The probability distribution learned is then used as the predicted optimal distribution for the new application. It has been reported that the Markov chain oracle outper-

forms the IID oracle, followed by the RIC methodology using a uniform probability distribution.

Kulkarni et al. [8] used a depth-first search algorithm to produce the next sequence to evaluate in an exhaustive exploration of the phase ordering problem. To evaluate their method, the authors used hill-climbing, simulated annealing, genetic-algorithm and a random search on an embedded architecture.

Leather et al. [104] 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. [87] 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.

1.3.1.5 Other Supervised Methods

For conciseness proposes, we decided to classify other supervised learning methods in this subsection. These include Lazy learning, ANOVA, K-nearest neighbor, Gaussian process learning [89], and others.

Moss et al. [18] 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 [35] used JIT Java compiler and SPECjvm98 benchmark and rule set induction learning model to decide whether to schedule. The induced function chooses for each block between list scheduling or not scheduling the block. 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. [76] proposed a technique using profile-driven parallelism detection in which they could overcome the limitations of static analysis, and allowing the identification of more application parallelism. This approach only relied on the user for final approval. It 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.

1.3.2 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 [89, 122]. Unsupervised learning is closely tightened with the problem of density estimation in statistics [123]. However, unsupervised learning also encompasses many other techniques that seek to summarize and explain key features of the data.

1.3.2.1 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 correspond to each other, i.e., targets loop-nests or scalar values, or they should follow each other in the same sequence. Another benefit of clustering/downsampling is to reduce the compiler optimization space which as mentioned in Sect. 1.2.3 are in tens of thousands orders of magnitudes.

Thomson et al. [75] present a new approach to reduce the training time of a machine learning based compiler. They focused on the programs which best characterized 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 they can reduce the number of training runs by more than a factor of seven.

Ashouri et al. [32, 110] 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 Sect. 1.3.3.3.)

Martins et al. [84, 111] 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.

1.3.3 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 [89].

1.3.3.1 Reinforcement Learning and NEAT

Reinforcement learning (RL) is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment to maximize some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics, and genetic algorithms [89]. The interesting difference in RL is that the training and testing phases are intermixed. To collect information the learner should actively interact with the environment.

We decided to group RL with a neuroevolution approach called NEAT as suggested by [124] 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 historical 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).

Kulkarni et al. developed two approaches to tackle both problems of selection [95] and the phase-ordering [9] 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. Here, they 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. By using a stop-condition, the NEAT knows when to stop expanding the sequence and converging to the final sequence.

1.3.3.2 Graph-Based Methods

Recently, graph-based methods have emerged as a powerful means of exploiting many different machine learning applications on a wide range of applications from

semi-supervised learning [125] to clustering and classification [126]. 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. [62] 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. 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 transformations.

Nobre et al. [105] proposed an iterative compilation approach using graph-based features of the optimized code to mitigate the selecting and the ordering of the compiler optimization passes in LLVM. After iterative evaluations, the authors could find solutions with a speedup factor over the baseline and they provided a clustering method to organize their finding.

1.3.3.3 Statistical Methods

Terminology across fields is quite varied for statistical methods [127]. 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 [15, 89], 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 the literature which involved with the most frequent 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 an individual section for each to classify in a more fine-grained manner.

Pinker et al. [64] 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. [51] 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. [59] proposed a general method for systematically generating numerical features from a program. The authors implemented the approach 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.

Ashouri et al. [32, 110] introduced a statistical technique to cluster and choose the 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 the ANOVA and Kruskal-wallis tests. Using a performance distribution graphs as their fitness function, they selected a number of compiler parameters to be used for a VLIW [128, 129] architecture.

1.4 Conclusions

Using compiler optimizations to exploit large-scale parallelism available on architectures and power-aware hardware is an essential task. In the coming decades, research on compilation techniques and code optimization will play a key role in alleviating various challenges within computer science and the high performance computing field. This includes auto-parallelization, security, exploiting multi/many-core processors, reliability, reproducibility, and energy efficiency.

By the advancement we have seen in GPUs, deep learning has emerged as a viable mean of addressing many classification problems. These complex learners allow automated systems to efficiently perform tasks with minimal programmer effort.

In this chapter, we have synthesized the research work on compiler autotuning using machine learning and we showed 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 when machine learning models were used. 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. In the next chapters, we tackle the aforementioned problems by means of a design space explorations and machine learning approaches.

References

1. Schaller RR (1997) Moore's law: past, present and future. IEEE Spectr 34(6):52–59
2. Hall M, Padua D, Pingali K (2009) Compiler research: the next 50 years. Commun ACM 52:60–67
3. Aho AV, Sethi R, Ullman JD (1986) Compilers, principles, techniques. Addison-Wesley, Boston
4. Palermo G, Silvano C, Zaccaria V (2005) Multi-objective design space exploration of embedded systems. J Embedded Comput 1(3):305–316
5. Chaitin GJ, Auslander MA, Chandra AK, Cocke J, Hopkins ME, Markstein PW (1981) Register allocation via coloring. Comput Lang 6(1):47–57

6. Freudenberger SM, Ruttenberg JC (1992) Phase ordering of register allocation and instruction scheduling. In: Code generation concepts, tools, techniques. Springer, pp 146–170
7. Triantafyllis S, Vachharajani M, Vachharajani N, August DI (2003) Compiler optimization-space exploration. In: International symposium on code generation and optimization, 2003. CGO 2003. IEEE Computing Society, pp 204–215
8. Kulkarni PA, Whalley DB, Tyson GS (2007) Evaluating heuristic optimization phase order search algorithms. In International symposium on code generation and optimization (CGO'07). IEEE, Mar 2007, pp 157–169
9. Kulkarni S, Cavazos J (2012) Mitigating the compiler optimization phase-ordering problem using machine learning. ACM SIGPLAN Notices
10. Vegdahl SR (1982) Phase coupling and constant generation in an optimizing microcode compiler. ACM SIGMICRO Newsl 13(4):125–133
11. Whitfield D, Soffa ML (1990) An approach to ordering optimizing transformations. In: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming—PPOPP '90, vol 25. ACM Press, New York, New York, USA, pp 137–146
12. Ashouri AH, Mariani G, Palermo G, Park E, Cavazos J, Silvano C (2016) Cobayn: compiler autotuning framework using Bayesian networks. ACM Trans Archit Code Optim (TACO) 13(2):21:1–21:25. http://doi.acm.org/10.1145/2928270 (June)
13. Bodin F, Kisuki T, Knijnenburg P, O'Boyle M, Rohou E (1998) Iterative compilation in a non-linear optimisation space. In: Workshop on profile and feedback-directed compilation
14. Chen Y, Fang S, Huang Y, Eeckhout L, Fursin G, Temam O, Wu C (2012) Deconstructing iterative optimization. ACM Trans Archit Code Optim (TACO) 9(3):21
15. Alpaydin E (2014) Introduction to machine learning. MIT Press, Cambridge
16. Cooper KD, Schielke PJ, Subramanian D (1999) Optimizing for reduced code space using genetic algorithms. ACM SIGPLAN Notices
17. Koseki A (1997) A method for estimating optimal unrolling times for nested loops. In: Proceedings of third international symposium on parallel architectures, algorithms, and networks, 1997 (I-SPAN'97), pp 376–382
18. Moss E, Utgoff P, Cavazos J, Precup D, Stefanovic D, Brodley C, Scheeff D (1998) Learning to schedule straight-line code. Adv Neural Inf Process Syst 10:929–935
19. Loveman DB (1977) Program improvement by source-to-source transformation. J ACM (JACM) 24(1):121–145
20. Padua DA, Wolfe MJ (1986) Advanced compiler optimizations for supercomputers. Commun ACM 29(12):1184–1201
21. Pollock LL, Soffa ML (1990) Incremental global optimization for faster recompilations. In: International conference on computer languages, 1990, pp 281–290 (March)
22. Whitfield DL, Soffa ML (1997) An approach for exploring code improving transformations. ACM Trans Program Lang Syst 19(6):1053–1084
23. Bacon DF, Graham SL, Sharp OJ (1994) Compiler transformations for high-performance computing. ACM Comput Surv 26(4):345–420
24. Schneck PB (1973) A survey of compiler optimization techniques. In: Proceedings of the ACM annual conference. ACM, pp 106–113
25. Dagum L, Menon R (1998) Openmp: an industry standard api for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
26. Wienke S, Springer P, Terboven C, an Mey D (2012) OpenACC—first experiences with real-world applications. European conference on parallel processing. Springer, Berlin, pp 859–870
27. Leverett BW, Cattell RGG, Hobbs SO, Newcomer JM, Reiner AH, Schatz BR, Wulf WA (1979) An overview of the production quality compiler-compiler project. Carnegie Mellon University, Department of Computer Science
28. Steuer RE (1986) Multiple criteria optimization: theory, computation, and applications. Wiley, Hoboken
29. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI (2006) Using machine learning to focus iterative optimization. In: Proceedings of the international symposium on code generation and optimization. IEEE Computer Society, pp 295–305

30. Almagor L, Cooper KD (2004) Finding effective compilation sequences. ACM SIGPLAN Notices 39(7):231–239
31. Ansel J, Kamil S (2014). In Proceedings of the 23rd international conference on parallel architectures and compilation, pp 303–316
32. Ashouri AH (2012) Design space exploration methodology for compiler parameters in vliw processors. Master's thesis, Politecnico Di Milano, Italy. http://hdl.handle.net/10589/72083
33. Ashouri AH (2016) Compiler autotuning using machine learning techniques. Ph.D. thesis, Politecnico Di Milano, Italy. http://hdl.handle.net/10589/129561
34. Cavazos J, Fursin G, Agakov F (2007) Rapidly selecting good compiler optimizations using performance counters. In: International symposium on code generation and optimization (CGO'07)
35. Cavazos J, Moss JEB (2004) Inducing heuristics to decide whether to schedule. ACM SIGPLAN Notices
36. Cavazos J, Moss JEB, O'Boyle MFP (2006) Hybrid optimizations: which optimization algorithm to use?. Compiler construction. Springer, Berlin, pp 124–138
37. Cavazos J, O'Boyle MFP (2006) Method-specific dynamic compilation using logistic regression. ACM SIGPLAN Notices
38. Chen Y, Huang Y, Eeckhout L, Fursin G, Peng L, Temam O, Wu C (2010) Evaluating iterative optimization across 1000 datasets. In: Proceedings of the 2010 ACM SIGPLAN conference on programming language design and implementation, PLDI '10. ACM, New York, NY, USA, pp 448–459
39. Childers BR, Soffa ML (2005) A model-based framework: an approach for profit-driven optimization. In: International symposium on code generation and optimization. IEEE, pp 317–327
40. Dubach C, Cavazos J, Franke B (2007) Fast compiler optimisation evaluation using code-feature based performance prediction. In: Proceedings of the 4th international conference on Computing frontiers, pp 131–142
41. Dubach C, Jones TM, Bonilla EV (2009) Portable compiler optimisation across embedded programs and microarchitectures using machine learning, pp 78–88
42. Fang S, Xu W, Chen Y, Eeckhout L (2015) Practical iterative optimization for the data center. ACM Trans Archit Code Optim (TACO) 12(2):15
43. Franke B, O'Boyle M, Thomson J, Fursin G (2005) Probabilistic source-level optimisation of embedded programs. ACM SIGPLAN Notices
44. Fursin G, Cavazos J, O'Boyle M, Temam O (2007) Midatasets: creating the conditions for a more realistic evaluation of iterative optimization. In: International conference on high-performance embedded architectures and compilers, pp 245–260
45. Fursin G, Kashnikov Y, Memon AW (2011) Milepost gcc: machine learning enabled self-tuning compiler. Int J Parallel Prog 39(3):296–327
46. Fursin G, Miranda C, Temam O (2008) MILEPOST GCC: machine learning based research compiler. GCC Summit
47. Fursin G, Temam O (2010) Collective optimization: a practical collaborative approach. ACM Trans Archit Code Optim (TACO) 7(4):20
48. Fursin GG (2004) Iterative compilation and performance prediction for numerical applications
49. Fursin GG, O'Boyle MFP, Knijnenburg PMW (2002) Evaluating iterative compilation. In: International workshop on languages and compilers for, parallel computing, pp 362–376
50. Fursin G, Temam O (2009) Collective optimization. In: International conference on high-performance embedded architectures and compilers. Springer, pp 34–49
51. Haneda M (2005) Optimizing general purpose compiler optimization. In: Proceedings of the 2nd conference on Computing frontiers, pp 180–188
52. Hoste K, Eeckhout L (2008) Cole: compiler optimization level exploration. In: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, pp 165–174
53. Killian W, Miceli R, Park E, Alvarez M, Cavazos J (2014) Performance improvement in kernels by guiding compiler auto-vectorization heuristics. In: PRACE-RI.EU

54. Knijnenburg PMW, Kisuki T, O'Boyle MFP (2003) Combined selection of tile sizes and unroll factors using iterative compilation. J Supercomput 24:43–67
55. Li F, Tang F, Shen Y (2014) Feature mining for machine learning based compilation optimization. In: Proceedings—2014 8th international conference on innovative mobile and internet services in ubiquitous computing, IMIS 2014, pp 207–214
56. Lokuciejewski P, Plazar S, Falk H, Marwedel P, Thiele L (2010) Multi-objective exploration of compiler optimizations for real-time systems. In: ISORC 2010—2010 13th IEEE international symposium on object/component/service-oriented real-time distributed computing, vol 1, pp 115–122
57. Luo L, Chen Y, Wu , Long S, Fursin G (2014) Finding representative sets of optimizations for adaptive multiversioning applications. arXiv preprint arXiv:1407.4075
58. Mars J, Hundt R (2009) Scenario based optimization: a framework for statically enabling online optimizations. In: Proceedings of the 7th annual IEEE/ACM international symposium on code generation and optimization
59. Namolaru M, Cohen A, Fursin G (2010) 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
60. Pan Z, Eigenmann R (2004) Rating compiler optimizations for automatic performance tuning. In: Proceedings of the 2004 ACM/IEEE conference on supercomputing, p 14
61. Pan Z, Eigenmann R (2006) Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: International symposium on code generation and optimization (CGO'06). IEEE, 12–pp
62. Park E, Cavazos J, Alvarez MA (2012) Using graph-based program characterization for predictive modeling. In: Proceedings of the international symposium on code generation and optimization, pp 295–305
63. Park EJ, Kartsaklis C, Cavazos J (2014) HERCULES: strong patterns towards more intelligent predictive modeling. In: 2014 43rd international conference on parallel processing, pp 172–181
64. Pinkers RPJ, Knijnenburg PMW, Haneda M, Wijshoff HAG (2004) Statistical selection of compiler options. In: Proceedings—IEEE computer society's annual international symposium on modeling, analysis, and simulation of computer and telecommunications systems, MASCOTS, pp 494–501
65. Pouchet LN, Bastoul C (2007) Iterative optimization in the polyhedral model: part I, one-dimensional time. In: International symposium on code generation and optimization (CGO'07), pp 144–156
66. Pouchet LN, Bastoul C, Cohen A, Cavazos J (2008) Iterative optimization in the polyhedral model: part II, multidimensional time. ACM SIGPLAN Notices
67. Pouchet LN, Bondhugula U (2010) Combined iterative and model-driven optimization in an automatic parallelization framework. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis, pp 1–11
68. Sanchez RN, Amaral JN, Szafron D, Pirvu M, Stoodley M (2011) Using machines to learn method-specific compilation strategies. In: Proceedings of the 9th Annual IEEE/ACM international symposium on code generation and optimization, pp 257–266
69. Sarkar V (2000) Optimized unrolling of nested loops. In: Proceedings of the 14th international conference on Supercomputing, pp 153–166
70. Schkufza E, Sharma R, Aiken A (2014) Stochastic optimization of floating-point programs with tunable precision. ACM SIGPLAN Notices
71. Stephenson M, Amarasinghe S (2003) Meta optimization: improving compiler heuristics with machine learning. ACM SIGPLAN Notices 38:77–90
72. Stephenson M, Amarasinghe S (2005) Predicting unroll factors using supervised classification. In: International symposium on code generation and optimization
73. Stephenson M, O'Reilly UM (2003) Genetic programming applied to compiler heuristic optimization. In: European conference on genetic programming, pp 238–253

74. Stock K, Pouchet LN, Sadayappan P (2012) Using machine learning to improve automatic vectorization. ACM Trans Archit Code Optim (TACO) 8(4):50
75. Thomson J, O'Boyle M, Fursin G, Franke B (2009) Reducing training time in a one-shot machine learning-based compiler. In: International workshop on languages and compilers for parallel computing, pp 399–407
76. Tournavitis G, Wang Z, Franke B, O'Boyle MFP (2009) Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. ACM SIGPLAN Notices, pp 177–187
77. Vaswani K (2007) Microarchitecture sensitive empirical models for compiler optimizations. In: International symposium on code generation and optimization (CGO'07), pp 131–143
78. Wang Z, O'Boyle MFP (2009) Mapping parallelism to multi-cores: a machine learning based approach. ACM SIGPLAN Notices
79. Wolczko MI, Ungar DM (2000) Method and apparatus for improving compiler performance during subsequent compilations of a source program. US Patent 6,078,744
80. Zhao M, Childers B, Soffa ML (2003) 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, vol 38. ACM Press, New York, New York, USA, p 1
81. Ashouri AH, Bignoli A, Palermo G, Kulkarni S, Silvano C, Cavazos J (2017) Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. ACM Trans Archit Code Optim (TACO) 13(2):21:1–21:25 (October). https://doi.org/10.1145/3124452
82. Ashouri AH, Bignoli A, Palermo G, Silvano C (2016) 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. ACM, New York, NY, USA, pp 7–12. http://doi.acm.org/10.1145/2872421.2872424
83. Kulkarni P, Hines S, Hiser J (2004) Fast searches for effective optimization phase sequences. ACM SIGPLAN Notices 39(6):171–182
84. Martins LGA, Nobre R, Delbem ACB, Marques E, Cardoso JMP (2014) Exploration of compiler optimization sequences using clustering-based selection. In: ACM SIGPLAN Notices, vol 49. ACM, pp 63–72
85. Reis L, Nobre R, Cardoso JMP (2016) Compiler phase ordering as an orthogonal approach for reducing energy consumption. In: CPC
86. Park E, Cavazos J, Pouchet LN (2013) Predictive modeling in a polyhedral optimization space. Int J Parallel Prog 41:704–750
87. Purini S, Jain L (2013) Finding good optimization sequences covering program space. ACM Trans Archit Code Optim (TACO) 9(4):56
88. Queva MSB (2007) Phase-ordering in optimizing compilers
89. Mohri M, Rostamizadeh A, Talwalkar A (2012) Foundations of machine learning. MIT Press, Cambridge
90. Ashouri AH, Mariani G, Palermo G, Silvano C (2014) A Bayesian network approach for compiler auto-tuning for embedded processors. In: 2014 IEEE 12th symposium on embedded systems for real-time multimedia, ESTIMedia 2014, pp 90–97. http://doi.acm.org/10.1109/ESTIMedia.2014.6962349
91. Park E, Kulkarni S, Cavazos J (2011) An evaluation of different modeling techniques for iterative compilation. In: Proceedings of the 14th international conference on compilers, architectures and synthesis for embedded systems, pp 65–74
92. Ding Y, Ansel J, Veeramachaneni K (2015) Autotuning algorithmic choice for input sensitivity. ACM SIGPLAN Notices 50(6):379–390
93. Fraser CW (1999) Automatic inference of models for statistical code compression. ACM SIGPLAN Notices 34(5):242–246
94. Fursin G, Cohen A (2007) Building a practical iterative interactive compiler. In: Workshop proceedings

95. Kulkarni S, Cavazos J (2013) Automatic construction of inlining heuristics using machine learning. In: 2013 IEEE/ACM international symposium on code generation and optimization (CGO), pp 1–12
96. Lokuciejewski P, Gedikli F (2009) Automatic WCET reduction by machine learning based heuristics for function inlining. In: 3rd workshop on statistical and machine learning approaches to architectures and compilation (SMART), pp 1–15
97. Monsifrot A, Bodin F, Quiniou R (2002) A machine learning approach to automatic production of compiler heuristics. In: International conference on artificial intelligence: methodology, systems, and applications, pp 41–50
98. Cavazos J, Dubach C, Agakov F (2006) 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, pp 24–34
99. Cavazos J, O'Boyle MFP (2005) Automatic tuning of inlining heuristics. In: Proceedings of the ACM/IEEE SC 2005 conference on supercomputing, 2005, p 14
100. Cooper KD, Grosul A, Harvey TJ (2005) ACME: adaptive compilation made efficient. 40(7):69–77, 2005
101. Cooper KD, Subramanian D, Torczon L (2002) Adaptive optimizing compilers for the 21st century. J Supercomput 21:7–22
102. Garciarena U, Santana R (2016) 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, 2016. ACM, New York, NY, USA, pp 1159–1166
103. Hoste K, Georges A, Eeckhout L (2010) Automated just-in-time compiler tuning. In: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, pp 62–72
104. Leather H, Bonilla E, O'Boyle M (2009) Automatic feature generation for machine learning based optimizing compilation. In: International symposium on code generation and optimization, CGO'09, pp 81–91
105. Nobre R, Martins LGA, Cardoso JMP (2016) A graph-based iterative compiler pass selection and phase ordering approach. In: Proceedings of the 17th ACM SIGPLAN/SIGBED conference on languages, compilers, tools, and theory for embedded systems. ACM, pp 21–30
106. Park EJ (2015) Automatic selection of compiler optimizations using program characterization and machine learning title. Ph.D. thesis
107. Stephenson MW (2006) Automating the construction of compiler heuristics using machine learning
108. Ashouri AH, Palermo G, Silvano C (2016) An evaluation of autotuning techniques for the compiler optimization problems. In: RES4ANT2016 co-located with DATE 2016, pp 23–27. http://ceur-ws.org/Vol-1643/#paper-05
109. Fursin G, Cohen A, O'Boyle M, Temam O (2005) A practical method for quickly evaluating program optimizations. In: International conference on high-performance embedded architectures and compilers, pp 29–46
110. Ashouri AH, Zaccaria V, Xydis S, Palermo G, Silvano C (2013) A framework for Compiler Level statistical analysis over customized VLIW architecture. In: VLSI-SoC, pp 124–129. http://dx.doi.org/10.1109/VLSI-SoC.2013.6673262
111. Martins LGA, Nobre R (2016) Clustering-based selection for the exploration of compiler optimization sequences. ACM Trans Archit Code Optim (TACO) 13(1):8
112. Dietterich TG (2000) Ensemble methods in machine learning. In: International workshop on multiple classifier systems. Springer, pp 1–15
113. Friedman N, Geiger D, Goldszmidt M (1997) Bayesian network classifiers. Mach Learn 29(2–3):131–163
114. Pearl J (1985) Bayesian networks: 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, vol 3, pp 329–334

115. Fursin G (2010) 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
116. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to GPU codes. In: Innovative Parallel Computing (InPar), 2012. IEEE, pp. 1–10
117. Pouchet L-N (2012) Polybench: the polyhedral benchmark suite. http://www.cs.ucla.edu/~pouchet/software/polybench/ [cited July], 2012
118. Lokuciejewski P, Plazar S (2011) Approximating Pareto optimal compiler optimization sequences a trade off between WCET, ACET and code size. Softw Pract Experience 41(12):1437–1458
119. Specht DF (1990) Probabilistic neural networks. Neural Netw 3(1):109–118
120. Deb K, Pratap A, Agarwal S, Meyarivan TAMT (2002) A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Trans Evol Comput 6(2):182–197
121. Lee C, Potkonjak M, Mangione-Smith WH (1997) Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In: Proceedings of the 30th annual ACM/IEEE international symposium on microarchitecture. IEEE Computer Society, pp 330–335
122. Hastie T, Tibshirani R, Friedman J (2009) Unsupervised learning. In: The elements of statistical learning. Springer, pp 485–585
123. Silverman BW (1986) Density estimation for statistics and data analysis, vol 26. CRC Press, Boca Raton
124. Stanley KO (2002) Efficient reinforcement learning through evolving neural network topologies. In: Proceedings of the genetic and evolutionary computation conference (GECCO-2002). Citeseer
125. Chapelle O, Scholkopf B, Zien A (2009) Semi-supervised learning (Chapelle, O. et al., eds.; 2006) [Book reviews]. IEEE Trans Neural Netw 20(3):542–542
126. Camps-Valls G, Marsheva TVB (2007) Semi-supervised graph-based hyperspectral image classification. IEEE Trans Geosci Remote Sens 45(10):3044–3054
127. Getoor L (2007) Introduction to statistical relational learning. MIT press, Cambridge
128. Fisher JA, Faraboschi P, Young C (2009) Vliw processors: once blue sky, now commonplace. IEEE Solid-State Circuits Mag 1(2):10–17
129. Fisher JA, Faraboschi P, Young C (2004) Embedded computing: a VLIW approach to architecture, compilers and tools. Morgan Kaufmann, San Francisco

Chapter 2
Design Space Exploration of Compiler Passes: A Co-Exploration Approach for the Embedded Domain

Abstract Very Long Instruction Word (VLIW) 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. In this chapter, we provide a methodology 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 analysis on a Design of Experiments (DoEs) procedure that statistically captures 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. In the next chapters, we go deeper in building machine learning models to tackle the problem.

2.1 VLIW

Embedded system 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 differ, the computational structure and design are tightly connected with the platform they rely on. Platform-based designs have been proposed as a promising alternative for designing complex systems by redefining the problem of tuning specific design parameters of the platform template.

The scientific and commercial urge to use VLIW technology seems to be raised again after three decades of existence [1]; VLIW processor templates are being used particularly in embedded processors, designed to perform special-purpose functions, usually for real-time or hardware acceleration. Being able to use VLIW power-saving

A. H. Ashouri et al., *Automatic Tuning of Compilers Using Machine Learning*,
PoliMI SpringerBriefs, https://doi.org/10.1007/978-3-319-71489-9_2

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 are becoming a very complex task. VLIW can potentially achieve greater 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 [1, 2]. 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 [3] and introducing specific compiler optimization for VLIW processors [4, 5], 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 [6] raises the opportunity for system designers to exploit sophisticated compiler passes and customize their compiler infrastructure accordingly. Given the large optimization 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 behavior, 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 [7]. 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) a statistical analysis of the effects of compiler level transformations.

We target the design problem of compiler/architecture co-exploration in embedded computing. Thus, we focus on enabling application-specific compiler tuning over customized VLIW system architectures. First, a multi-objective exploration loop targeting application-specific micro-architectural customization is applied for extracting the best VLIW architecture candidates. We utilize the newly introduced *Roofline* processor architecture model [8] 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 [9] 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 a 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 in cross-application manner.

The rest of the chapter is organized as follows. Section 2.2 provides a brief discussion on related work and the current state of the art in the field. In Sect. 2.3, we introduce the basic methodology for architecture customization and statistical compiler level analysis. Section 2.3.2.1 presents the experimental evaluation of the proposed methodology on differing customized VLIW architectures and benchmark applications. Section 2.4 summarizes the work and concludes this chapter.

2.2 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 [1, 10, 11]. VLIW architectures are also emerging in the modern many-core embedded accelerator devices, i.e., KALRAY MPPA256 for image and signal processing applications.

Several research works have targetted the generation of Pareto optimal VLIW architectural configurations [3, 11] by exploring the space using pre-allocated compiler sequences over differing architecture instances. Towards the same direction of VLIW architectural configuration, Wong et al. [12] introduced r-VEX, a reconfigurable and extensible VLIW processor. The source code is mapped using the VEX (VLIW Example) environment [13], which forms a compilation-simulation system that targets a broad class of VLIW processor architectures, and enables compiling, simulating, analyzing and evaluating C programs [2].

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 Chap. 1. 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 [13] 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.

2.3 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 comprises 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 tailor 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 Roofline performance model [8] as the basis for both generating the custom architecture configurations and characterizing the effect of the compiler passes. Roofline relates processor performance to off-chip memory traffic. It characterizes processor architectures in a two-dimensional space, i.e., performance (Mops/sec) versus 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 Roofline 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, Fig. 2.1 depicts the Roofline model of a specific VLIW configuration and the superposition of application configurations

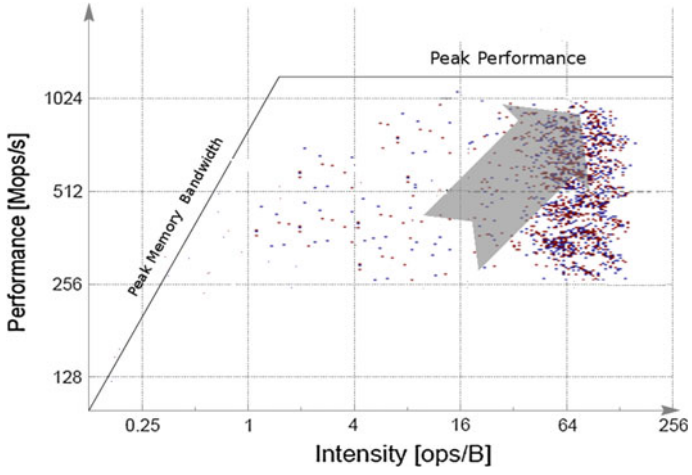


Fig. 2.1 Roofline example

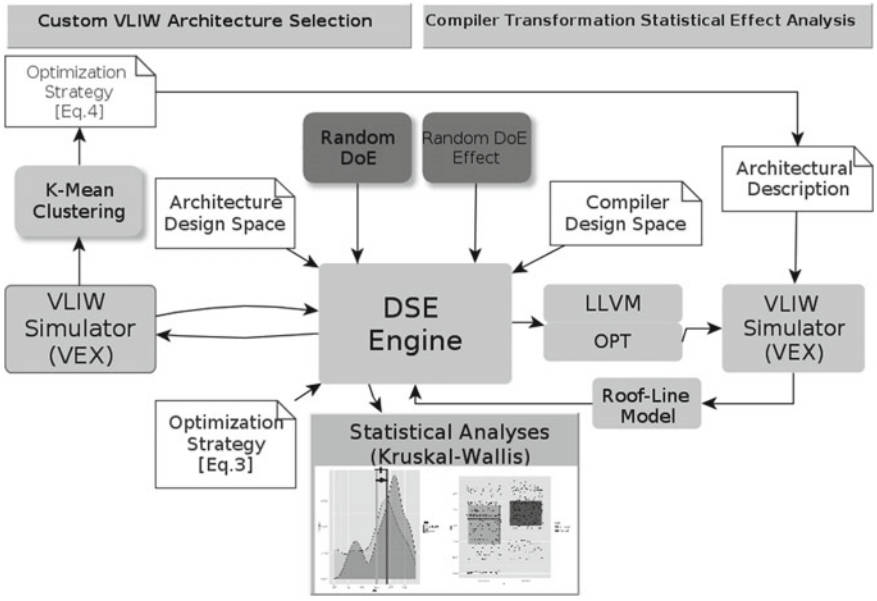


Fig. 2.2 Tool-chain implementing the proposed methodology

derived from an experimental campaign of 4 K different compiler parameter combinations. A general trend (highlighted by the arrow in Fig. 2.1) can be easily detected towards higher performance and operational intensity points. Given this visual representation, a designer can identify promising compiler passes to be applied. A custom exploration and analysis framework (Fig. 2.2) has been developed based on the integration of open source tools to implement the proposed methodology. Specifically, we used Multicube Explorer [14, 15] 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 [13] 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 Roofline model. Statistical analysis and visualization of results are performed using the statistical language R [16].

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

2.3.1 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 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 [3, 17] 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, here we invoke a relaxed optimization search strategy that is based on a random sampling of the targeted design space rather than on an optimization oriented strategy, e.g. simulated annealing or NSGA-II genetic optimization [17] etc.

Table 2.1 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 Roofline model, where:

$$Performance(\mathbf{x}) = \frac{\#Operations(\mathbf{x})}{\#NumCycles(\mathbf{x}) \times ClkFreq(\mathbf{x})} \tag{2.1}$$

$$Intensity(\mathbf{x}) = \frac{\#Operations(\mathbf{x})}{\#CacheMisses(\mathbf{x}) \times CacheLineSize(\mathbf{x})} \tag{2.2}$$

After the formation of the X, we are interested 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:

$$\min_{\mathbf{x}\in\Omega} \begin{bmatrix} \frac{1}{Performance(\mathbf{x})} \\ \frac{1}{Intensity(\mathbf{x})} \\ \#CompResources(\mathbf{x}) \\ \#MemResources(\mathbf{x}) \end{bmatrix} \tag{2.3}$$

Table 2.1 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]

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. 2.3 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. 2.3 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 [18] clustering for the aforementioned procedure, 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 each cluster, we iteratively apply the following single-objective minimization problem in every $X_p^{c_i}$ produced by the k-means clustering:

$$\min_{\mathbf{x} \in X_p^{c_i}} \frac{Area(\mathbf{x})}{Performance(\mathbf{x}) \times Intensity(\mathbf{x})} \tag{2.4}$$

For the calculation of the area cost in Eq. 2.4, the area model provided by the McPAT [19] micro-architecture framework has been used, assuming a processor technology of 90 nm.

Finding an architectures which is optimized by using the 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. Compilers perform optimization based on the program knowledge. Not all optimizations are controlled directly by an optimization pass. In this work, we select 15 compiler passes supported by LLVM compiler are as described in the Table 2.2.

2.3.1.1 DoE

Given a huge multi-objective optimization problem, it is necessary to use the design of experiment (DoE) methods, i.e., Taguchi Design of experiment [20]. 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 besides the Pareto points (if exists).

2.3.2 Compiler Transformation Statistical Effect Analysis

The 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 2.2).

As a first step in our analysis, we have to determine a reasonable number 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.

Table 2.2 Selected compiler transformations from LLVM framework

Compiler transformation	Abbreviation	Short description
Constant propagation	Constprop	Constant operands instructions are replaced with a constant value and propagated
Dead code elimination	Dce	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 applies algebraic simplification
Loop invariant dode motion	Licm	Removes 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	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	A simple loop unrolling
Unswitch loops	Loop-unswitch	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
Memory copy optimizations	Memcpyopt	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 cond const 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

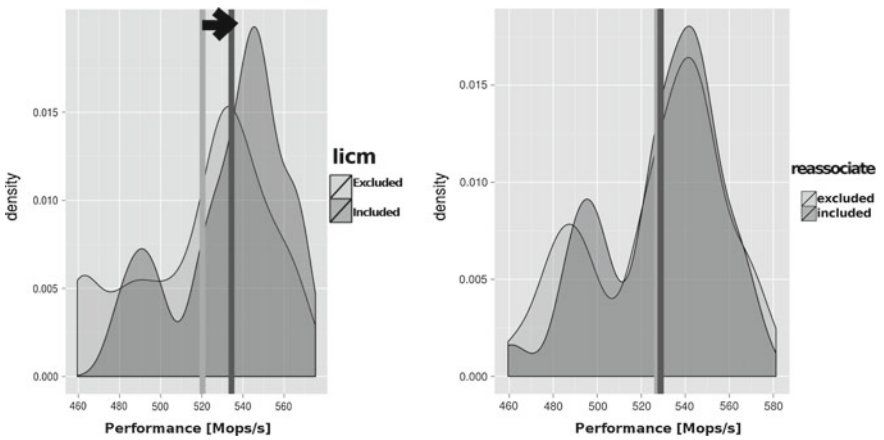
To accommodate our goal, we defined a randomized design of experiments $D_N(p)$ for each compiler parameter p. $D_N(p)$ is a list of *options sets*:

$$D(p) = \left[o_{1+}, o_{1-}, o_{2+}, o_{2-}, \ldots, o_{N+}, o_{N-}\right] \tag{2.5}$$

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, Fig. 2.3 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 optimization 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)



(a) licm (b) reassociate

Fig. 2.3 **a** licm's having significant positive effect, **b** reassociate's causing no significant effect

nature of the underlying distributions,[2] cannot be assessed with as a simple ANOVA but, instead, with a Kruskal-Wallis test [21]. 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%.

2.3.2.1 Statistical Analysis

As mentioned in Sect. 2.2, there have been several works involving the machine learning techniques and predictions [22–24]. In this research work we have focused 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 [25] test has been widely used as a reliable tester for normal distributions. In addition, using Kruskal-Wallis [21], 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 comply with the model.

In this chapter, 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 Fig. 2.5 will be focused 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 analyzing the effects of compiler transformations across differing VLIW configurations. Furthermore, we analyze the compiler transformation effects in a cross-application manner, by considering a larger set of embedded applications mapped onto a default (application independent) 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

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

in Sect. 2.3.1 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 2.1. We configure the search procedure to randomly generate and evaluate 30 K configurations, using a uniform sampling over the targeted configuration space (Table 2.1). Applying the multi-objective optimization problem defined in Eq. 2.3 over the 30 K 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 of k = 4 clusters over the generated Pareto surface, aiming at generation of four GSM-specific VLIW architectures. Figure 2.4 shows results of clustering of the extracted Pareto surface and its mapping onto the two-dimensional performance versus intensity space. Each cluster has been characterized according to its position on the performance versus 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.

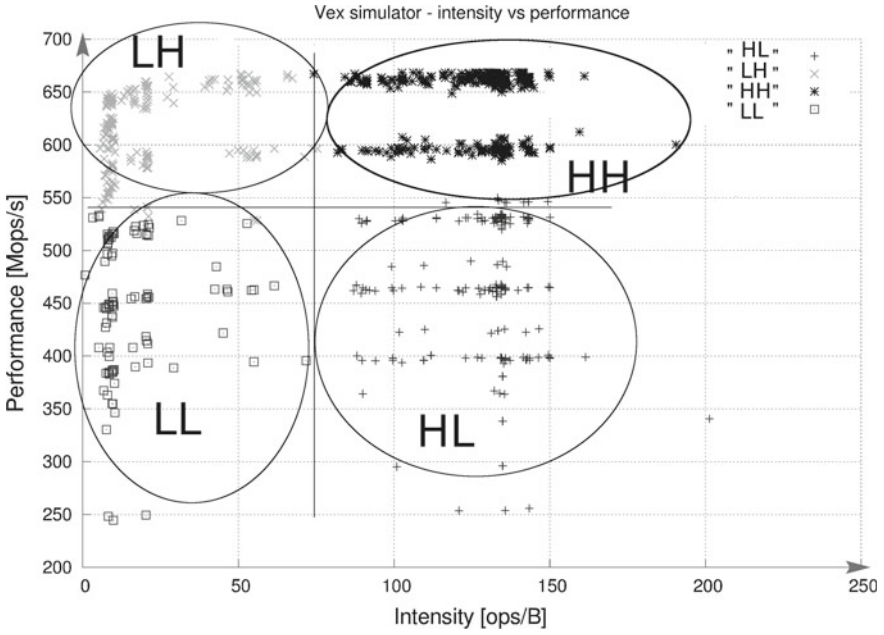


Fig. 2.4 Four clustered Pareto-sets

Table 2.3 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

The final k = 4 representative VLIW architectures are derived after applying within each cluster the optimization operator of Eq. 2.4. Table 2.3 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 2.2. We generate the non-parametric distribution of the performance and intensity for each compiler transformation considering 500 samples per transformation. As described in Sect. 2.3.2, the non-parametric distributions are analyzed 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 2.4 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, had a significant impact on performance when activated. In addition, Fig. 2.5, 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

Table 2.4 Summary of Kruskal-Wallis analysis on performance for GSM-specific VLIW architectures

Compiler transformation	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	–	–	–	–

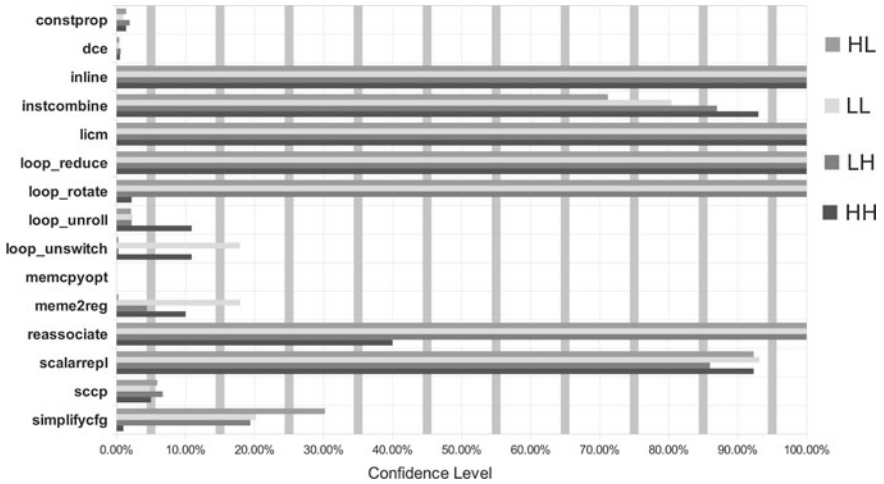


Fig. 2.5 Confidence level characterization of compiler transformations regarding the effect on performance for each on of the GSM specific VLIW architectures, resulted after Kruskal-Wallis statistical test

Table 2.5 Kruskal-Wallis analysis on performance for multiple applications

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

Blowfish block cipher). The performance of each applications has been evaluated considering a user specified VLIW architecture, Arch-User, defined in the last column of Table 2.3. For each benchmark the compiler transformation statistical effect analysis (Sect. 2.3.2) is applied, considering distributions of 500 samples per compiler transformation. Table 2.5 summarizes in an aggregated manner the results of the Kruskal-Wallis analysis considering in each case a confidense level $\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 the 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 Fig. 2.6, 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 and 23%.

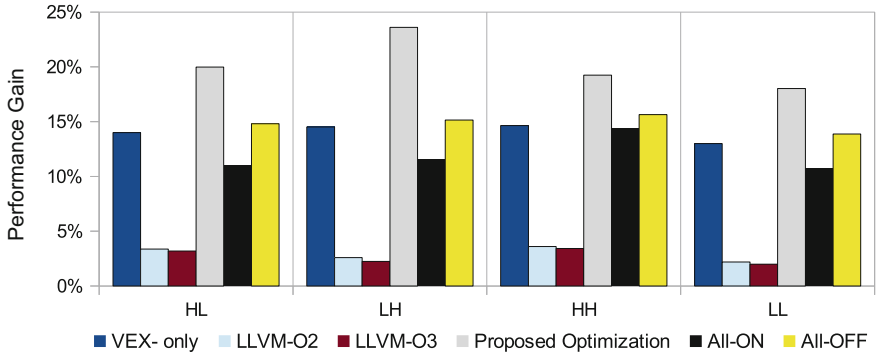


Fig. 2.6 The gained speed-up we gained comparing to the default LLVM-O1 optimization level in GSM benchmark

2.4 Conclusions and Future Work

This chapter presented a methodology for a compiler/architecture co-exploration of VLIW platform design. It 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 design frameworks focusing on performance optimization through iterative compilation and architecture specialization.

References

1. Fisher JA, Faraboschi P, Young C (2009) VLIW processors: once blue sky, now commonplace. IEEE Solid-State Circuits Mag 1(2):10–17
2. Fisher JA, Faraboschi P, Young C (2004) Embedded computing: a VLIW approach to architecture, compilers and tools. Morgan Kaufmann, Burlington, MA
3. Ascia G, Catania V, Palesi M, Patti D (2005) A system-level framework for evaluating area/performance/power trade-offs of vliw-based embedded systems. Design automation conference. In: Proceedings of the ASP-DAC 2005. Asia and South Pacific, vol 2., pp 940–943
4. Fisher JA (1981) Trace scheduling: a technique for global microcode compaction. IEEE Trans Comput 30(7):478–490
5. Hwu WMW, Mahlke SA, Chen WY, Chang PP, Warter NJ, Bringmann RA, Ouellette RG, Hank RE, Kiyohara T, Haab GE et al (1993) The superblock: an effective technique for VLIW and superscalar compilation. J Supercomput 7(1–2):229–248

6. Quinlan D (2000) Rose: compiler support for object-oriented frameworks. Parallel Process Lett 10:215–226
7. Fenacci D, Franke B, Thomson J (2010) 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, p 5. ACM
8. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76
9. The LLVM website (2013). http://www.llvm.org/
10. Faraboschi P, Homewood F (2000) ST200: a VLIW architecture for media-oriented applications. In: Microprocessor Forum. San Jose, CA
11. Saptono D, Brost V, Yang F, Prasetyo E (2008) 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, pp 416–421, Washington, DC, USA. IEEE Computer Society
12. Wong S, Van As T, Brown G (2008) ρ-vex: a reconfigurable and extensible softcore VLIW processor. In: International conference on ICECE Technology. FPT 2008, pp 369–372. IEEE
13. Hewlett-packard laboratories. vex toolchain. [online], available. http://www.hpl.hp.com/downloads/vex/
14. Multicube explorer. http://m3explorer.sourceforge.net/
15. Zaccaria V, Palermo G, Castro F, Silvano C, Mariani G (2010) Multicube explorer: an open source framework for design space exploration of chip multi-processors. In: 23rd International conference on architecture of computing systems (ARCS), pp 1–7. VDE
16. R Core Team et al. (2013) R: a language and environment for statistical computing. Vienna, Austria
17. Palermo G, Silvano C, Valsecchi S, Zaccaria V (2003) A system-level methodology for fast multi-objective design space exploration. In: Proceedings of the 13th ACM Great Lakes symposium on VLSI, pp 92–95. ACM
18. Kanungo T, Mount DM, Netanyahu NS, Piatko CD, Silverman R, Wu AY (2002) An efficient k-means clustering algorithm: analysis and implementation. IEEE Trans Pattern Anal Mach Intell 24(7):881–892
19. Li S, Ahn JH, Strong RD, Brockman JB, Tullsen DM, Jouppi NP (2009) McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In: International symposium on microarchitecture. MICRO-42. 42nd Annual IEEE/ACM, pp 469–480. IEEE
20. Roy RK (2001) Design of experiments using the Taguchi approach: 16 steps to product and process improvement. Wiley, Hoboken
21. Breslow N (1970) A generalized Kruskal-Wallis test for comparing k samples subject to unequal patterns of censorship. Biometrika 57(3):579–594
22. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MF, Thomson J, Toussaint M, Williams CK (2006) Using machine learning to focus iterative optimization. In: Proceedings of the international symposium on code generation and optimization. IEEE Computer Society, pp 295–305
23. Cavazos J, Dubach C, Agakov F (2006) 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, pp 24–34
24. Dubach C, Cavazos J, Franke B (2007) Fast compiler optimisation evaluation using code-feature based performance prediction. In: Proceedings of the 4th international conference on computing frontiers, pp 131–142
25. Thompson B (2002) Statistical, practical, and clinical: how many kinds of significance do counselors need to consider? J Couns Dev 80(1):64–71

Chapter 3
Selecting the Best Compiler Optimizations: A Bayesian Network Approach

Abstract After presenting our DSE approach for finding good compiler optimizations, we present our autotuning framework to tackle the problem of selecting the best compiler passes. It leverages machine learning and an application characterization to find the most promising optimization passes given an application. This chapter proposes *COBAYN*: Compiler autotuning framework using Bayesian Networks. An autotuning methodology based on 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 static analysis and hybrid feature collection approaches. Besides, we compare our approach against several state-of-the-art machine-learning models. Experiments are 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.

3.1 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 performance metrics. Depending on the strategy, these performance metrics could

A. H. Ashouri et al., *Automatic Tuning of Compilers Using Machine Learning*,
PoliMI SpringerBriefs, https://doi.org/10.1007/978-3-319-71489-9_3

be *execution time, code size* or *power consumption*. A holistic exploration approach to trade-off these metrics also represents a challenging problem [1].

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 [2]. 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 an application-specific embedded system domain; constructing an optimized compiler optimization heuristic 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* [3] and (ii) *machine-learning predictive modeling* [4–6]. The former approach relies on several recompilation phases and then selecting the best set of optimizations. Despite being effective, this approach has high overhead as it needs to be evaluated iteratively over all the enumerations of an optimization space. 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 normally require fewer compilation try-outs, however, their downside is the predicted optimizations lead to an execution binary that performs 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. In contrast, our proposed methodology starts by applying a statistical methodology to infer the probability distribution of the compiler optimizations to be enabled. Then, we perform an 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, here are our contributions:

- The introduction of a BN capable of capturing the correlation between the application features and the compiler optimizations. BN allows 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 against several state-of-the-art machine learning algorithms on 39 different benchmark applications.

The remainder of the chapter is organized as follows. Section 3.2 presents a quick review of the recent related literature. Readers are referred to Chap. 1 for a more comprehensive review on the state-of-the-art. Section 3.3 presents how the BN model can infer the probability of the distribution. Section 3.3.1 presents different techniques for collecting program features. Section 3.4 elaborates on the proposed framework. Sections 3.4.4 and 3.4.5 will introduce the results obtained on the application suites selected. Finally, Sect. 3.4.6 presents the comparison of the proposed methodology with state-of-the-art models.

3.2 Previous Work

Optimizations carried out at compilation have been broadly used, mainly in embedded computing applications. We refer to the holistic review on the survey in the Chap. 1. However, for the chapter completeness, we would like to remind a few related works. There are two major classes of optimization in the field of compilation research: (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* [7] and (*b*) *machine-learning* based approaches [8, 9]. Nonetheless, these two approaches have also been combined in many ways [4] that they cannot be distinguished easily.

Iterative compilation techniques were introduced as a mean to outperforming static handcrafted optimization sequences, those usually exposed by compiler interfaces as *optimization levels*. Since its introduction [7, 10], the goal of iterative compilation has been to identify the most appropriate compiler passes for a target application.

Other authors exploit iterative compilation jointly with architectural design space exploration for VLIW architectures [11]. 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 and the effects of different compiler optimizations are identified at the early design stages.

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 [4, 6, 8, 12, 13].

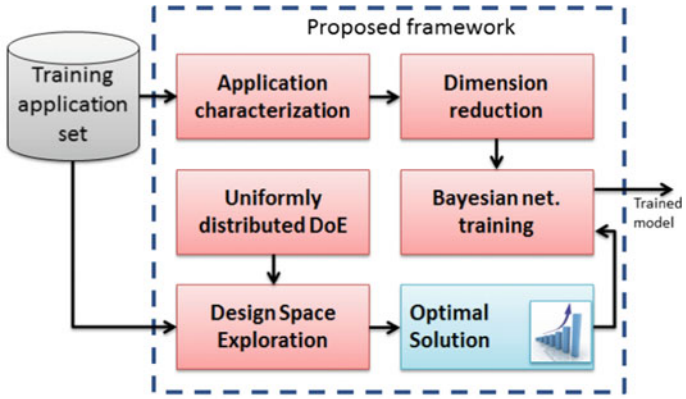
More recent literature on using different program features with machine learning have been proposed by [14] where the authors collected *IR (intermediate representation)* of the kernels and utilized *graph-kernels* to derived the similarities between those fetched *IRs*. Our approach is significantly 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.

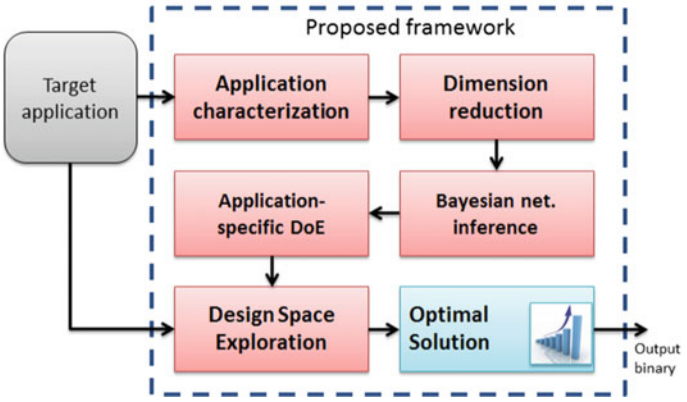
Additionally, in our approach, program features are dynamic and obtained through micro-architecture-independent characterization [15] and compared with the results using the static profiling [16]. The adoption of dynamic profiling provides insight into the actual program execution with the purpose of 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).

3.3 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



(a) Training the *Bayesian network*



(b) Optimization process for a new target application

Fig. 3.1 Overview of the proposed methodology

under analysis in terms of its main features. With little information loss, these features are pre-processed by means of statistical *dimension reduction* techniques to identify a more compact representation. A statistical model based on *BN* correlates these reduced representations to the compiler optimizations to maximize application performance.

The optimization flow is shown in Fig. 3.1 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 Fig. 5.1a). 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 Fig. 3.3).

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 Fig. 5.1a) or an *application-specific distribution* inferred through the BN (during the *exploitation phase* as in Fig. 5.1b).

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

3.3.1 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* [15], (ii) *static analysis* using *MilePost* [16] 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 [15]. It is a plugin for the Linux-PIN tool [17] 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 [16, 18] 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.

3.3.2 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 [5], PCA was used. In this chapter, 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. For a quantitative comparison the readers can refer to Sect. 3.4.4, Table 3.5.

Let γ 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 γ. The application profiles can be organized in a matrix P with m rows and l columns. Each vector γ (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 γ can be well represented with a vector α of smaller size.

Both PCA and EFA are statistical techniques aimed at identifying a way to represent γ with a shorter vector α while minimizing the information loss. Nevertheless, they rely on different concepts for organizing this reduction [19, 20]. 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 represent-

ing the hidden process behind the feature generation. In both techniques, the output columns cannot be called by their header name and are not directly observable.

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

In EFA, the elements in the vector of reduced size α are meant to explain the structure underlying the variables γ, while α, represents a vector of *latent variables* that cannot be directly observed. The variables γ are expected to be a linear combination of the variables in α. In EFA, this relationship explains the correlation between the different variables in γ; that is, correlated variables in γ are likely to depend on the same hidden variable in α. The relationship between the latent α 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 α tends to be a mixture of all variables in γ. Therefore, it is hard to tell what a component represents. When adopting EFA instead, the components α tend to depend on a smaller set of elements in γ that are correlated with each other. That is, when applying EFA, α is a compressed representation of γ, where elements in γ that are correlated (i.e. that carry the same information) are compressed into a reduced number of elements in α. Note that reducing the profile size by means of EFA results in a α that better describes the type of application under analysis in reference to PCA [21].

Consequently, having obtained γ 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 is that P needs to be transformed, as to obtain the final γ in *positive-definite covariance* form [22]. 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 [23, 24] or use packages in *R* statistical tool [25] i.e., *nearPD* to compute nearest positive definite matrix.

3.3.3 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 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 chapter, the *phase ordering* problem [26] is not taken into account. We rather consider how different optimizations o_i are organized in a predefined order embedded in the compiler. A compiler optimization sequence represented by the vector 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 α of the k reduced components computed either via PCA or EFA from its software features. Elements α_i in vector α generally belong to the continuous domain.

The optimal compiler optimization sequence $\bar{o} \in \mathcal{O}$ that maximizes the performance of an application is 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 in the optimization sequence. 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 $\bar{o}$ is unknown a priory is because it is not possible to capture, in a deterministic way, the dependencies among the variables in the vectors $\bar{o}$ and α. There is no way to identify an analytic model to exactly fit the vector function $\bar{o}(\alpha)$. As a matter of fact, the best optimization sequence $\bar{o}$ 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 $\bar{o}$ 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:

- Their expressiveness allows one to include heterogeneous variables in the same framework such as Boolean variables (in the optimization vector o) and continuous variables (in the application characterization α).
- 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 o) given the *evidence* on other variables (the application characterization α).

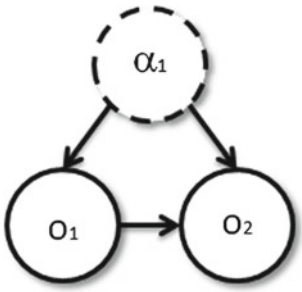


Fig. 3.2 A *Bayesian Network* example

This enables us to infer an *application-specific distribution* for the vector o from the vector α 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 3.2 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 [27]. 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 α, thus reducing the number of nodes in the graph.

For efficiency reasons, the algorithm used for selecting the graph topology is a heuristic algorithm, named *K2*, initialized with the *Maximum Weight Spanning Tree* (MWST) ordering method as suggested in the Matlab toolbox in use [27]. The initial ordering of the nodes for the MWST algorithm is given to let the elements α to appear first and then the elements of o. Even if the final topological sorting of the nodes changes according to the algorithm described in [28], by using this initialization criterion, it always happens that the dependencies are directed from elements of α to elements of 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 a 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 o [27]. This is a mathematical necessity to map in the *Bayesian framework* the dependencies of Boolean variables in o with respect to continuous variables in α. 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 [27]. Training data are gathered by analyzing a set A of training applications (Fig. 5.1a). 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 α. Then, an experimental compilation campaign is carried out for each application by sampling several compiler optimization sequences from the compiler optimization space $\mathscr{O}$ 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 α characterizing the application.

Inferring an application-specific distribution. Once the *Bayesian Network* has been trained, the principal component vector α obtained for a new application can be fed as evidence to the framework to bias the distribution of the compiler optimization vector 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 α 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 $P(o_i = 0 \mid \mathscr{P})$ and $P(o_i = 1 \mid \mathscr{P})$ can be computed on the basis of the parent node vector value $\mathscr{P}$ (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 finding the most probable optimization sequence as the first sample.

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

3.4 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 [29] with *ArchLinux* and *GCC-ARM 4.6.3*.

3.4.1 Benchmark Suites

To assess the proposed methodology, we have used two major benchmark suites: (i) *cBench* [30], and (ii) *PolyBench* [31, 32]. 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 3.1 for the list of applications selected in the two benchmark suites.

3.4.1.1 cBench

The *cBench* suite [30] 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.

3.4.1.2 PolyBench

The *PolyBench* benchmark suite [31, 32] 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.

Table 3.1 Benchmark suites used in this work

cBench list	Description
(a) *cBench* applications selected for this work	
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

(continued)

Table 3.1 (continued)

PolyBench list	Description
(b) Linear-algebra/applications of the *PolyBench* suite selected for this work	
2 mm	2 Matrix multiplications (D = A×B; E = C×D)
3 mm	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 = αA×B + βC
gemver	Vector multiplication and matrix addition
gesummv	Scalar, vector and matrix multiplication
mvt	Matrix vector product and transpose
symm	Symmetric matrix-multiply
syr2k	Symmetric rank—2k operations
syrk	Symmetric rank—k operations
trisolv	Triangular solver
trmm	Triangular matrix-multiply

3.4.2 Compiler Transformations

The compiler transformations analyzed have been reported in Table 3.2. We based our design space on the work of [3]. 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 or disabled using their 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 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.

Table 3.2 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 trade-off

3.4.3 Bayesian Network Results

In this work, Matlab environment [27, 33] 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 5.2. As stated in Sect. 3.3.2, one of the features of using EFA is that the factors are linear combinations that maximize the shared portion of the variance. Therefore, as a prerequisite, the covariance matrix should be positive definite. The 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. The decision on the numbers of factors has been derived from the *Kaiser* test [34]. The 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 3.3 reports the factors derived for each benchmark and characterization method.

Table 3.3 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.

Table 3.3 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]

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 (*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 Fig. 3.3. 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 Fig. 3.3 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

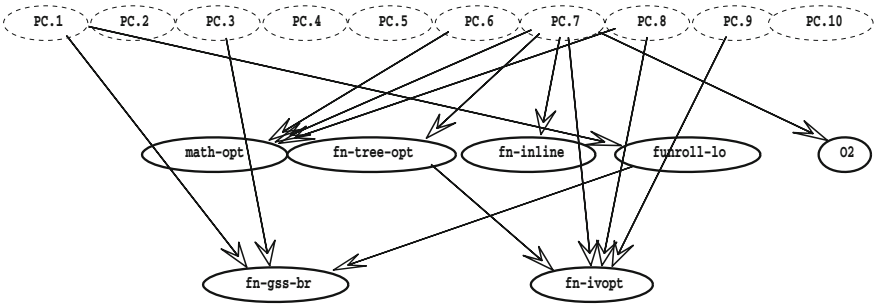


Fig. 3.3 Topology of the *Bayesian Network* if *security_rijndael_e* is left out of the training set

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 Fig. 3.3. 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 3.4 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, depends not only on the number but also on the applications and data-set used for the training. The same applies to the time needed for compiling and executing the target application during the online compiler autotuning phase. To this end, Table 3.4 reports the numbers for each specific phase considering the Cbench as training set and Susan as the 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 configurations and the feature collection phase. The time needed to post-process the data and to generate the Bayesian Network model is around 70 s.

During the online phase (inference phase), the time needed for extracting the software features from the target application is 14.4 s while querying BN is less than 1 s. The compilation and execution-time on the target platform for Susan are 4.5

Table 3.4 COBAYN timing breakdown for offline training and online inference for *Susan* application

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

and 8.9 s, 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.

3.4.4 Comparison Results

It is well known that *Random Iterative Compilation* (RIC) can improve application performance compared with static handcrafted compiler optimization sequences [4]. 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 [3–5, 12]. Accordingly, to evaluate the proposed approach, we compared our results with (1) standard optimization levels -O2 and -O3 (2) the Random Iterative Compilation (RIC) methodology that samples compiler optimization sequences from the uniform distribution, and (3) two state-of-the-art methodologies coupling machine learning with an iterative methodology, and a optimization speedup predictor methodology, respectively.

The proposed methodology samples different compiler optimization sequences from the BN. The performance achieved by the best application binary depends on the number of sequences sampled from the model. In this section, the results 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 [5].

3.4.4.1 Bayesian Networks Performance Evaluation

Table 3.5 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 Sect. 3.3.2 with respect to PCA in [5]. 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.

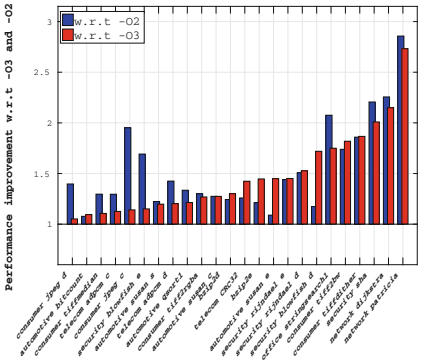
Table 3.5 COBAYN (BN using EFA) speedup w.r.t standard optimization levels (-O2 and -O3) and *Random Iterative Compilation* (RIC) and our previous approach of BN in [5] using PCA

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

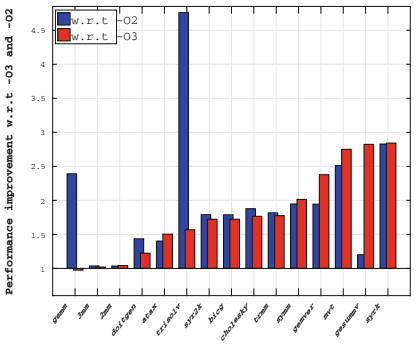
Using two major benchmarks and three different application characterization techniques, we report six different plots showcasing the benefits of the proposed methodology with respect to the GCC standard optimization levels. Figure 3.4, reports the speedups by considering a sample of eight different compiler 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 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 3.5 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.

3.4.4.2 COBAYN's Portability Analysis

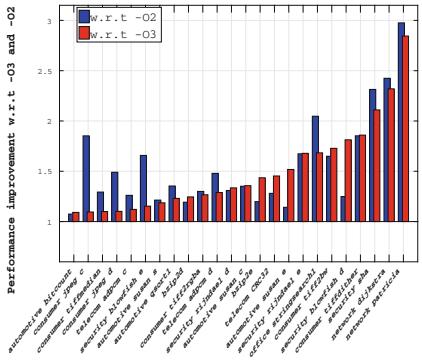
The results reported in this section are computed by means of LOO cross-validation on the two individual benchmark suites separately, one with 24 and the other with 15 applications. As the nature of these two benchmark suites is totally different, we believed it would be unfair to train on one and test on the other, so we analyzed the feasibility of mixing these applications in a fair heterogeneous set of *BigSet* so



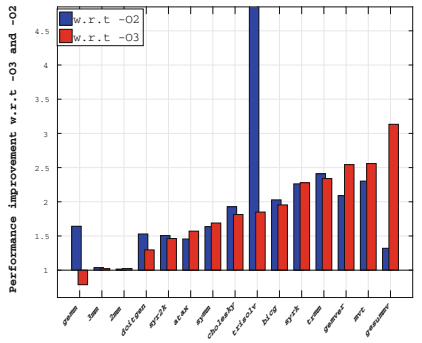
(a) BN with dynamic features on cBench



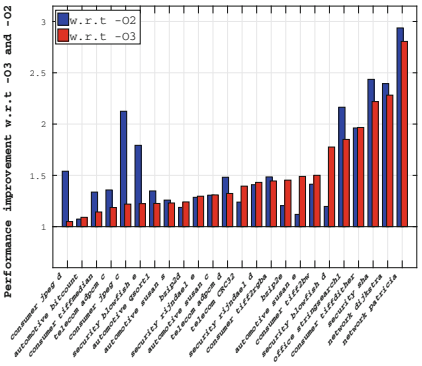
(b) BN with dynamic features on PolyBench



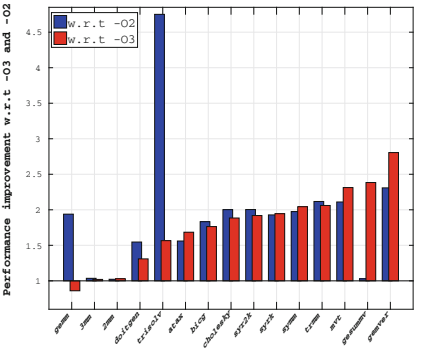
(c) BN with static features on cBench



(d) BN with static features on PolyBench



(e) BN with dynamic+static features on cBench



(f) BN with dynamic+static features on PolyBench

Fig. 3.4 Performance speedup w.r.t -O2 and -O3

Table 3.6 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. COBAYN's default
Cbench	PolyBench	
24 (All)	15 (All)	1.1143
15	15	1.0743
10	10	1.0432
5	5	0.9896

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 3.6 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 3.6 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).

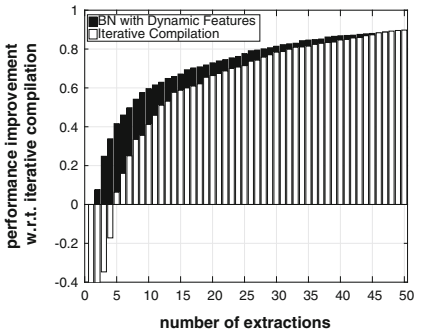
3.4.4.3 Performance Improvement

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

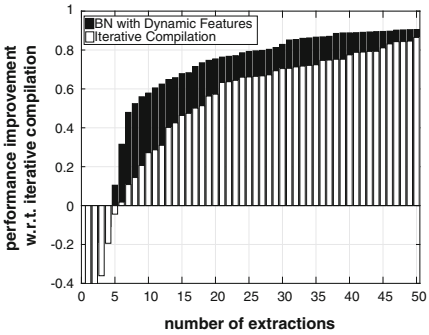
$$NPI = \frac{E_{ref} - E}{E_{ref} - E_{best}} \tag{3.1}$$

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.

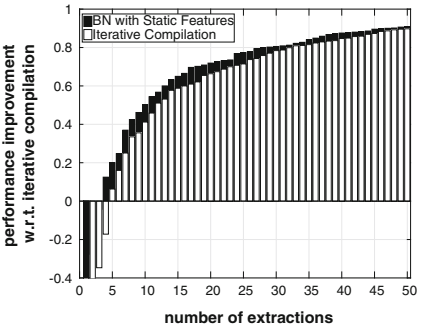
Figure 3.5 reports for six different benchmark/feature selection methods. The NPI achieved by the *proposed optimization* technique and by the RIC technique in refer-



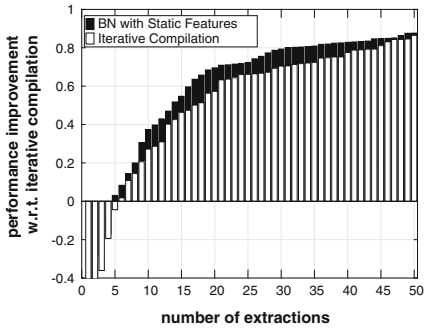
(a) BN with dynamic features on cBench



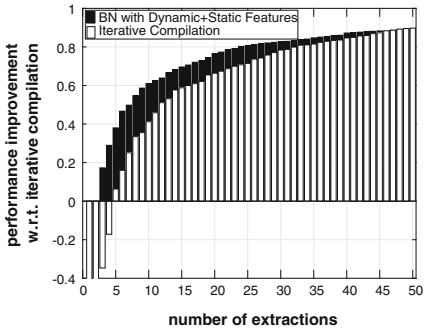
(b) BN with dynamic features on PolyBench



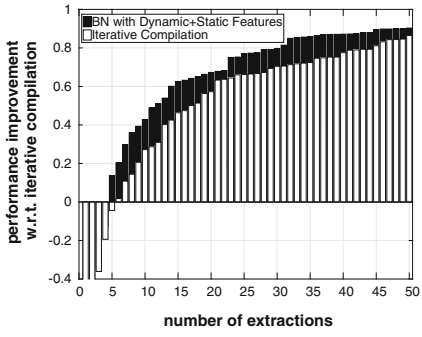
(c) BN with static features on cBench



(d) BN with static features on PolyBench



(e) BN with hybrid features on cBench



(f) BN with hybrid features on PolyBench

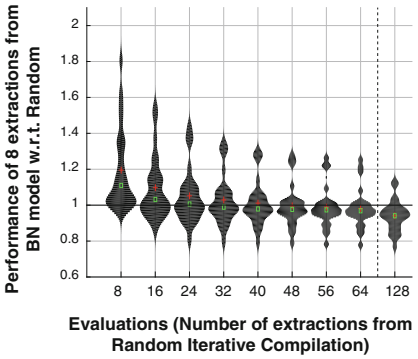
Fig. 3.5 Normalized performance improvement (NPI) w.r.t. RIC model

ence 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 Fig. 3.5 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.

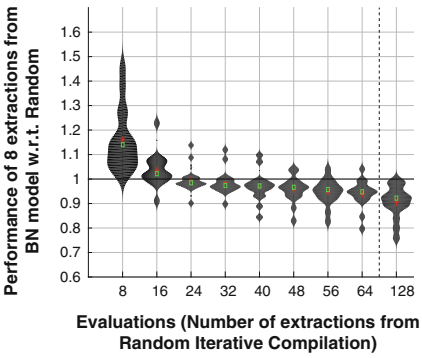
3.4.5 A Practical Usage Assessment

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 Fig. 3.6, 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 Fig. 3.6 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× 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.

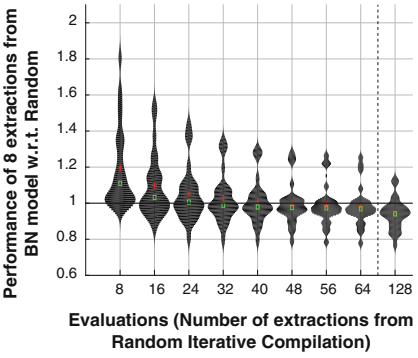
Accordingly, by increasing the compilation efforts on *RIC*, while keeping the exploration efforts of the proposed approach constant, the application speedup of COBAYN decreases. On average, RIC needs 24-32 extractions to achieve the application performance obtained with eight extractions by COBAYN. This means that COBAYN provides a speedup of 3-4× in terms of optimization efforts, that is only slightly impacted by the initial overhead (less than 2 evaluations) reported in Sect. 3.4.3. 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 by a vertical dashed-line.



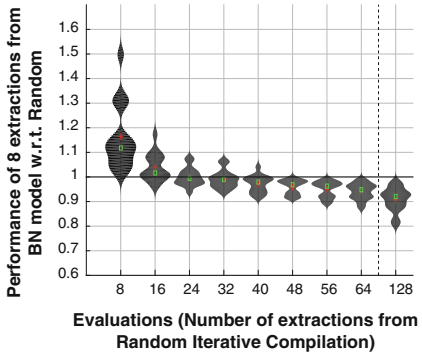
(a) BN with dynamic features on cBench



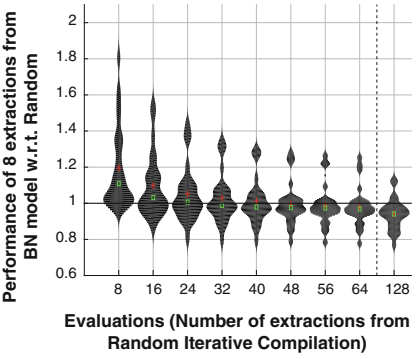
(b) BN with dynamic features on PolyBench



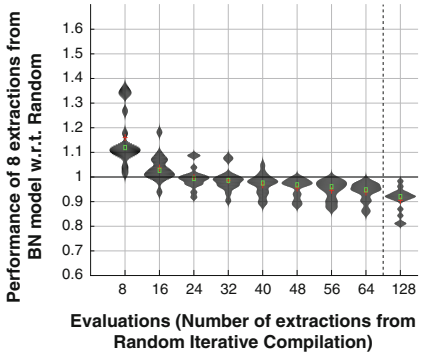
(c) BN with static features on cBench



(d) BN with static features on PolyBench



(e) BN with dynamic+static features on cBench



(f) BN with dynamic+static features on PolyBench

Fig. 3.6 Exploration speedup of 8 extractions w.r.t different evaluations of RIC

3.4.6 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.

3.4.6.1 Comparison to a Iterative Compilation Methodology

Agakov et al. [4] leveraged machine-learning models to focus on the exploration of the compiler optimization. Their methodology exploits a Markov chain oracle and an independent identically distributed (IID) probability distribution oracle. These two models learned offline bias certain optimizations over others and replace the uniform probability distribution we applied earlier for the RIC reference methodology. Their work 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 Sects. 4.2 and 4.3 of [4] as:

$$P(S_{IID}) = s_1, s_1, ..., s_L = \prod_{i=1}^{L} P(s_i) \tag{3.2}$$

$$P(S_{Markov}) = P(s_1) \prod_{i=2}^{L} P(s_i|s_{i-1}) \tag{3.3}$$

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 [4], 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 Fig. 3.7. 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 (Eq. 5.5). COBAYN is able to capture a more realistic probability matrix of the compiler optimization problem and achieves with faster convergence

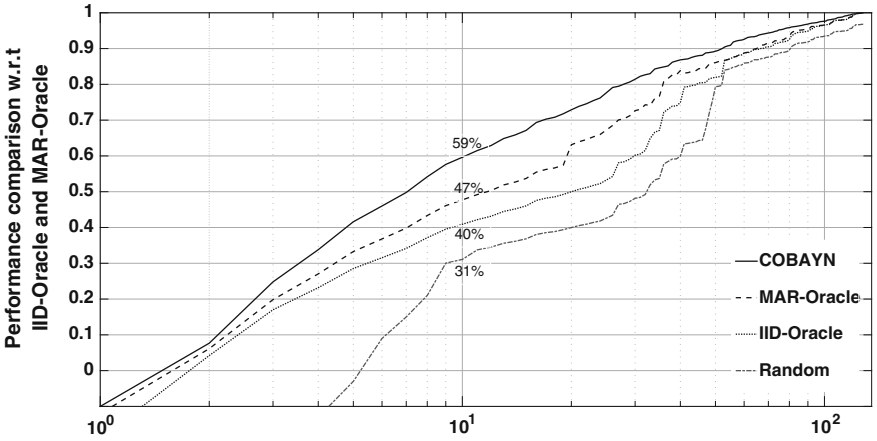


Fig. 3.7 NRI-scales speedup comparison of COBAYN with IID-oracle and MAR-oracle reported in [4]

towards the optimal result. It brings 1.25× and 1.47× speedup with respect to IID and the Markov oracle, respectively.

3.4.6.2 Comparison to a Non-iterative Compilation Methodology

Park et al. [35] 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* [36] 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. [35].

Nonetheless, we compare their methodology by applying it to the problem at hand. Table 3.7 reports the results obtained by using the machine-learning models in [35] 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.

Table 3.7 COBAYN speedup w.r.t. the average speedup gained with predictive modeling in both 1-shot and 8-shot scenarios reported in [35]

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

(continued)

Table 3.7 (continued)

Algorithm and parameter configuration	COBAYN Speedup	
	w.r.t 1-shot	w.r.t 8-shot
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

Note that Harmonic mean is always less than or equal to the arithmetic mean [37]. 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 [35].

3.5 Conclusions

This chapter presented 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 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.

In the following chapter, we move towards tackling the phase-ordering problem of compiler optimizations using machine learning.

References

1. Palermo G, Silvano C, Zaccaria V (2005) Multi-objective design space exploration of embedded systems. J Embed Comput 1(3):305–316
2. Hoste K, Eeckhout L (2008) Cole: compiler optimization level exploration. In: Proceedings of the 6th annual IEEE/ACM international symposium on code generation and optimization, pp 165–174
3. Chen Y, Fang S, Huang Y, Eeckhout L, Fursin G, Temam O, Chengyong W (2012) Deconstructing iterative optimization. ACM Trans Archit Code Optim (TACO) 9(3):21

4. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI (2006) Using machine learning to focus iterative optimization. In: Proceedings of the international symposium on code generation and optimization. IEEE Computer Society, pp 295–305
5. Ashouri AH, Mariani G, Palermo G, Silvano C (2014) A Bayesian network approach for compiler auto-tuning for embedded processors. In: 2014 IEEE 12th symposium on embedded systems for real-time multimedia, ESTIMedia 2014, pp 90–97. http://doi.acm.org/10.1109/ESTIMedia.2014.6962349
6. Ashouri AH, Mariani G, Palermo G, Park E, Cavazos J, Silvano C (2016) Cobayn: compiler autotuning framework using bayesian networks. ACM Trans Archit Code Optim (TACO), 13(2):21:1–21:25. http://doi.acm.org/10.1145/2928270
7. Bodin F, Kisuki T, Knijnenburg P, O'Boyle M, Rohou E (1998) Iterative compilation in a non-linear optimisation space. In: Workshop on profile and feedback-directed compilation
8. Cooper KD, Schielke PJ, Subramanian D (1999) Optimizing for reduced code space using genetic algorithms. In: ACM SIGPLAN Notices
9. Kisuki T, Knijnenburg PMW, O'Boyle MFP (2000) 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, 15–19 October 2000, pp 237–248
10. Kisuki T, Knijnenburg PMW, O'Boyle MFP, Bodin F, Wijshoff HAG (1999) A feasibility study in iterative compilation. In: High performance computing. Springer, pp 121–132
11. Ashouri AH, Zaccaria V, Xydis S, Palermo G, Silvano C (2013) A framework for compiler Level statistical analysis over customized VLIW architecture. In: VLSI-SoC, pp 124–129. http://dx.doi.org/10.1109/VLSI-SoC.2013.6673262
12. Cavazos J, Fursin G, Agakov F (2007) Rapidly selecting good compiler optimizations using performance counters. In: International symposium on code generation and optimization (CGO'07)
13. Stephenson M, Amarasinghe S (2003) Meta optimization: improving compiler heuristics with machine learning. ACM SIGPLAN Notices 38:77–90
14. Park E, Cavazos J, Alvarez MA (2012) Using graph-based program characterization for predictive modeling. In: Proceedings of the international symposium on code generation and optimization, pp 295–305
15. Hoste K, Eeckhout L (2007) Microarchitecture-independent workload characterization. IEEE Micro 27(3):63–72
16. Fursin G, Miranda C, Temam O (2008) MILEPOST GCC: machine learning based research compiler. In: GCC Summit
17. Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) 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, pp 190–200, New York, NY, USA. ACM
18. Fursin G, Kashnikov Y, Memon AW (2011) Milepost gcc: machine learning enabled self-tuning compiler. Int J Parallel Prog 39(3):296–327
19. Gorsuch RL (1988) Exploratory factor analysis. In: Handbook of multivariate experimental psychology. Springer, pp 231–258
20. Thompson B (2002) Statistical, practical, and clinical: how many kinds of significance do counselors need to consider? J Couns Dev 80(1):64–71
21. Jin Z, Cheng AC (2008) Improve simulation efficiency using statistical benchmark subsetting—an implantbench case study. In: 45th ACM/IEEE on design automation conference, DAC 2008, pp 970–973
22. Bhatia R (2009) Positive definite matrices. Princeton University Press, Princeton
23. Lee J, Mathews VJ (1994) A stability condition for certain bilinear systems. IEEE Trans Signal Process 42(7):1871–1873
24. Tanaka M, Nakata K (2014) Positive definite matrix approximation with condition number constraint. Optim Lett 8(3):939–947

25. R Core Team et al (2013) R: A language and environment for statistical computing. Austria, Vienna
26. Kulkarni PA, Whalley DB, Tyson GS, Davidson JW (2009) Practical exhaustive optimization phase order exploration and evaluation. ACM Trans Archit Code Optim 6(1):1:1–1:36
27. Murphy KP (2001) The bayes net toolbox for matlab. Comput Sci Stat 33:2001
28. Heckerman D, Chickering DM (1995) Learning bayesian networks: the combination of knowledge and statistical data. Mach Learn 20:197–243
29. Texas Instruments (2012) Pandaboard. OMAP4430 SoC dev. board, revision A, 2
30. Fursin G (2010) 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
31. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to gpu codes. In: Innovative parallel computing (InPar), pp 1–10. IEEE
32. Pouchet L-N (2012) Polybench: the polyhedral benchmark suite. http://www.cs.ucla.edu/~pouchet/software/polybench/[cited July,]
33. Roberto S, Concha B, Pedro L, Lozano JA, Echegoyen C, Mendiburu A, Armananzas R, Shakya S (2010) Mateda-2.0: estimation of distribution algorithms in matlab. J Stat Softw 35(7):1–30
34. Kaiser HF (1958) The varimax criterion for analytic rotation in factor analysis. Psychometrika 23(3):187–200
35. Park E, Cavazos J, Pouchet LN (2013) Predictive modeling in a polyhedral optimization space. Int J Parallel Prog 41:704–750
36. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. ACM SIGKDD Explorations Newsletter 11(1):10–18
37. Hoefler T, Belli R (2015) 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, p 73. ACM

Chapter 4
The Phase-Ordering Problem: An Intermediate Speedup Prediction Approach

Abstract This chapter presents the first of two methods to tackle the phase-ordering problem of compiler optimizations. Here, we present an intermediate speedup prediction approach followed by a full-sequence prediction approach in the next chapter and we show pros and cons of each approach in detail. 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 to 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.

4.1 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-hard. 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 com-

A. H. Ashouri et al., *Automatic Tuning of Compilers Using Machine Learning*,
PoliMI SpringerBriefs, https://doi.org/10.1007/978-3-319-71489-9_4

piler 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 because embedded systems are usually designed with tight extra-functional properties constraints. Second, the large variety of embedded platforms cannot 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 a peak performance of several 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 [25]. *Polyhedral compilation* is a recent attempt to bring mathematical representation focusing on the loop-nest of the *polyhedral model* including many different tools [4, 5, 16, 20].

In this chapter, we tackle the phase-ordering problem by using predictive modeling. Our predictive model can predict the next-best compiler optimization to apply given the current status of the application under analysis. 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. We call this approach an intermediate speedup prediction as it finds the best available optimization to apply at each intermediate state of an application. The proposed predictive model has been trained off-line with different permutations of the compiler flags (allowing repetitions and dynamic sequence length). Thus, 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 to be applied given the current status of the application under optimization.

We apply prediction modeling techniques originally proposed in [9] 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 4.2 provides a brief discussion on the related work. In Sect. 4.3, we introduce the predictive modeling approach to tackle *phase-ordering*. Section 4.4 presents experimental evaluation of the proposed methodology on an Unix-based Intel platform. Finally, Sect. 4.5 summarizes the outcome of the work and some future paths.

4.2 Related Work

We have extensively presented the literature in the Chap. 1. However, for the sake of completeness, we reiterate on a few important related work 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* [1, 3, 7, 8] and (ii) *applying machine learning to compilation* [2, 24]. 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 [19] 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 [12]. Authors in [18] exhaustively exploring the ordering space at functions' granularity level and evaluate their methodology with search tree algorithms and in [22] the authors have exploited iterative compilation with the information relying on relative passes in previously generated compiler options in the sequence in function level.

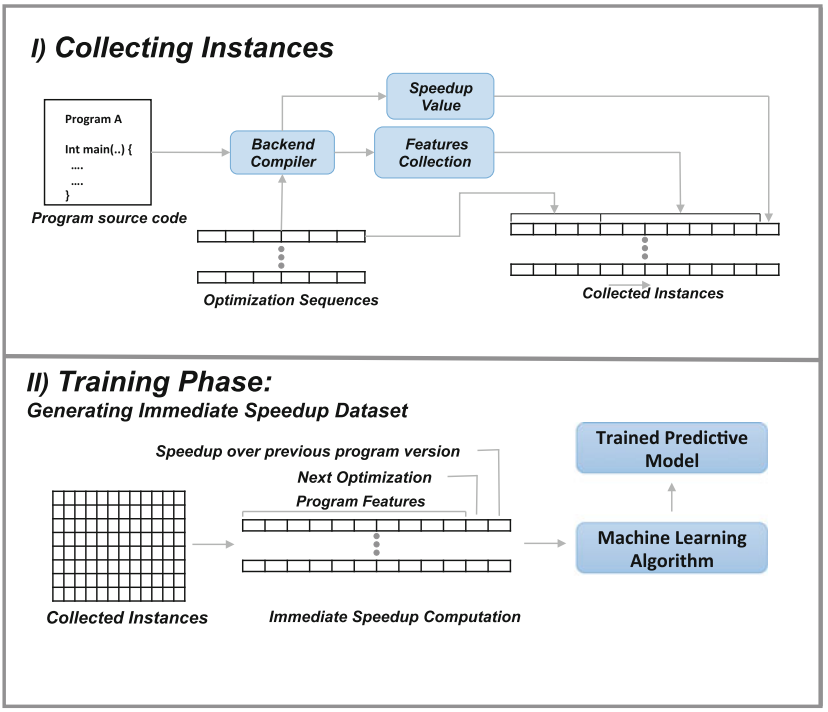
Our approach is rather different with respect to the literature, given that we propose a predictive modeling methodology utilizing an independent micro-architecture characterization features for all different *permutations with repetitions* of the compiler options and come up with the prediction of the *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.

4.3 The Proposed Methodology

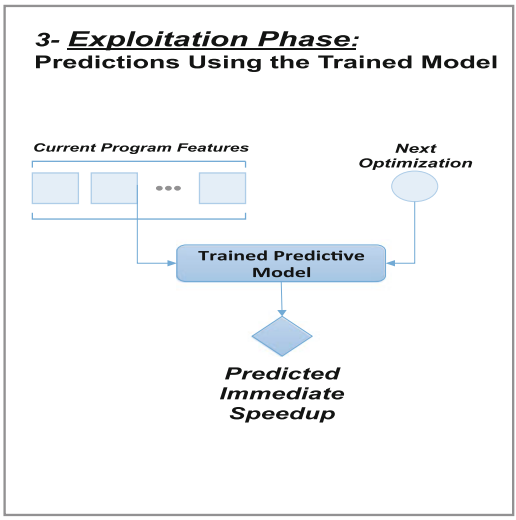
Main goal of the proposed methodology is to identify the feasibility of tackling the phase ordering problem using a predictive modeling methodology. Each application optimized with a unique compiler options sequence is passed through a characterization phase, that generates a parametric representation of its dynamic features. A model based on predictive modeling correlates these features to the compiler optimizations applied such as to predict the application speedup by using the next-best compiler optimization at each level.

The optimization flow is represented in Fig. 4.1. It consists of three main phases: (1) *Data collection* where different instances of the application are executed and an application characterizations is collected with the speedup achieved by utilizing the specific compiler pass, (2) *Training phase* where a predictive modeling is learned on the base of a set of training applications, and (3) *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 intermediate speedups associated with 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.



(a) Data Collection and Training



(b) Testing Prediction Model

Fig. 4.1 Proposed framework: offline-training phase which is done once, and online-prediction phase for optimizing new unseen applications

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 Fig. 4.1) 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 Sects. 4.3.3.1 and 4.3.3.2 respectively.

4.3.1 Compiler Phase-Ordering Problem

We have formulated the optimization space of the phase-ordering problem in Chap. 1 in details. Here is a quick recap of what we discussed earlier in this book.

The *phase-ordering* problem, let us define a Boolean vector o whose elements o_i are the different compiler optimizations. A *Phase-ordering* compiler optimization sequence represented by the vector 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:

$$|\mathcal{O}phases\_repetition| = \sum_{i=0}^{m} n^i \tag{4.1}$$

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.

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 Sect. 4.4; Table 4.2. 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.

4.3.2 Application Characterization

For our predictive modeling presented in this chapter, again we exploit PIN [21] based *dynamic profiling* framework to analyze the behavior of the different applications at execution time. Refer to the previous chapters for more information on how MICA can characterize an application.

Upon collecting the features from an application, we use PCA. To reiterate, 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 [15]. 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 [14]. 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*.

4.3.3 Intermediate Speedup Prediction

We present our prediction approach in this section. We call this approach *intermediate* speedup prediction. 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 [9]. 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 ρ, whose elements α_i are the first k principal components of its dynamic profiling features. Elements α_i in the vector ρ generally belong to the continuous domain. The optimal compiler optimization sequence $\bar{o} \in \mathcal{O}$ 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 Sect. 4.4, 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.

4.3.3.1 DFS Search Heuristic

Depth-First Search (DFS) and its optimized version Depth-First Iterative Deepening [17] 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 o_o, Considering sequence o_i, for each of the possible optimizations we predict the immediate speedup δ_i derived from applying o_i after o_i in the compilation process. The *Intermediate speedup* is computed as:

$$e_i = Exec\_time(o_i)/Exec\_time(o_j) \tag{4.2}$$

where e_i is the ratio between the execution time of the program compiled using o_i and the execution time of the version of the program generated using o_j. We define o_j as 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 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 o_j, that is 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 o_i tested, the algorithm backtracks to the previously considered sequence o_k, that is, o_i without its last optimization. If a sequence o_i has reached the maximum optimization sequence length N to be considered, we stop applying further optimizations after it. We then evaluate 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 o_o.

4.3.3.2 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 selecting 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 Sect. 4.3.3.1; Eq. 4.3, the model we trained is able to predict only intermediate speedups δ (i.e. the speedup of o_j over o_i, where o_j is the optimization sequence obtained by applying o_i after o_i. We are able to predict the actual speedup of optimization sequence o by *Multiplying* the immediate speedups δ_i predicted at each optimization $o_i \in o$:

$$o_o : \prod_{O_i \in o} \delta(o_i) \tag{4.3}$$

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 o_o to the complete optimization space. In this way, we are able to predict the speedup of every optimization sequence $o \in \mathcal{O}$ and map our system to the classic exhaustive predictions methodology we described.

4.4 Experimental Evaluation

We assess the proposed methodology of the *intermediate speedup* 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 [10] consisting of six different applications has been integrated within the framework. The list of selected applications has been reported in Table 4.1. Table 4.2 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 thus being able to reach higher potential speedups. 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 4.3 represents the maximum achievable speedup enabling repetition feature on every individual application. 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 [13].

Table 4.1 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

Table 4.2 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		**30**
Unique number of optimizations		**13**

Table 4.3 Best found optimization passes by enabling repetition

Application	With repetition	Without repetition	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

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 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 [11] has been integrated into 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 Eq. 4.1, given the complexity of the problem, we assessed the feasibility of our proposed approach with four different

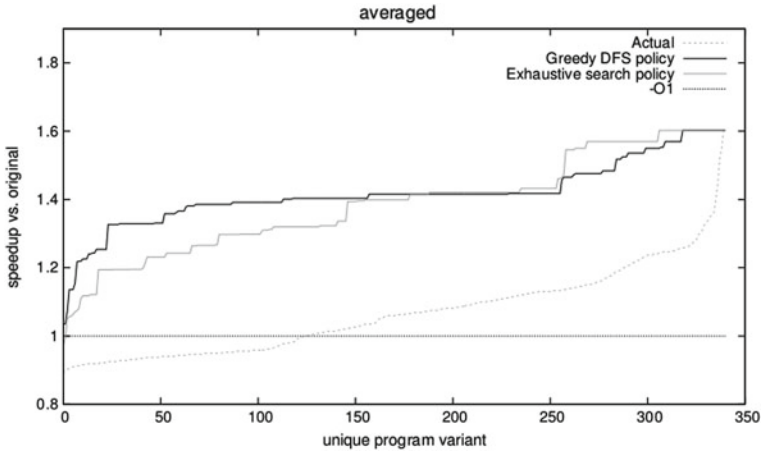


Fig. 4.2 Average Speedup of the proposed methodology across all applications

sequences of compiler options with a total of 30 compiler optimizations (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 Fig. 4.2. 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 Sect. 4.3.3. We define the *actual speedup* line as the speedup observed from running the application using the compiler optimization prediction of the machine learning model. Table 4.4 is presenting the quantitative values of utilizing the two predictive modeling algorithms within our framework. We evaluated our *iterative greedy approaches* with the classic 1-shot predictive speedup approach mentioned in [6, 9, 23] and the results reported in Table 4.4 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 Fig. 4.2 demonstrates that the performance of the greedy exploration policy in the 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.

Table 4.4 Average speedup of the one-shot prediction for both approaches w.r.t LLVM baseline

Application	Greedy DFS	Exhaustive search
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

4.5 Conclusions

This chapter presented a method based on predictive modeling to select the next-best compiler option (intermediate speedup prediction) 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.

In the next chapter, we provide another prediction approach namely, full-sequence prediction to tackle the phase-ordering problem of optimizations.

References

1. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI (2006) Using machine learning to focus iterative optimization. In: Proceedings of the international symposium on code generation and optimization, IEEE Computer Society, pp 295–305
2. Ashouri AH, Mariani G, Palermo G, Park E, Cavazos J, Silvano C (2016) COBAYN: compiler autotuning framework using bayesian networks. ACM Trans Archit Code Optim (TACO) 13(2):21:1–21:25. http://doi.acm.org/10.1145/2928270
3. Ashouri AH, Zaccaria V, Xydis S, Palermo G, Silvano C (2013) A framework for compiler level statistical analysis over customized VLIW architecture. In: VLSI-SoC, pp 124–129. http://dx.doi.org/10.1109/VLSI-SoC.2013.6673262
4. Benabderrahmane M-W, Pouchet L-N, Cohen A, Bastoul C (2010) The polyhedral model is more widely applicable than you think. Compiler constr, 6011, pp 283–303
5. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) 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), Citeseer, Tucson, AZ, June 2008
6. Cavazos J, Dubach C, Agakov F (2006) 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, pp 24–34
7. Cavazos J, Fursin G, Agakov F (2007) Rapidly selecting good compiler optimizations using performance counters. In: International symposium on code generation and optimization (CGO'07)

8. Chen Y, Fang S, Huang Y, Eeckhout L, Fursin G, Temam O, Chengyong W (2012) Deconstructing iterative optimization. ACM Trans Archit Code Optim (TACO) 9(3):21
9. Dubach C, Cavazos J, Franke B (2007) Fast compiler optimisation evaluation using code-feature based performance prediction. In: Proceedings of the 4th international conference on computing frontiers, pp 131–142
10. Fursin G (2010) 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
11. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA data mining software: an update. ACM SIGKDD Explor Newslett 11(1):10–18
12. Haneda M (2005) Optimizing general purpose compiler optimization. In: Proceedings of the 2nd conference on computing frontiers, pp 180–188
13. Hoefler T, Belli R (2015) 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, ACM, p 73
14. Hoste K, Eeckhout L (2007) Microarchitecture-independent workload characterization. IEEE Micro 27(3):63–72
15. Johnson RA, Wichern DW (2002) Applied multivariate statistical analysis, vol 5. Prentice Hall, Upper Saddle River, NJ
16. Karp RM, Miller RE, Winograd, S (1967) The organization of computations for uniform recurrence equations. J ACM (JACM) 14(3):563–590
17. Korf RE (1985) Depth-first iterative-deepening: An optimal admissible tree search. Artif Intell 27(1):97–109
18. Kulkarni PA, Whalley DB, Tyson GS, Davidson JW (2009) Practical exhaustive optimization phase order exploration and evaluation. ACM Trans Archit Code Optim 6(1):1:1–1:36
19. Kulkarni S, Cavazos J (2012) Mitigating the compiler optimization phase-ordering problem using machine learning. In: ACM SIGPLAN Notices
20. Loechner V (1999) PolyLib: a library for manipulating parameterized polyhedra. Technical report, ICPS, Universite Louis Pasteur de Strasbourg, France, Mar. 1999
21. Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) 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), ACM, New York, NY, USA, pp 190–200
22. Nobre N, Martins LGA, Cardoso JMP (2015) 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, ACM, pp 58–67
23. Park E, Cavazos J, Pouchet LN (2013) Int J Parallel Prog 41:704–750. https://doi.org/10.1007/s10766-013-0241-1
24. Park E, Kulkarni S, Cavazos J (2011) An evaluation of different modeling techniques for iterative compilation. In: Proceedings of the 14th international conference on compilers, architectures and synthesis for embedded systems, pp 65–74
25. Vuduc R, Demmel JW, Bilmes JA (2004) Statistical models for empirical search-based performance tuning. Int J High Perform Comput Appl 18(1):65–94

Chapter 5
The Phase-Ordering Problem: A Complete Sequence Prediction Approach

Abstract This chapter proposes our second approach to tackle the phase-ordering problem. We already showed our intermediate speedup prediction method in Chap. 4. Here, we present our full-sequence speedup prediction method called MiCOMP. MiCOMP: Mitigating the Compiler Phase-ordering problem using optimization subsequences and machine learning, is an autotuning framework to mitigate the compiler phase-ordering problem based on machine-learning techniques effectively. 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) 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 performed 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.

5.1 Intermediate Versus Full-Sequence Speedup Prediction

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 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.

A. H. Ashouri et al., *Automatic Tuning of Compilers Using Machine Learning*,
PoliMI SpringerBriefs, https://doi.org/10.1007/978-3-319-71489-9_5

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 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 [1, 2].
- (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 code. Although this technique has been extensively used in the selection problem of the compiler optimizations [3–6], there has been no work tackling the phase-ordering with the aforementioned method.

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 of 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. We refer to time as 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 [7] 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 [3–6]. 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 [8] 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 with-

out 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: Sect. 5.2 presents related work. Section 5.3 introduces our proposed methodology including all its components. In Sect. 5.4, we present our experimental results and evaluate the results by means of several comparisons in the Sect. 5.5. We conclude this chapter with future work and the conclusion.

5.2 Related Work

As we discussed in the previous chapter, literature [9] 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 Chap. 1 for the complete survey on the literature.)

Our approach, MiCOMP, is significantly different compared with those mentioned in the literature. Our work mostly resembles the approach of Park et al. [5, 6]. 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 [1, 2]. 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 Sect. 5.4, we improve the machine-learning model through Recommender Systems technique and assess the experimental results we obtain against the state-of-the-art phase-ordering approaches.

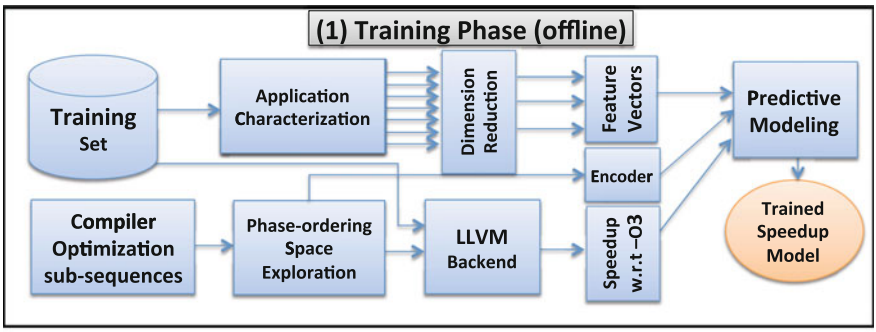
5.3 The Proposed Methodology

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`, to achieve the 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.

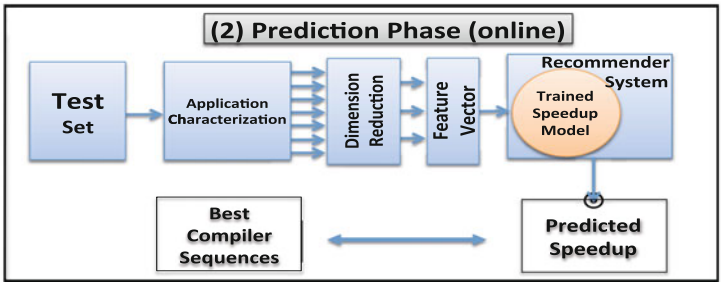
There are certain limitations facing the ever increasing complexity of the problem. The phase-ordering problem is also complicated by enabling the possibility of variable-length compiler sequences. State-of-the-art approaches for selecting the right set of optimizations used fixed length feature-vectors [3, 5, 6]. 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 apply traditional machine learning algorithms. We provide a simple encoding-scheme with mathematical deduction in Sect. 5.3.3. 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* [5, 6] techniques.

Figure 5.1 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



(a) Training MiCOMP



(b) Testing MiCOMP

Fig. 5.1 Proposed framework: offline-training phase which is done once, and online-prediction phase for optimizing new unseen applications

prediction model considering application features and applied optimizations (including order and repetitions). This phase is performed once for each compiler and the model is built on a set of representative applications. In this phase, each application is passed through a single round of feature collection 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. PCA (Principal Component Analysis) [10]). This speeds up the learning during the model construction process. Application-profiling and dimension-reduction techniques are extensively described in Sect. 5.3.1. 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 Sect. 5.3.3) are fed to a machine learning algorithm to induce the speedup predictor (see Sect. 5.3.4). 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 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 Sect. 5.3.5). 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.

5.3.1 Application Characterization

In this work, we used a PIN-based [11] 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) [12] 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. Readers can refer to Chap. 3 for more details on the MICA framework.

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.

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 [12]. In particular, we set $k = 5$, which captures more than 98% of the overall variance across all training sets.

5.3.2 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 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. Enabling optimizations to be repeatedly applied and a variable length sequence of optimizations will expand the problem space to:

$$|\Omega_{phases\_repetition}| = \sum_{i=0}^{m} n^i \tag{5.1}$$

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 [1].

5.3.2.1 The Optimization Dependence Graph

Mitigating the phase-ordering problem with previous approaches is not practical due to a 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.[1]

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 sub-sequences 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.

Let $o = \{o_1, \ldots, o_N\}$ 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 5.2 shows the constructed graph on `-O3`.

[1] http://llvm.org/docs/Passes.html.

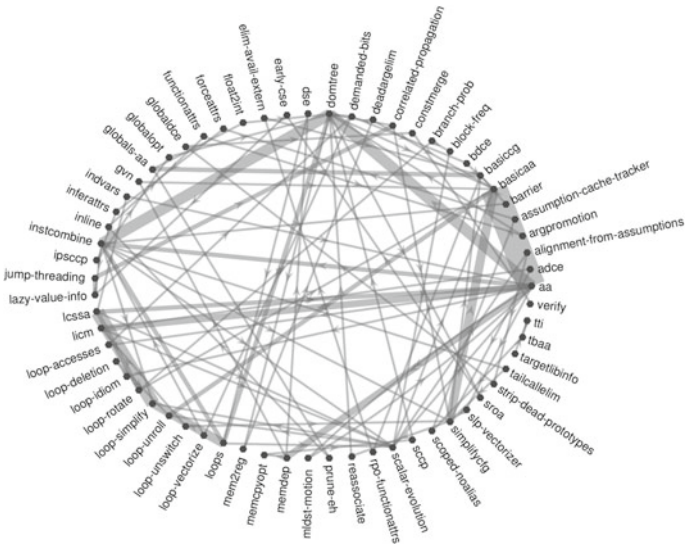


Fig. 5.2 Generated directed graph for LLVM's -O3. Each node in the graph represents an optimization pass. The edge thickness indicates the strengths in the connection between two nodes

5.3.2.2 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.,) [13]. We selected *agglomerative clustering* [14] which is an iterative clustering technique that merges smaller clusters and improves the complexity of k-mean clustering on graphs [13]. 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 $C = \{c_1, \ldots, c_{nc}\}$. (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*) [14]. 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 desired clusters until no clusters could be added. The final five clusters, namely, the best optimization sub-sequences the algorithm could find are reported in Sect. 5.4; Table 5.2.

5.3.2.3 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.

5.3.3 The Proposed Mapper

Constructing prediction models for the problem of selecting the right compiler optimizations with fixed-length feature vectors has been extensively studied [3–6]. 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 [15]. 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 $\mathscr{A} = \{\alpha_1, \ldots, \alpha_N\}$ 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 $\mathscr{W}$ 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 $f : \mathscr{A} \to \{0, 1\}^N$ 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 $F : \mathscr{W} \to \{0, 1\}^{N \times M}$ by mapping each word $\alpha_{i_1}, \alpha_{i_2} \cdots \alpha_{i_k}$ to the binary string

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

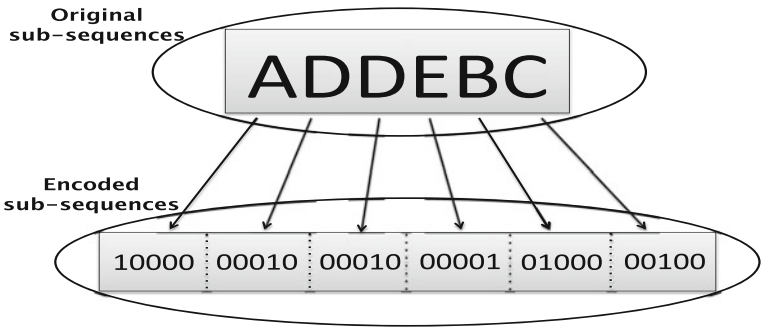


Fig. 5.3 An example of the proposed encoder where $\{N = 5, M = 6\}$: Each letter represents a compiler sub-sequences containing different compiler optimizations

where $\mathbf{0} = 0 \cdots 0$ is the zero string of length N. Evidently the map F is one-to-one. The image $F(\mathscr{W})$ 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.

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 $F(\mathscr{W})$ 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 $\mathscr{A} = \{\alpha_1, \ldots, \alpha_N\}$ 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 Fig. 5.3. Our adapted mapping function uses a *one-hot encoding* approach [16] 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.

5.3.4 Predictive Modeling

The methodology in Fig. 5.1 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 pedict the speedup of a complete optimization sequence without the need to actually apply the sequence of optimizations to the code.

5.3.4.1 Constructing the Prediction Model

Predictive modeling is the process of constructing, testing and validating a model to predict an unobserved outcome based on a 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.

5.3.4.2 Analysis of Selecting the Compilation Baseline

As explained in Sect. 5.3.2, 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 Sect. 5.3.2.2). 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 5.4 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 suggest 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

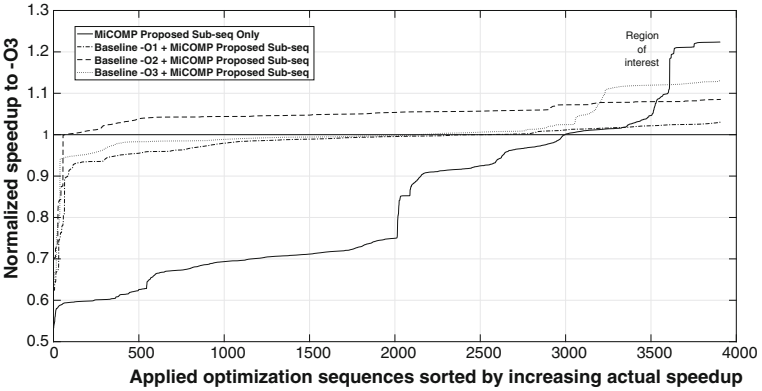


Fig. 5.4 Empirical analysis of having different compilation baseline across all CBench applications (Harmonic mean). Region of interest is depicted where MiCOMP sub-sequences outperformed other compiler sequences having a fixed standard compilation baseline

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.

5.3.4.3 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 an 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 [4, 6] 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.

5.3.5 *Recommender Systems Heuristic*

Mitigating the phase-ordering problem imposes a proper exploration strategy. In the initial steps taken by [1, 2], 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.

5.3.5.1 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 [7] 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:

$$sim(i,j) = \frac{\sum_{p \in P}(S_{p,i} - \bar{S_p})(S_{p,j} - \bar{S_p})}{\sqrt{\sum_{p \in P}(S_{p,i} - \bar{S_p})^2}\sqrt{\sum_{p \in P}(S_{p,j} - \bar{S_p})^2}} \tag{5.2}$$

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 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 fewer exploration steps.

5.4 Experimental Results

In this section, we evaluate our proposed methodology on an Intel Xeon architecture. We adapted our instrumentation and architecture-independent tool (Sect. 5.3.1) to extract characteristics from a large set of benchmarks from the Ctuning CBench suite [8]. 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 5.1. Table 5.2 illustrates the list of different compiler optimizations that are clustered into 5 different *sub-sequences* (refer to Sect. 5.3.2.2) that are derived from LLVM's `-O3`. We used the sub-sequences with

Table 5.1 Applications under analysis (CTuning CBench suite [8])

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

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. The 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,

Table 5.2 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

Table 5.3 List of the predictive models used in our experiments

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
Linear regression (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

Note that the proposed methodology is independent from any specific machine-learning algorithm (classifier) and it can be paired with any algorithm desired

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 5.3. 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 [17] has been integrated into our framework. We 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

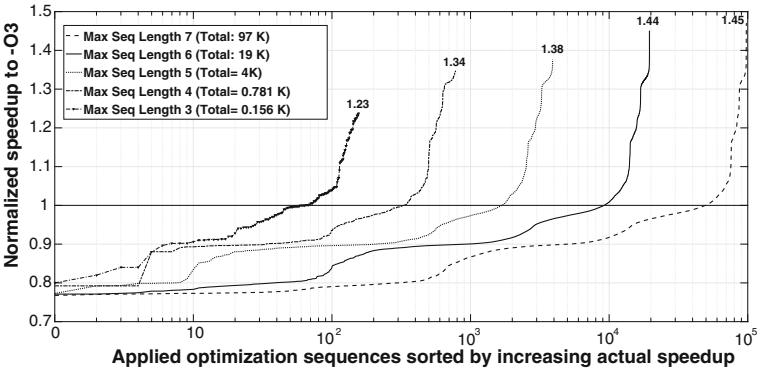


Fig. 5.5 Empirical analysis of having different compiler sequence lengths on 5 candidate applications: *telecom_adpcm_d, jpeg_d, bzipd, network_dijkstra, automotive_bitcount*. Note that X axis is in logarithmic scale

prediction purpose. In MiCOMP, cross validation is done in a few minutes for each application under analysis.[2]

5.4.1 Analysis of Longer Sequence Length

As described in Sect. 5.3, MiCOMP requires an upper bound on the sequence length for using the encoding scheme. To this end, we evaluate MiCOMP by 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.

Figure 5.5 gives the Harmonic mean (as suggested by [18].[3]) 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, 19 and 97 k distinct permutations of sub-sequences with repetitions enabled (refer to Eq. 5.1 for the optimization

[2]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.

[3]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.

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 100 K sequences per each application in the training set. For our optimization subsequences, this empirically found upper bound value is the right trade-off between choosing a good optimization sequence space and the possibility to explore efficiently.

5.4.2 MiCOMP Prediction Accuracy

Unlike sequence prediction models [6, 19, 20] 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.

5.4.2.1 Mean Absolute Error

In statistics, the Mean Absolute Error (MAE) [21] is a quantity used to measure how close predictions are to the eventual outcomes. The mean absolute error is given by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|f_i - y_i| \tag{5.3}$$

where we define e_i as $|f_i - y_i|$ given f_i as the prediction values and y_i the actual values. Consequently, the value e_i is inverse proportional to the accuracy of the prediction.

5.4.2.2 Approximation Error

Complementary to MAE, Approximation Error (AE) [22] 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:

$$\delta = \frac{|\varepsilon|}{|v|} = \frac{|v - v_{approx}|}{|v|} \tag{5.4}$$

Table 5.4 Average error rate for the proposed mapping function versus an arbitrary mapping

M.L	MiCOMP mapping function		No 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×

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.

5.4.2.3 Prediction Accuracy

We provide the prediction's error rate in Table 5.4. 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 5.4 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.[4]

5.4.2.4 Iterative Compilation Max Speedups

Iterative compilation is known to be able to achieve good performance results when compiling applications [23]. However, the approach is expensive and should be combined with more intelligent search algorithms [19, 20]. Table 5.5 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 5.4 illustrates the trend 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

[4] 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 5.5 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	ECDDCC	1.35
security_sha	ACCACE	1.06
security_blowfish_e	BCCEEA	1.03
automotive_susan_e	AABACA	1.15
consumer_tiffdither	DCEDCD	1.20
security_rijndael_e	CAEEC	1.10
consumer_tiff2bw	CCDCD	1.30
bzip2e	CBADCA	1.30
automotive_susan_s	ECCCDE	1.22
office_stringsearch1	ABCBAC	1.07
telecom_adpcm_d	DCAACA	1.13
consumer_jpeg_c	DDC	1.14
network_patricia	CECBAA	1.08
automotive_susan_c	BDBCCB	1.23
consumer_tiff2rgba	DEDDC	1.32
automotive_qsort1	CBAAAC	1.04
security_blowfish_d	DACECA	1.05
network_dijkstra	EECBBE	1.51
security_rijndael_d	ECEACD	1.05
bzip2d	CBDACA	1.29
automotive_bitcount	BEACCA	1.07
consumer_jpeg_d	CCED	1.18
consumer_tiffmedian	BCBACB	1.15
telecom_gsm	BACBAC	1.07
Harmonic mean		1.26

and the third columns of Table 5.5. Readers can refer to Table 5.2 to find the exact set of compiler optimizations clustered in each sub-sequence.

5.4.3 MiCOMP Versus 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 5.6 reports the comparison between the best speedup found by our approach and a state-of-the-art N-shot approach [4, 6]. The results, averaged using a Harmonic mean across all applications, show that using the same

Table 5.6 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

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.

5.5 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 5.7, 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 Fig. 5.5 that it takes a pure iterative compilation technique 100–5000 iterations to surpass the performance of `-O3`.

5.5.1 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

Table 5.7 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%)

per architecture/application/dataset. As we showed in Table 5.7, MiCOMP can surpass the performance of `-O3` with a few predictions on application bases. Here we provide Table 5.8 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.

Table 5.8 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%)

5.5.2 Comparisons with State-of-the-Art Models

In this section, we compare MiCOMP with two state-of-the-art intermediate-sequence prediction approaches proposed in [1, 2].

5.5.2.1 Intermediate Speedup Comparison Case. (A)

Kulkarni et al. [2] 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 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. [2] 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 5.9. 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 19 k 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× speedup (5% performance improvement) against the mentioned work.

Table 5.9 Performance comparison of the single prediction by MiCOMP against the intermediate speedup approach reported in in previous work [2]

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

All values are normalized by `-O3`

5.5.2.2 Intermediate Speedup Comparison Case. (B)

Ashouri et al. [1] 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 Sect. 5.5.2.1, 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 setting in LLVM, that is, `-O3`, in only a few predictions.). Figure 5.6 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.

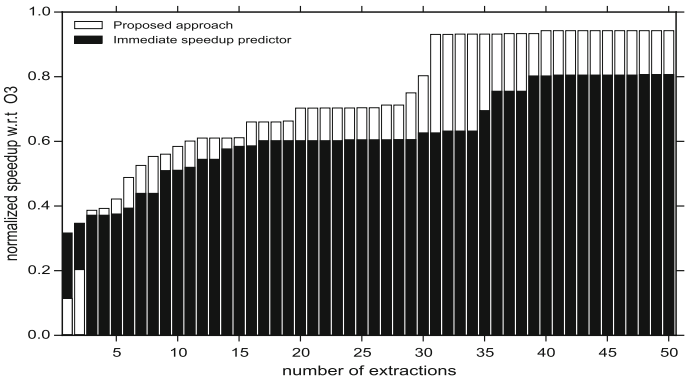


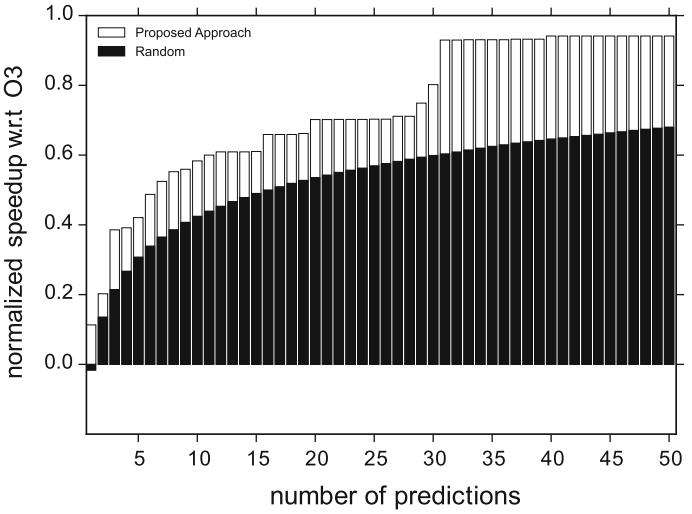
Fig. 5.6 Performance of MiCOMP w.r.t the performance of intermediate speedup predictor approach [1]

5.5.3 Comparison with Random Iterative Compilation

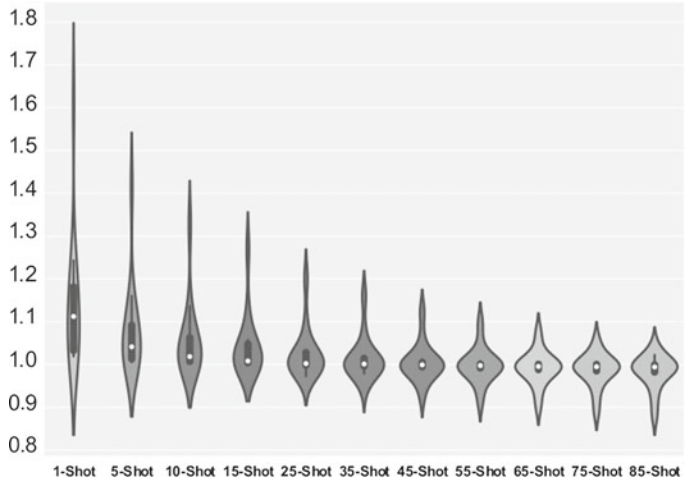
As we illustrated in Sect. 5.4.2.4, iterative compilation can improve application performance over standard compiler optimization sequences [19, 23]. Additionally, several published works have shown that drawing compiler optimization sequences at random can often be as good as using other more complicated search algorithms, such as genetic algorithms or simulated annealing [3, 19, 24]. 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 Fig. 5.7. To present our results, we define *Normalized Performance Improvement* (NPI) as the ratio of the performance improvement achieved over the potential performance improvement:

$$NPI = \frac{E_{ref} - E}{E_{ref} - E_{best}} \tag{5.5}$$

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 5.7a shows results for both MiCOMP and RIC with the same. The X axis pertains to the number of predicted optimization sequences



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



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

Fig. 5.7 MiCOMP performance comparison versus Random Iteration Compilation

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 5.5 gives the the absolute speedup values.

Figure 5.7b 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 5.7b 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× exploration speedup versus the RIC method.

5.6 Conclusion

This chapter presented 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 outperforms 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 outperforms 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 a variety of different optimization sequences.

References

1. Ashouri AH, Bignoli A, Palermo G, Silvano C (2016) 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, pp 7–12, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2872421.2872424
2. Kulkarni S, Cavazos J (2012) Mitigating the compiler optimization phase-ordering problem using machine learning. ACM SIGPLAN Notices
3. Cavazos J, Fursin G, Agakov F (2007) Rapidly selecting good compiler optimizations using performance counters. In: International symposium on code generation and optimization (CGO'07)
4. Park E, Cavazos J, Alvarez MA (2012) Using graph-based program characterization for predictive modeling. Proceedings of the international symposium on code generation and optimization, pp 295–305
5. Park E, Cavazos J, Pouchet LN (2013) Predictive modeling in a polyhedral optimization space. Int J Parallel Prog, 704–750
6. Park E, Kulkarni S, Cavazos J (2011) An evaluation of different modeling techniques for iterative compilation. In: Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, pp 65–74
7. Sarwar B, Karypis G, Konstan J, Riedl J (2001) Item-based collaborative filtering recommendation algorithms. In: Proceedings of the 10th international conference on World Wide Web, pp 285–295. ACM
8. Fursin G (2010) 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
9. Ashouri AH, Killian W, Cavazos J, Gianluca P, Silvano C (2018) A survey on compiler autotuning using machine learning, ACM Computing Surveys (CSUR)
10. Johnson RA, Wichern DW (2002) Appl Multivar Stat Anal, vol 5. Prentice hall Upper Saddle River, NJ
11. Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) 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, pp 190–200, New York, NY, USA, ACM
12. Hoste K, Eeckhout L (2007) Microarchitecture-independent workload characterization. IEEE Micro 27(3):63–72
13. Schaeffer SE (2007) Graph clustering. Comput Sci Rev 1(1):27–64
14. Zhang W, Zhao D, Wang X (2013) Agglomerative clustering via maximum incremental path integral. Pattern Recogn 46(11):3056–3065
15. Agresti A, Liu I et al (1999) Modeling a categorical variable allowing arbitrarily many category choices. Biometrics 55(3):936–943
16. Orup H (1999) On-the-fly one-hot encoding of leading zero count, October 26. US Patent 5,974,432
17. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. ACM SIGKDD Explor Newsl 11(1):10–18
18. Hoefler T, Belli R (2015) 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, p 73. ACM
19. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI (2006) Using machine learning to focus iterative optimization. In: Proceedings of the international symposium on code generation and optimization, pp 295–305. IEEE Computer Society

20. Ashouri AH, Mariani G, Palermo G, Park E, Cavazos J, Silvano C (2016) Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim. (TACO)*, 13(2):21:1–21:25, June 2016. http://doi.acm.org/10.1145/2928270
21. Hyndman RJ, Koehler AB (2006) Another look at measures of forecast accuracy. Int J Forecast 22(4):679–688
22. Smale S, Zhou D-X (2003) Estimating the approximation error in learning theory. Anal Appl 1(01):17–41
23. Bodin F, Kisuki T, Knijnenburg P, O'Boyle M, Rohou E (1998) Iterative compilation in a non-linear optimisation space. In: Workshop on Profile and Feedback-Directed Compilation
24. Chen Y, Fang S, Huang Y, Eeckhout L, Fursin G, Temam O, Chengyong W (2012) Deconstructing iterative optimization. ACM Trans Archit Code Optim (TACO) 9(3):21

Chapter 6
Concluding Remarks

In this book, 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 6.1 summarizes the main contribution and Sect. 6.2 provides a list of open issues and our future direction towards tackling those challenges.

6.1 Main Contributions

This book presented the following contributions.

1. We extensively surveyed the literature of the past 25 years and classified the papers by the type of contribution, the prediction models, and many other sub-features. We hope the survey is useful for a wide range of researchers and industry professionals.
2. We provided a co-exploration framework to statistically analyzing the compiler level parameters over a customized VLIW architecture. Moreover, we applied several statistical tests, e.g., ANOVA, Kruskal-Wallis, and clustering techniques and we showed that not all the available compiler parameters are beneficial to use within the embedded domain.
3. Chapter 4 presented COBAYN framework for autotuning compiler passes using Bayesian networks. It uses application's characteristics to predict the best set of optimization passes on given application. We experimentally showed that this framework outperforms GCC's standard optimization levels, i.e., `-O2` and `-O3` and betters the state-of-the-art approaches.
4. An intermediate speedup predictor was presented towards tackling the phase-ordering problem in Chap. 4. Choosing the best local alternative as the next-best

A. H. Ashouri et al., *Automatic Tuning of Compilers Using Machine Learning*,
PoliMI SpringerBriefs, https://doi.org/10.1007/978-3-319-71489-9_6

optimization to use might lead to a local minimum, whereas other paths with less steep initials, might end up in a better global point. The main goal of this chapter was to familiarize with the difficult problem of the phase ordering of the compiler optimizations and an intuitive approach to tackling such.
5. Last but not least, MiCOMP was presented as a framework for mitigating the phase-ordering problem. It uses predictive modeling to form the best ordering of phases for an application under analysis. A major advantage of MiCOMP is the fact that it predicts the full optimization sequence at once. We experimentally showed that the framework outperforms state-of-the-art techniques, also within a few predictions it can outperform standard optimization levels.

6.2 Open Issues and Future Directions

We identify the challenges and open issues involved in optimizing compilers.

1. Leveraging COBAYN and MiCOMP address a few research questions. However, these approaches can be further improved and be more accurate. Inducing prediction models always deals with approximations; therefore there wouldn't be a definite optimal result for a given problem. By the advancement in new machine learning techniques, i.e., deep learning, etc. it is expected that we adapt more accurate models to achieve results, yet this remains an open question to be answered.
2. Parallel and heterogeneous computing brought both new applications and challenges for autotuning. Now, it is up to the compiler researchers to adapt and introduce novel tuning techniques for CPU, GPU, and other accelerators. This direction is indeed both challenging and interesting for future research.
3. Multi-objective optimization strategies need to be exploited for power-aware and energy efficient systems. Finding the right Pareto-curve representing the right set of optimization is still an open challenge.

We hope that the book paves the way to 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.

Index

A. H. Ashouri et al., *Automatic Tuning of Compilers Using Machine Learning*,
PoliMI SpringerBriefs, https://doi.org/10.1007/978-3-319-71489-9