Compilers that Learn to Optimise:
A Probabilistic Machine Learning Approach

Edwin V. Bonilla

Doctor of Philosophy
Institute for Adaptive and Neural Computation
School of Informatics
University of Edinburgh
2008

Abstract

Compiler optimisation is the process of making a compiler produce *better* code, i.e. code that, for example, runs faster on a target architecture. Although numerous program transformations for optimisation have been proposed in the literature, these transformations are not always beneficial and they can interact in very complex ways. Traditional approaches adopted by compiler writers fix the order of the transformations and decide when and how these transformations should be applied to a program by using hard-coded heuristics. However, these heuristics require a lot of time and effort to construct and may sacrifice performance on programs they have not been tuned for.

This thesis proposes a *probabilistic* machine learning solution to the compiler optimisation problem that automatically determines "good" optimisation strategies for programs. This approach uses predictive modelling in order to search the space of compiler transformations. Unlike most previous work that learns when/how to apply a single transformation in isolation or a fixed-order set of transformations, the techniques proposed in this thesis are capable of tackling the general problem of predicting "good" *sequences* of compiler transformations. This is achieved by exploiting *transference* across programs with two different techniques: Predictive Search Distributions (PSD) and multi-task Gaussian process prediction (multi-task GP). While the former directly addresses the problem of predicting "good" transformation sequences, the latter learns regression models (or proxies) of the performance of the programs in order to rapidly scan the space of transformation sequences.

Both methods, PSD and multi-task GP, are formulated as general machine learning techniques. In particular, the PSD method is proposed in order to speed up search in combinatorial optimisation problems by learning a distribution over good solutions on a set of problem instances and using that distribution to search the optimisation space of a problem that has not been seen before. Likewise, multi-task GP is proposed as a general method for multi-task learning that directly models the correlation between several machine learning tasks, exploiting the shared information across the tasks.

Additionally, this thesis presents an extension to the well-known analysis of variance (ANOVA) methodology in order to deal with sequence data. This extension is used to address the problem of optimisation space characterisation by identifying and quantifying the main effects of program transformations and their interactions.

Finally, the machine learning methods proposed are successfully applied to a data set that has been generated as a result of the application of source-to-source transformations to 12 C programs from the UTDSP benchmark suite.

Acknowledgements

First of all I would like to thank my principal Supervisor Prof. Chris Williams for his invaluable guidance throughout my PhD studies. I have greatly benefited from his extensive knowledge and expertise in machine learning. I also thank Prof. Michael O'Boyle and Dr. Amos Storkey for their assistance and feedback as members of my PhD research committee.

I am grateful to all members of the COLO and MilePost projects, especially to Prof. Michael O'Boyle who has led numerous discussions in the compiler optimisation area and to Felix Agakov for helpful and extensive discussions in various interesting machine learning topics. Thanks to Björn Franke who developed the software tool that has been the basis for the generation of the data used in this project and to John Cavazos and Christophe Dubach for their active work on this tool and for providing the code features that have been used for the results presented in Chapter 7 and Chapter 8. I also thank John Thomson for contributing to the maintenance and development of such software and for the generation of the data for the AMD architecture.

In addition to my principal supervisor, I would like to thank those that have provided me feedback on specific drafts of this thesis: Prof. Michael O'Boyle (Chapter 2), Adrian Haith (Chapters 4, 5 and 6) and Catalina Voroneanu (Chapters 3, 7 and 8).

This work has been supported under EPSRC grant GR/S71118/01 (*Compilers that Learn to Optimize*) and EU FP6 STREP MILEPOST IST-035307.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Edwin V. Bonilla*)

To my mother María Inés Pabón.

Table of Contents

List of Figures

List of Tables

Notation and Abbreviations

General Statistical and Mathematical Notation

Matrices are denoted by capital letters (e.g. A), vectors by lower case bold letters (e.g. $\mathbf{x}$) and sets by upper case calligraphic letters (e.g. $\mathcal{A}$).

Symbol	Meaning
$\sim$	Distributed as
$\mathcal{N}(\mu, \Sigma)$	Normal distribution with mean μ and covariance Σ
$\lvert\mathcal{A}\rvert$	The cardinality of set $\mathcal{A}$
$\stackrel{\text{def}}{=}$	An equality that acts as a definition
$\mathbf{w}^T$	The transpose of vector $\mathbf{w}$
$(A)_{ij}$	The (i,j)th entry of matrix A
$\lVert\mathbf{w}\rVert$	The Euclidean length of vector $\mathbf{w}$
$\lVert\mathbf{w}\rVert_\infty$	The infinity norm of vector $\mathbf{w}$
iid	Independent and identically distributed
cdf	Cumulative distribution function
$\mathrm{KL}(p,q)$	The Kullback-Leibler divergence of distributions p and q
$\langle\cdot\rangle_{\mathbf{x}}$	Expectation over $\mathbf{x}$
$\log(\theta)$	The natural logarithm of θ
$\mathcal{GP}$	Gaussian process: $f(\mathbf{x}) \sim \mathcal{GP}(\mu(\mathbf{x}), k(\mathbf{x},\mathbf{x}'))$, the function $f(\mathbf{x})$ is distributed as a Gaussian process with mean function $\mu(\mathbf{x})$ and covariance function $k(\mathbf{x},\mathbf{x}')$
$k(\mathbf{x},\mathbf{x}')$	Covariance function evaluated at $\mathbf{x}$ and $\mathbf{x}'$
$\mathrm{diag}(\boldsymbol{\ell})$	A diagonal matrix containing the elements of vector $\boldsymbol{\ell}$
δ_{pq}	Kronecker delta: $\delta_{pq} = 1$ if $p = q$ and $\delta_{pq} = 0$ otherwise
$\mathbb{E}$	Expectation
$\lvert A\rvert$	The determinant of matrix A
$\mathrm{tr}\,(A)$	The trace of matrix A
$\bullet$	The Hadamard product
$\propto$	Proportional to
$\mathrm{vec}(Y)$	The vector obtained by stacking the columns of matrix Y

Symbol	Meaning
$\approx$	Approximately equal to
$\otimes$	The Kronecker product

Notation Used for Multi-task Learning (and Compiler Optimisation)

Symbol	Meaning
$\mathbf{x}$	A representation for an input point (or a compiler transformation sequence) X
$\mathbf{t}$	A representation for task (or program) T
y	The target value (or performance speed-up)
M	The number of training tasks (or programs)
N	The number of training inputs (or transformation sequences) per task

Chapter 1

Introduction

This chapter presents an introduction to the compiler optimisation problem and an overview of the machine learning approach to compiler optimisation proposed in this thesis. The motivation for using machine learning in order to tackle the compiler optimisation problem is presented in section 1.1. The general approach to compiler optimisation with machine learning proposed in this thesis is given in section 1.2. The contributions and the impact of the research carried out in this thesis, and the declaration of collaborations with other researchers are described in sections 1.3, 1.4 and 1.5. Finally, the general structure of the remainder of this document is presented in section 1.6.

1.1 Motivation

The rapid advance of microprocessor architectures with the construction of highly integrated systems has made possible the effective delivery of Moore's law. However, there is a tendency for current applications to soak up this extra speed with additional software complexity. This, in some sense, has overshadowed the invaluable effort of hardware designers and has forced the scientific community to start looking for alternatives that can satisfy current technological requirements.

Nonetheless, even the most powerful machines can be underexploited and prevented from running at maximum speed. The reason for this lies on an application responsible for the translation of a code written in a high level programming language into a code that a machine can directly understand: **the compiler**. In fact, unlike hardware components that have continuously changed in order to satisfy user requirements, most compilers have remained with static structures based on designs proposed many years ago. Thus, compilers have become the bottleneck in the development of new applications for the 21st century.

The compiler community has been aware of this problem and has proposed numerous program transformations that allow a compiler to create *better code*. The problem of making a

compiler produce better code, that for example runs faster, is known as **compiler optimisation**. However, it is widely accepted that such program transformations for compiler optimisation are not always beneficial and that they can *interact* in very complex ways. Therefore, determining when and how these transformations should be applied to a program has become the new problem to be solved[1].

Traditional approaches adopted by compiler writers fix the order of the transformations and decide when and how these transformations should be applied to a program by using hard-coded heuristics. However, these heuristics require a lot of time and effort to construct and may sacrifice performance on programs they have not been tuned for.

Searching for good sequences of compiler transformations yields indeed significant improvements over baseline compiler optimisations and over fixed-ordered sets of transformations. For example, Cooper et al. (1999) showed that such an approach leads to improvements in execution time of 14 different programs from 20% to 83% and Franke et al. (2005) obtained an average speed-up of 1.71 across three different platforms and 13 benchmarks.

Consequently, what the compiler optimisation problem requires is a solution that automatically generates these heuristics and discovers optimisation opportunities even for programs that have never been seen before. Thus, an ideal compiler should be able to maximally exploit the physical resources of a target architecture, to adapt to different environments such as operating systems and target architectures and, more importantly, it should be able to *tune itself* in order to optimise programs.

1.2 Compiler Optimisation and Machine Learning

Considering the general goals of an ideal compiler, a natural solution to the problem can be formulated with **machine learning** techniques. The general idea of machine learning is that of learning from past experience. Thus, given a database containing examples of good sets of transformations for different programs, a model can be constructed with the hope of capturing the knowledge that features of the programs provide about "good" code transformations. Additionally, given a new program that has not been seen before, such a model should be able to generalise across programs and predict a set of transformations that lead to an improvement in performance of its execution time.

This thesis is about compiler optimisation using machine learning. Therefore, the general goal is to **investigate the application of machine learning techniques to compiler optimisation**. However, the work presented in this thesis does not adopt a "black-box" approach to

[1]Note that although in this thesis the problem of compiler optimisation is focused on finding good sequences of transformations that improve the performance of a program, there is also a lot of effort and research in developing new compiler transformations and in providing better analyses so as to enable the application of program transformations.

the problem, where standard machine learning techniques are used in order to evaluate their suitability to this specific application. On the contrary, it considers the characteristics of the problem and the state-of-the-art methods in machine learning in order to push the boundaries on both fields: compiler optimisation and machine learning. Thus, the techniques proposed in this thesis do not only attempt to solve the compiler optimisation problem but they also develop novel machine learning approaches that can be used in other applications.

A general framework for compiler optimisation based upon machine learning techniques identifies two different targets: *global optimisation* and *predictive modelling*. Global optimisation is the task of finding the best or at least a good set of transformations for a given program. Predictive modelling uses the output of global optimisation and a set of features that describe the programs in order to predict a suitable set of transformations for a new program. Unlike previous approaches that consider these problems independently, this thesis proposes a unified framework that uses predictive modelling in order to search the space of compiler transformations (i.e. global optimisation). Furthermore, unlike most previous approaches that deal with a fixed-order set of transformations, the techniques proposed in this thesis are formulated in a general way so that they are capable of tackling the problem of predicting "good" *sequences* of compiler transformations.

Two different approaches are proposed herein in order to address the compiler optimisation problem. The main goal of both approaches is that of achieving *transference* across programs. In other words, exploiting the shared information about "good" compiler transformation sequences and their performances across different programs.

In the first approach, "good" compiler transformation sequences are modelled directly by learning a Predictive Search Distribution (PSD, Chapter 7) on a set of training programs. Given a new program, the predictive distribution is used in order to focus search.

In the second approach to compiler optimisation proposed in this thesis, the performance of a program obtained by the application of transformation sequences is modelled by learning regression functions across multiple benchmarks (multi-task Gaussian process prediction, Chapter 8). These predictors are used as *proxies* of the performance of a new program in order to search for "good" compiler transformation sequences.

Both approaches, are formulated in order to deal with general machine learning problems. While the predictive search distribution technique is proposed as a method to tackle combinatorial optimisation problems that can be described by a set of features, multi-task Gaussian process prediction is proposed as a technique for effectively exploiting transference across different but related machine learning tasks. Figure 1.1 illustrates the general idea of an adaptive optimising compiler based on machine learning. See section 3.1 for more details.

In addition to predicting "good" compiler transformations for programs, there is a general interest in the compiler community in the characterisation of the search space in compiler opti-

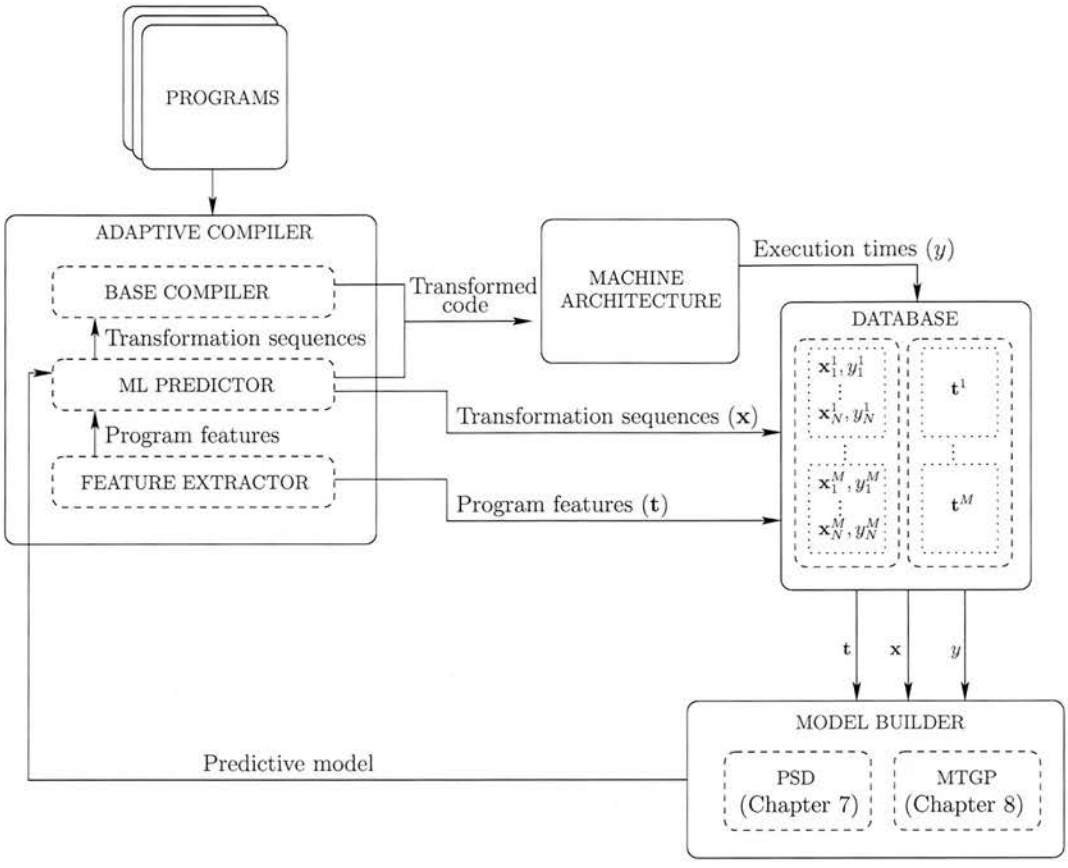

Figure 1.1: The general framework for an adaptive optimising compiler. A machine learning (ML) approach builds predictive models based on transformation sequences $\mathbf{x}$, their execution times (or speed-ups) y and program features $\mathbf{t}$. The key idea is that the constructed models are used to drive optimisations on unseen programs. For simplicity in the notation it is assumed that there are M training programs, for which N transformation sequences are evaluated in order to build the models. Two different approaches to building these models are proposed in this thesis: Predictive Search Distributions (PSD) and multi-task Gaussian process (MTGP) prediction.

misation (see section 6.5.3 for details). In particular, this interest is focused on the problem of identifying and quantifying the effects of program transformations and their interactions. This thesis proposes a solution to this problem by extending the well-known analysis of variance (ANOVA) methodology in order to deal with sequence data (Chapter 6).

A data set useful for the analysis and application of machine learning techniques to compiler optimisation has been generated. This data set is based upon the application of source-to-source transformations to 12 C programs from the UTDSP suite (Lee, 1997) and will be called throughout this thesis as the SUIF data set. Most results presented in this thesis are based upon a subset of this data that is a complete enumeration of sequences of up to length 5 drawn from 14 code transformations. This will be referred to as the small space of the SUIF data set. (See Chapter 5 for more details.)

1.3 Contributions

The specific contributions made on this thesis are the following:

1. A general framework for compiler optimisation based upon machine learning techniques is proposed. This framework tackles the problems of global optimisation and predictive modelling in a unified manner by using a *transfer learning* approach. Thus, transference is exploited across different programs by learning predictive models on these programs in order to search the optimisation space of programs that have not been seen before, or programs for which very little data is available. Within this framework, a direct or an indirect approach can be adopted. In the direct approach, the problem is formulated as a sequential prediction task, i.e. predicting "good" transformation sequences. In the indirect approach, the optimisation task is formulated as a regression problem where proxy models of the performance of the programs under the application of compiler transformation sequences are constructed, which are then used in order to search the optimisation space of new programs or programs for which very little data is available (Chapter 3).

2. The direct approach to compiler optimisation (i.e. the sequential prediction task) is addressed with the technique of Predictive Search Distributions (PSD), which is proposed as a general method for speeding up search on combinatorial optimisation problems. The main idea is to learn a distribution over good solutions on a collection of optimisation problems that can be characterised by a set of features and use this distribution to focus search on a problem that has not been seen before.

   Thus, the method of Predictive Search Distributions (PSD) is used to learn a distribution over "good" compiler transformation sequences across different programs, and this distribution is utilised to focus the search of transformation sequences when a new program is presented. Significant improvements in performance are achieved by this method on the SUIF data set (Chapter 7).

3. The indirect approach to compiler optimisation (i.e. the use of performance models for optimisation) is formulated with the Multi-task Gaussian process prediction technique, which is proposed as a general method for achieving *transference* across different machine learning tasks. The general idea is that of exploiting the shared information across the different tasks by directly modelling the correlations between them. This method can be used when task-features are available (multi-task GP with task-specific features) or when these features are unavailable or are difficult to define correctly (multi-task GP without task-specific features). An important characteristic of the technique is that observations on one task affect the predictions on the others.

Thus, Multi-task GP is used to exploit the shared information across different programs and their performances in order to predict the performance speed-up of a program when being applied a sequence of code transformations. This method is shown, in general, to outperform the "no transfer" scenario (i.e. learning each performance prediction task on a single benchmark basis without using data from the other programs). Additionally, the predictions obtained with Multi-task GP are used to search the optimisation spaces of the (small) SUIF data set, and significant speed-ups are obtained (Chapter 8).

4. The problem of identifying and quantifying the main effects of program transformations and their interactions is approached by extending the well-known statistical technique of analysis of variance (ANOVA) to deal with sequence data. Results are reported on the small space of the SUIF data set (Chapter 6, sections 6.4 and 6.5).

5. An extensive review and characterisation of the related work on compiler optimisation with machine learning or/and artificial intelligence is presented. This review is focused on the problems of *global optimisation*, *predictive modelling*, *performance prediction* and optimisation space *characterisation* (Chapter 4).

1.4 Impact: Publications During PhD Studies

The work presented in this thesis has had a theoretical and practical impact on both areas of research: compiler optimisation and machine learning. This is reflected on the publication of the following papers that contain or are based on some of the ideas developed in this thesis:

- *Predictive search distributions* (in Proceedings of the 23rd International Conference on Machine Learning, Bonilla et al., 2006).
- *Kernel multi-task learning using task-specific features* (in Proceedings of the 11th International Conference on Artificial Intelligence and Statistics, Bonilla et al., 2007).
- *Multi-task Gaussian process prediction* (to appear in Advances in Neural Information Processing Systems, Bonilla et al., 2008).
- *Using machine learning to focus iterative optimization* (in Proceedings of the International Symposium on Code Generation and Optimization, Agakov et al., 2006).
- *Automatic performance model construction for the fast software exploration of new hardware designs* (in Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Cavazos et al., 2006).

- *Rapidly selecting good compiler optimizations using performance counters* (in Proceedings of the International Symposium on Code Generation and Optimization, Cavazos et al., 2007).

1.5 Declaration of Collaborations

As mentioned in section 1.4, parts of the material presented herein have been done in collaboration with other researchers and have been previously published. In particular, Chapter 7 is mostly based on Bonilla et al. (2006) and some of the analysis has also been published in Agakov et al. (2006). Chapter 8 is mostly based on Bonilla et al. (2007, 2008) and some of the ideas regarding performance prediction have also been presented in Cavazos et al. (2006) although with different methods. Björn Franke developed the original tool that has been used for the generation of the data used throughout this thesis. John Cavazos and John Thomson significantly contributed to the improvement of this tool and John Thomson generated the data corresponding to the AMD architecture.

The idea of Predictive Search Distributions (Chapter 7) was originated in one of the COLO meetings based on a proposal by John Thomson of doing clustering on the program feature space. Marc Toussaint's expertise on Estimation of Distribution Algorithms (EDAs) proved crucial on my interest in this area and Figure 7.1 is based upon a figure included in one of his presentations at a COLO meeting. I also thank Felix V. Agakov for pointing out the relevant work regarding the improvement of the performance of EDAs (section 7.2.5) and for his collaboration on the description of an EDA algorithm (section 7.2.1). The program features used for the results presented in this chapter were provided by John Cavazos.

My interest in multi-task learning was originally motivated by some collaboration with Felix V. Agakov who proposed the gating network method (section 8.4.2). The idea of the canonical responses was originated as a result of several discussions with Felix V. Agakov and John Cavazos. I have also benefited from multiple discussions with Kian Ming A. Chai who pointed out the work in Zhang (2007) and contributed to the review of the relevant literature regarding Gaussian process approximations (section 8.5.5) and some of the related work (section 8.7). He and I jointly worked out the result of cancellation of inter-task transfer for noiseless observations and a grid design (section 8.5.4). Finally, Christophe Dubach provided the code features used for the results in Figure 8.3.

1.6 Organisation

Before describing how this thesis is structured, it is important to emphasise that the reader is not expected to have a background on compilers. However, some familiarity with basic

concepts in machine learning and statistics is expected. In particular, the following is a (non-exhaustive) list of topics that can prove helpful to understand most concepts explained in this thesis: learning from past examples, generalisation, overfitting, supervised learning, unsupervised learning, feature selection, feature extraction, regression, classification, basic probability theory, the Gaussian distribution, maximum likelihood estimation and parameter optimisation. A suitable reference for an introduction to machine learning is given e.g. by Bishop (2006, chapters 1 to 3). The structure of the remainder of this thesis is given below.

Chapter 2 introduces the **compiler optimisation problem** by explaining basic concepts in compilation and compiler optimisation. The definition of a program transformation is presented and some examples of transformations are given. The chapter ends with the description of a popular approach in compiler optimisation called *iterative compilation* and a description of the measure of performance (i.e. the speed-up) of a program when being applied a sequence of transformations.

Chapter 3 presents the **general framework for compiler optimisation** with machine learning proposed in this thesis. The problems of global optimisation and predictive modelling are described and a *transfer* learning approach is proposed in order to tackle these problems. Additionally, compiler optimisation is formulated from two different perspectives: as a sequence prediction problem and as a performance prediction task.

Afterwards, the most relevant **work related to compiler optimisation with machine learning** and/or artificial intelligence techniques is reviewed in Chapter 4. Previous work is categorised in five different areas: global optimisation, predictive modelling, performance prediction, optimisation space characterisation and unsupervised learning. The related literature is described and their contributions, caveats and differences with the techniques proposed in this thesis are presented.

The **experimental set-up** underlying the generation of the data set that is used for the application of the models and techniques proposed in thesis is presented in Chapter 5. Thus, the benchmarks, transformations, compiler infrastructure and target architectures involved in the experiments are described in this chapter. The results presented throughout this thesis are mostly based on an exhaustively enumerated space (described in Chapter 5) of sequences of up to length 5 drawn from 14 code transformations and applied to 12 different benchmarks. This space is called the small space of the SUIF data set.

This data set is analysed in Chapter 6 by providing a description of the performances achieved with the experiments and a discussion of the difficulty of the search spaces of the benchmarks used. Additionally, the **ANOVA for sequence data** technique is proposed and applied to the small space of the SUIF data set with the goal of identifying and quantifying the effects of program transformations and their interactions.

The technique of **Predictive Search Distributions** is proposed in Chapter 7 as a method

for tackling combinatorial optimisation problems that can be characterised by a set of features. This method is based on learning a distribution over "good" solutions to these optimisation problems and using such distribution in order to focus search on a problem that has not been seen before. Learning and predictions with this method are explained and the results of applying this method to the created data set are also presented and evaluated.

Multi-task Gaussian process (GP) prediction is proposed in Chapter 8 as a general method for achieving *transference* across different (but related) machine learning tasks. The focus is on regression problems for which there may be or there may not be a set of features that characterise each problem. Additionally, the task of predicting the performance speed-up of a program when applying a sequence of code transformations is formulated with this technique and the results on the small space of the SUIF data set are given. The predictions of the regression models constructed are used in order to find "good" compiler transformations for the programs on such data set and the results on this task are also presented.

Finally, Chapter 9 concludes with a **summary** and a discussion, and describes **specific areas of future research** in compiler optimisation with machine learning in order to continue the work presented in this thesis.

Chapter 2

The Compiler Optimisation Problem

Chapter 1 (section 1.1) has emphasised that achieving peak performance in current architectures is highly dependent on a particular application responsible for the translation of a program written in a high-level language into a code that a machine can actually understand: *a compiler*. Compilers are, in general, very large and complex software applications that map high-level languages into low-level languages effectively. Understanding how compilers work and the underlying issues of constructing a compiler requires at least a complete course at undergraduate level and in fact, entire books are dedicated to the study of these particular applications. Nevertheless, this chapter aims at describing the most important concepts in compilation and compiler optimisation, which may be required in order to understand the following chapters. Most ideas explained in this chapter are based on Cooper and Torczon (2004, chapters 1, 8 and 10), Bonilla (2004, Chapter 2) and Bacon et al. (1994). The interested reader is referred to e.g. Cooper and Torczon (2004) for a deeper description and explanation of the main tasks performed by a compiler and the key issues involved in compiler construction and compiler optimisation. Additionally, a classic text book in compilers is Aho et al. (2006).

The organisation of this chapter is as follows. The main concepts in compilation, the general structure of a compiler and the goals of a modern compiler are explained in sections 2.1, 2.2 and 2.3 respectively. The problem of compiler optimisation is described in section 2.4. Afterwards, the concept of program transformations, their scope, classification, and some examples of transformations are given in section 2.5. Subsequently, the popular approach of iterative compilation to compiler optimisation is presented in section 2.6. Finally, the speed-up measure is defined in section 2.7 and a summary of the chapter is given in section 2.8.

2.1 Basic Concepts in Compilers

In general, a compiler can be thought as a large piece of software that takes a representation of a program (e.g. in a high-level programming language such as C or Fortran) and outputs a version

of this program into another representation. This latter representation can be, for example, a language that a specific architecture can actually understand, and therefore the ultimate code generated by a compiler can be executed in such architecture. The crucial guarantee that a compiler must provide is that the *meaning* of the original program must be preserved. Indeed, it would make no sense to transform a program if its meaning was not maintained. This is known in the literature as *correctness* and it is the most important principle of *compilation*.

Thus, in short, we can define compilation as the process of transforming or translating a program from one version to another while maintaining the meaning of the original program. Commonly, the original version of the program is written in a high-level programming language such as C or Fortran and the ultimate code generated by the compiler is an *object code* that can be executed on a specific architecture.

A compiler may also want to *transform* a program in a given representation into a version of this program in the same representation. For example, a compiler can receive as its input a program written in C and produce as its output another C version of this program. In this case the goal may well be, for example, to produce a program that runs faster or that occupies less space than the original one. In any case, such a compiler would generate a code that can be portable from architecture to architecture as, for example, there are C compilers for most platforms. An example of such "source-to-source" translator is the SUIF compiler (Hall et al., 1996), which is in fact the software infrastructure used for the experiments reported in this thesis (Chapter 5).

As we shall see in section 2.3, modern compilers are focused on the two goals described above: *code generation* and *optimisation*. In the following section we will explain the general structure of a modern compiler. We will refer henceforth to a compiler as the application that processes a program written in a high-level language and produces executable machine-dependent code.

2.2 The General Structure of a Compiler

In general, a compiler is composed of three main structures: the *front-end*, the *optimiser* and the *back-end*. The front-end is responsible for taking the source code of a program, analysing it syntactically and semantically and producing an intermediate representation (IR). This IR is taken by the optimiser which attempts to improve the code for a specific objective such as execution time. The output of the optimiser is also an intermediate representation, which is taken by the back-end in order to produce the final object code to be executed on a specific architecture. Figure 2.1 illustrates the general structure of a compiler. These three main components of a compiler are described in more detail in the paragraphs below.

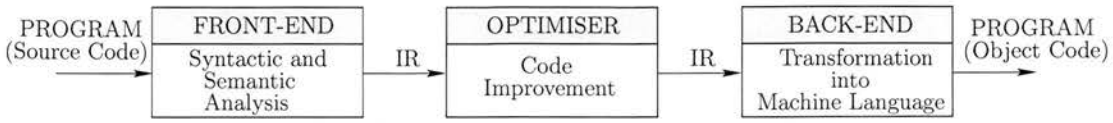

Figure 2.1: The General structure of a compiler.

The Front-End

The front-end of a compiler checks the syntax and the semantics of a program and transforms the program into an intermediate representation (IR). The syntax analysis (done by *scanning* and *parsing*) is concerned with the validity of the program in terms of well-formed sentences. For example, one task that a *front-end* may execute is checking that a variable used in a computation has been declared before. The semantic analysis (also called *context sensitive analysis*) is concerned with the validity of the program in terms of the language specification. In other words, the semantic analysis validates that the sentences of the program are indeed specific instances of the language grammar in which the input program is written.

The Optimiser

The optimiser takes the IR of the program output by the front-end and attempts to improve the code by applying several transformations. The optimiser carries out tasks such as *dependency analysis* and *data-flow analysis* in order to ensure that the changes to be introduced are correct so that the transformed code maintains the meaning of the original code. The output of the optimiser is an IR of the program, for which the final code generated is expected to have "good" performance in terms of a specific objective such as execution time. Since compiler optimisation is the focus of this thesis, this topic will be discussed in more detail in section 2.4.

The Back-End

The back-end of a compiler is responsible for the translation of the IR of the program output by the optimiser into a machine-dependent code that can actually be executed on a specific architecture. In order to achieve this the back-end carries out several tasks. Three of the most significant include: *instruction selection*, *register allocation* and *instruction scheduling*.

Instruction selection is the task of selecting the machine instructions (or operations) that implement the intermediate representation given by the optimiser. It is assumed during this task that there is an unlimited number of resources and, therefore, no restrictions on the use of registers are imposed.

Subsequently, the register allocator needs to assign the set of values produced by the instruction selector to the actual registers of the architecture. The problem of allocating a bounded number of registers in order to minimise the number of loads and stores is, in general,

NP-complete.

Finally, instruction scheduling is the problem of re-ordering the instructions so that they are executed as fast as possible on the target architecture. This step is required because, in general, different operations take different times to execute and also because modern processors can start executing a new operation while others are being executed, as long as the operators of the new instruction are readily available. The minimisation of the latencies (or stalls) introduced by the sequential execution of instructions is, in general, an NP-complete problem.

It is necessary to remark that although the tasks performed by a compiler have been described as separate processes, there are strong interactions between these tasks. For example, the scheduling of instructions may increase the demand for registers (i.e. register pressure). Thus, in addition to dealing with NP-complete problems, a compiler is forced to take decisions that are only an approximate solution to the problem of generating "optimal" code for a particular architecture.

2.3 Goals of a Modern Compiler

There has been a lot of work for many years at the front-end of the compiler on tasks such as scanning and parsing. Thus, the focus of modern compilers has been transferred to the problem of producing code that can maximally exploit the resources of a target architecture. In other words, modern compilers are mostly focused on the problems of *optimisation* and *code generation*. Obviously, when focusing on these tasks, a compiler must ensure that the main principle of compilation is not violated, i.e. that the meaning of the original program is preserved.

In this thesis we focus on the optimisation stage of a compiler and we approach this problem with machine learning techniques. However, we emphasise that tasks that are performed during code generation, such as register allocation and instruction scheduling, are also crucial in achieving good performance on current architectures.

2.4 Compiler Optimisation

In this section we describe the problem of compiler optimisation seen as the task of finding a version of a program that improves the ultimate code generated by the compiler with respect to some specific objective such as execution time, energy consumption or code size. Although in some scenarios one should consider a trade-off between these three different objectives, in this thesis we will tackle the problem of optimising a program for speed (i.e. execution time).

One can obtain different versions of a program by applying several *code transformations*. As we shall see below, a program transformation can be beneficial for some code and detri-

mental for others. Furthermore, code transformations interact with each other in very complex ways. Therefore, in this thesis we approach the problem of compiler optimisation by focusing on the *task of finding a "good" sequence of program transformations that improve the ultimate code generated by the compiler.* However, the reader should bear in mind that, as pointed out in section 1.1, the compiler optimisation problem does not correspond solely to the task of determining good sequences of transformations for programs but also to the development of new code transformations and analyses that enable the applicability of program transformations.

2.5 Program Transformations

The key issue in compiler optimisation is the concept of a program transformation. We can think of a program transformation as a process that changes the code of a program with the hope that the resulting code is in some way *better* than the original one. As mentioned above, in our specific case, *better* means that the ultimate code generated by the compiler runs *faster* than the original one. We will refer sometimes to a program transformation as a *program optimisation* or simply as an optimisation.

2.5.1 Legality, Improvement and Interactions

A program transformation must always preserve the meaning of the original code. If this is not the case, a transformation must not be applied. This is commonly known as the *safety* or *legality* of a program transformation. As described in Cooper et al. (2002, Chapter 8), a lot of effort during the optimisation phase of a compiler is devoted to the analysis of a transformation's legality. A suitable definition for legality is given in Bacon et al. (1994):

> "A transformation is legal if the original and transformed programs produce exactly the same output for identical executions."

A transformation is applied to a code with the hope of improving its run-time behaviour. For example, some transformations are designed to enhance instruction level parallelism (i.e. executing different instructions at the same time). Other transformations are expected to improve memory accesses or to eliminate redundant computations. However, a program transformation does not always provide an improvement in performance. Indeed, the effect of a transformation on the run-time behaviour of a program depends upon the code being transformed and the target architecture. In general, it is very difficult to ascertain when a program transformation should be applied. Additionally, many program transformations can be parameterised and setting the right values for their parameters is also a hurdle.

The problem of deciding when and how to apply a sequence of transformations is made even harder by realising that these transformations can enable or disable the applicability of

other transformations. Furthermore, a program transformation can make another transformation more or less effective. This is known in the compiler literature as the problem of *interactions*. As pointed out by Kulkarni et al. (2006), most interactions are very difficult to predict since they depend on "the program being compiled, the underlying architecture and the specific implementation of the compiler".

Thus, determining a good set of transformations, their parameters and the order in which they should be applied to a program is an interesting challenge for compiler writers and researchers.

2.5.2 Scope

The scope of a program transformation is the level of *granularity* at which it can be applied. For example, a transformation can be applied to statements, basic blocks, innermost loops, loops, functions (or procedures) or to the whole program. However, we will normally refer to the optimisation scope as the level of granularity at which a sequence of transformations is applied. In other words, the optimisation scope is the level of granularity for which one considers different optimisation strategies. For example, a program-level of granularity means that a different optimisation strategy (i.e. a different sequence of transformations) is used for different programs but the same strategy is used within the whole program. In this thesis we only consider a program-level optimisation strategy. However, as will be discussed in section 9.2.2, transformations applied at a program level may lead to inferior performance compared to the case when these transformations are applied at finer levels of granularity such as functions or loops. See section 9.2.2 for more details on the advantages and difficulties of approaches using finer levels of granularity.

2.5.3 Classification

A general classification of program transformations distinguishes those transformations that are *machine dependent* from those that are *machine independent*. Unlike machine-dependent transformations that consider the details of the target architecture, a machine-independent transformation neglects these details. Examples of machine-dependent transformations are instruction scheduling and register allocation. Examples of machine-independent transformations are loop unrolling, constant propagation and common subexpression elimination.

2.5.4 Examples of Program Transformations

Numerous transformations have been proposed in the literature with the aim of improving some characteristics of the code. For example, while some program transformations are expected to reduce the loop overhead, others aim at increasing instruction level parallelism, enhancing data

Type	Transformation
Data-flow loop transformations	Induction variable elimination
	Loop-invariant code motion
	Loop unswitching
Loop reordering	Loop interchange
	Loop tiling
	Loop fusion
Loop restructuring	Loop unrolling
	Loop normalisation
	Loop peeling
Memory access transformations	Array padding
	Scalar expansion
	Array contraction
	Scalar replacement
Partial evaluation	Constant propagation
	Copy propagation
	Strength reduction
Redundancy elimination	Common subexpression elimination
	Useless code elimination
	Dead variable elimination
Procedure call transformations	Procedure inlining
	Procedure cloning
	Loop pushing

Table 2.1: Examples of common program transformations (from Bacon et al., 1994).

cache locality, reducing redundancy in the computations or improving memory access. Table 2.1 presents a classification of some common program transformations as proposed in Bacon et al. (1994).

Common Subexpression Elimination

In order to illustrate the effect of a program transformation let us take for example *common subexpression elimination (cse)*. This transformation detects identical subexpressions at different locations, computes the value of the sub-expression once, stores it and reuses the stored value. An example of a code fragment written in C and transformed by common expression elimination is given in Figure 2.2.

It is clear that common subexpression elimination avoids redundant computations and,

```
                          tmp = i*4;
a = i*4*j;                a = tmp*j;
b = i*4*k;                b = tmp*k;
c = a+b;                  c = a+b;
```

(a) Original code. (b) Transformed code.

Figure 2.2: An example of a code fragment transformed by common subexpression elimination.

therefore, one could speculate that it should always be applied to a program. However, the effect of this transformation can be detrimental if the cost of storing the temporary variables is greater than the benefits obtained by reducing the computations, for example when additional spills to memory occur.

Loop Unrolling

Another classic example of the effect of a program transformation is the application of *loop unrolling*. Loop unrolling is a very simple transformation that replicates the loop body a certain number of times u, usually called the *unroll factor*. The iteration step of the loop is then modified accordingly and a prologue or an epilogue may be added before or after the loop in order to deal with the left-over operations (when the actual number of iterations is unknown). An example of a loop transformed by unrolling is given in Figure 2.3. We note that a prologue or an epilogue has not been added given that there are not left-over operations.

Unrolling a loop has the potential of increasing instruction level parallelism (ILP), reducing the overhead due to loop control and enabling the applicability of other transformations. However, loop unrolling can be detrimental due to a potential degradation of the instruction cache because of the increase in the size of the loop body.

As common subexpression elimination and loop unrolling, there are many other code transformations that can be applied to a program with the goal of improving the performance of the ultimate code generated by the compiler. However, it is difficult to ascertain when and how these transformations should be applied. Additionally, as mentioned in section 2.5.1, the interactions between program transformations make the problem of finding a "good" transformation sequence for a given program even harder.

The compiler community has been aware of this problem and several approaches have been proposed in the literature in order to find "good" compiler transformations for programs (see Chapter 4 for an overview of the related literature). A popular approach that has received a special interest in the embedded systems community is *iterative compilation*.

```
for (i=0; i<100; i++)
  a[i] = b[i] + c[i];
```

(a) Original loop.

```
for (i=0; i<100; i+=2){
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
 }
```

(b) Transformed loop.

Figure 2.3: An example of a loop unrolled twice.

2.6 Iterative Compilation

In this thesis we will refer to iterative compilation as the general process of searching for "good" compiler transformations (or transformation sequences) using an iterative approach such as uniform search. This simple technique has been shown to provide significant improvements in performance, for example, over static heuristics that aim at determining a good set of transformations for a program (see for example Kisuki et al., 2000; Knijnenburg et al., 2002; Fursin et al., 2002; Fursin, 2004). However, the improvements obtained are usually at the expense of a large number of evaluations. Thus, even in an embedded systems scenario, techniques such as the ones proposed in Chapters 7 and 8 are required in order to make iterative compilation useful in practice.

So far we have described the problem of compiler optimisation as the task of finding a transformation sequence that improves the performance of a program. However, we have yet to define a measure of the improvement of a program under the application of a sequence of code transformations. In this thesis we will use a measure of performance known in the compiler literature as the *speed-up*.

2.7 Performance Measure: Speed-up

In order to evaluate the quality of a transformation sequence we will use the speed-up (y) as a measure of performance:

$$y = \frac{\text{time}(T, \emptyset)}{\text{time}(T, \mathbf{x})}, \tag{2.1}$$

where $\text{time}(T, \emptyset)$ is the execution time of program T when no transformations are applied (the baseline) and $\text{time}(T, \mathbf{x})$ is the execution time of the program when a transformation sequence $\mathbf{x}$ is applied. Note that this measure of performance ranges in the interval $(0, \infty)$, where a number between zero and one means that a transformation sequence slows down the execution of the program and a speed-up greater than one indicates an improvement in performance. In practice, however, we can consider speed-ups greater than 1.05 as significant improvements and speed-ups close to 2 as excellent improvements since this means that the execution time has been reduced to half of the original program's. In order to illustrate this point, we can highlight some

of the improvements obtained in the compiler optimisation literature, for example Franke et al. (2005) obtain an average speed-up of 1.71 across three different platforms and 13 benchmarks; Almagor et al. (2004) report speed-ups between 1.17 and 1.33 across 10 programs at the cost of 200 to 4550 program evaluations in a optimisation space of 16^{10} transformation sequences; and Triantafyllis et al. (2005) report speed-ups between 1.05 and 1.4 on 16 SPEC 2000 benchmarks when using their optimisation space exploration (OSE) framework.

2.8 Summary

This chapter has presented a background in compilation and compiler optimisation that may be required in order to understand subsequent chapters. In particular, the basic notion of *compilation* as the process of translating a source program written in a high-level programming language into an object code that can be executed on a target architecture has been given. Additionally, the general structure of a modern compiler composed of a *front-end*, an *optimiser* and a *back-end* has been explained. The problem of *compiler optimisation* as the task of finding a sequence of *code transformations* has been described and the key role of a code transformation in optimisation has been emphasised.

Several examples of program transformations have been provided. In particular, it has been explained that common transformations such as *common subexpression elimination* and *loop unrolling* may be beneficial or detrimental depending upon the characteristics of the code and the target architecture. The difficulty of determining when a program transformation is beneficial and the interactions between different program transformations make *compiler optimisation* a very interesting problem for compiler writers and researchers.

This chapter has also described the popular approach of iteratively searching the space of code transformations in order to optimise a program, which is known as *iterative compilation*.

Finally, the performance speed-up of a program has been defined. This measure will be used throughout this thesis in order to evaluate the performance of a program under the application of a transformation sequence.

Chapter 3 will present a machine learning approach to the compiler optimisation problem, which will be the basis for the techniques proposed in chapters 7 and 8.

Chapter 3

A Machine Learning Approach to Compiler Optimisation

We have seen in Chapter 2 that the compiler optimisation problem can be considered as the task of making a compiler produce *better code*[1], which can be achieved with the application of sequences of *program transformations*. One can think of a program transformation as a process that changes (or transforms) the code of a program (commonly at an intermediate representation) with the hope that the resulting code, for example, runs faster than (and maintains the meaning of) the original version of the program.

Although numerous program transformations have been proposed with the aim of helping compilers to produce better code, selecting the right transformations that should be applied to a program, tweaking their parameters and finding the best order in which these transformations should be applied are usually hand-crafted tasks performed by compiler writers. In fact, for many years compilers have been hand-tuned on a limited number of benchmarks for specific architectures. This approach is difficult to generalise over programs and architectures, requires a lot of time and effort to develop and may sacrifice performance on programs that have not been included in the tuning process.

Therefore, it is necessary to develop new techniques that:

- automatically generate heuristics that indicate when and how code transformations should be applied;
- can be based on a great variety of programs while requiring very little human intervention;
- are able to perform well on programs for which they have not been tuned before; and

[1] Although in this thesis we focus on the problem of optimising a compiler to generate code that runs *fast*, we remind the reader that one may be interested in other objective functions such as code size or energy consumption or even in a trade-off between these different objectives.

- are easily implemented on different platforms.

In summary, it is necessary to develop techniques that maximally exploit the physical resources of an architecture while keeping enough flexibility in order to adapt to different environments.

As mentioned in Chapter 1, the concepts of generalisation, adaptation, and tuning based on previous examples lead to a "natural" solution based on machine learning techniques. This chapter explains in detail how machine learning techniques can be used in order to address the compiler optimisation problem. More specifically, this chapter describes two main areas of research where machine learning can be used within an optimising compiler: *global optimisation* and *predictive modelling*. Furthermore, it is explained how these two different areas can be tackled in a unified framework based upon a *transfer learning* approach.

We start by describing the general framework for an optimising compiler that is driven by machine learning in section 3.1. The problems of global optimisation and predictive modelling are described in sections 3.2 and 3.3 respectively. A transfer learning approach to compiler optimisation that relates global optimisation and predictive modelling is presented in section 3.4. Finally, the chapter is summarised in section 3.5.

3.1 The General Framework for an Optimising Compiler

A machine learning approach to compiler optimisation aims at modelling the relationship between programs and good sequences of transformations as illustrated in Figure 1.1. Given a set of M programs and a set of code transformations $\mathcal{A} = \{a_1, a_2, \ldots, a_{|\mathcal{A}|}\}$ that can be combined into different sequences $\mathbf{x}$ of 'arbitrary' length L, it is possible to execute two different processes: *feature extraction* and *global optimisation*.

The output of feature extraction is a set of features $\mathbf{t}$ that characterise each program. As in any machine learning application, selecting the right features to represent the input is a crucial step given that these features must provide predictive power on the performance of the programs.

The second process, global optimisation, executes a search algorithm over the space of transformation sequences $\mathcal{X}$. Thus, after a certain number of evaluations of a specific program, the optimiser outputs N sequences of transformations ($\mathbf{x}$) along with their corresponding execution times (or speed-ups y)[2].

Afterwards, the features extracted from the programs, the sequences of transformations and their execution times (output by the global optimisation process) are associated and stored in a database. Finally, a *model* $\mathcal{M}(\mathbf{t}, \mathbf{x}, y)$ that aims to capture the knowledge that the features of a program ($\mathbf{t}$) provide about transformation sequences ($\mathbf{x}$) and their performances (y) is

[2]In general, the optimiser can output a different number of transformation sequences per program. Here, for simplicity in the notation, we assume the same number of sequences for all programs.

built. This model must be able to generalise over new programs and can be used to drive the optimisation of programs that have not been seen before.

In the following sections we will focus on the main tasks that can be performed within a machine-learning based optimising compiler: *global optimisation* and *predictive modelling*.

3.2 Global Optimisation

Given an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_{|\mathcal{A}|}\}$ of compiler transformations that can be combined into sequences of arbitrary length, global optimisation aims at finding "good" transformations sequences $\mathbf{x}$, i.e. transformation sequences that improve performance, for a specific program. Thus, global compiler optimisation is a *sequential combinatorial* optimisation problem, where search must be performed on a discrete and unsmooth space ($\mathcal{X}$) of transformation sequences. Clearly, continuous optimisation techniques such as gradient descent cannot be directly applied. Thus, as we will see in section 4.1, popular approaches to the global optimisation problem have been e.g. genetic algorithms and hill climbing methods. An alternative approach, and in fact the one that has motivated the technique described in Chapter 7, is Estimation of Distribution Algorithms, where one "evolves" probability distributions over good solutions. Additionally, if we restrict ourselves to search for "good" settings of a fixed-order set of optimisation flags, statistical techniques such as fractional factorial designs can also be useful (see section 4.1).

It is necessary to emphasise that the problem of effectively sampling the space of transformation sequences in order to achieve "good" performance is very difficult. This is mainly due to the large number of transformations available within a compiler, which may be parameterised and can interact with each other in very complex ways. As an example, neglecting the sequential nature of the problem, let us consider a compiler such as GCC where more than 50 transformations are available. Assuming that there are 50 binary transformations that need to be turned on/off, there are 2^{50} different settings. Clearly, exhaustive enumeration of this space is prohibitive. Additionally, even for smaller spaces such as the one described in section 5.3.2, with a potential number of 14^5 transformation sequences, exhaustive search is impractical and automatic techniques that intelligently search for good optimisation sequences are required.

The importance of *global optimisation* within a machine learning framework is that it is the source of (training) data required by predictive modelling. Additionally, despite being a hard and expensive task, global optimisation is mainly an off-line process and therefore, a machine learning approach to compiler optimisation is not limited to iterative compilation (see section 2.6) but it can also be applied to general purpose compilation.

3.3 Predictive Modelling

Given a database of program features $\mathbf{t}$, sequences of transformations $\mathbf{x}$ and their corresponding speed-ups y, the aim of predictive modelling is to build a model $\mathcal{M}(\mathbf{t}, \mathbf{x}, y)$ that is able to predict "good" transformation sequences when a new program is presented.

One of the major difficulties in predictive modelling, is the determination of a suitable set of features for learning. Although previous approaches have made important contributions to this area (Monsifrot and Bodin, 2001; Monsifrot et al., 2002; Stephenson and Amarasinghe, 2005), there has not been substantial evidence that static features are informative enough about good transformation sequences and/or their performances. As we shall see in chapters 7 and 8, we have relied on the knowledge of compiler experts for the extraction of these features. However, as will be pointed out in section 8.8.3, static features may be insufficient for predictive modelling and features that rely on the dynamic behaviour of the baseline program may be required.

3.3.1 Sequence Prediction

The most straightforward formulation of the predictive modelling problem is as a *classification* task where the input is a set of features that represent a program and the output is the sequence of transformations for which the program experience the best (or a "good") improvement in performance. However, the problem of predicting sequences (that are in general of variable length) is hard and not commonly approached in the machine learning literature. Indeed, most common classification problems involve a well-defined number of classes where the order in which these classes are predicted is irrelevant. Additionally, large amounts of training data may be needed in order to learn such a mapping, especially if long sequences are considered along with a good variety of code transformations.

Other variations to the classification task are possible. For example, the prediction of a probability distribution over "good" transformation sequences (e.g. a Hidden Markov Model). Thus, given set of program features it is possible to predict those sequences that are more likely to improve performance. In fact, this approach is adopted by the technique described in Chapter 7 and referred to as Predictive Search Distributions, where a distribution over good solutions across different optimisation problems is learnt. This distribution is then used in order to focus search on a problem that has not been seen before. This approach can be seen as a smoother formulation of the sequential classification task and it may well be easier to learn a distribution over good solutions than a single best transformation sequence, as the former makes more effective use of the data available for training.

3.3.2 Performance Prediction

Alternatively to modelling "good" transformation sequences directly, an indirect approach can be followed by learning a model of the performance of transformation sequences. In other words, one can adopt a *regression* approach to the optimisation problem (as in the methods proposed in Chapter 8). Optimisation with this approach is performed in two stages. The first stage is the learning stage, where one learns a performance predictor, so called *proxy*, that is able to provide an estimation of the performance of a program under the application of a sequence of transformations. The second stage is the use of such predictors in order to find a good set of transformation sequences. One possible way to achieve this is to optimise the regression function itself by using, for example, gradient-based methods. However, given that the regression function may be rather complicated and that the optimisation space is discrete by definition, optimising the regression function may be very hard.

An alternative way of using proxies for optimisation, and indeed the one followed in this project (see Chapter 8), is to use their predictions in order to rapidly scan a vast number of points in the search space. From these large number of transformation sequences one can select say k data points that are expected to provide best performance as predicted by the proxy. Obviously, making predictions with the proxy must be significantly faster than the actual evaluation of a data point in the search space for this approach to be useful.

Although this is an indirect method, it may be advantageous over the sequence prediction approach as it effectively uses all the data available for learning (i.e. the transformation sequences and their performances). Additionally, the construction of performance predictors is important not only in compiler optimisation but also in architecture simulation, where very slow simulators are used in order to evaluate the performance of a program in a new architecture (see e.g. İpek et al., 2006, as a recent reference).

3.4 Learning across Programs: Transfer Learning

The problems of global optimisation and performance prediction can be addressed within a single-benchmark framework. For example, one can search the optimisation space of a given program by using solely a feedback directed optimisation approach, where a set of transformation sequences are evaluated and used in order to drive optimisation towards "good" regions of the space (see e.g. Cooper et al., 1999, 2002; Almagor et al., 2003, 2004; Kulkarni et al., 2003; Triantafyllis et al., 2003, 2005). Similarly, performance estimators can be built on a per-benchmark basis by training regression models on a set of samples that are specific to a program (see e.g. İpek et al., 2006).

However, these approaches do not make use of previously solved problems and searching (or learning) must be started from scratch each time a new program is presented. In this the-

sis we follow a rather different approach by assuming that there is shared information across different programs in terms of the relationship between transformation sequences and their performances. In other words, we follow a *transfer* learning approach where we want to exploit task-relatedness by learning across different programs and using this knowledge in order to drive optimisation on new programs that have not been seen before. The methods proposed in this thesis (and discussed in Chapter 7 and Chapter 8) rely on this assumption and they are shown to provide greater benefits compared to the single-task optimisation (or single-task learning) scenario.

3.5 Summary and Discussion

This chapter has described a machine learning approach to the compiler optimisation problem. In particular, it has presented two different (but related) areas where machine learning can be used in compiler optimisation: *global optimisation* and *predictive modelling*. Global optimisation is the problem of effectively searching for "good" compiler transformation sequences. Predictive modelling aims at building models that capture the correlations between program characteristics, transformation sequences and their performances. Furthermore, predictive modelling can be formulated as a *sequence prediction* problem or as a *performance prediction* task. In the former, the goal is to model directly "good" transformation sequences. In the latter, one first builds a regression model of the performance of programs under the application of transformation sequences and uses this model in order to find transformation sequences that are expected to improve performance.

This thesis puts together the ideas of global optimisation and predictive modelling by using a *transfer learning* approach. In other words, in the methods proposed in this thesis, predictive models are learnt on a set of programs and used in order to search the optimisation space of a program that has not been seen before. Thus, transference is achieved by exploiting the information shared across different programs.

It is necessary to remark that for many applications achieving transference across different tasks is hard and that assuming relatedness in a set of tasks can be detrimental (see e.g. Baxter, 2000; Caruana, 1997). However, for the compiler problems investigated in this thesis, the techniques proposed in Chapter 7 and Chapter 8, which exploit transference across different optimisation problems, yielded (in general) significant improvements in the performance of the programs.

Chapter 4

Related Work

This chapter presents the most relevant work that has used artificial intelligence and/or machine learning techniques in the field of compiler optimisation. Although a comprehensive review of previous work is infeasible, some of the most important approaches that relate to the objectives of this thesis are briefly described and analysed.

As seen in Chapter 3, there are two main topics underlying the problem of compiler optimisation with machine learning: *global optimisation* and *predictive modelling*. While the former focuses on searching the space of compiler transformations the latter uses supervised learning to predict good compiler transformations for programs. The literature related to these two topics is reviewed in section 4.1 and section 4.2. Two additional topics of interest that will be addressed in Chapter 8 and Chapter 6 are: *performance prediction* and *optimisation space characterisation*. The related work on these areas is presented in sections 4.3 and 4.4 respectively. Finally, some previous work on using unsupervised learning methods in compiler optimisation is described in section 4.5.

It is important to highlight that this chapter only describes the previous work on compiler optimisation with machine learning. The (machine learning) literature related to the methods proposed in Chapter 7 and Chapter 8 as general machine learning techniques is analysed within each corresponding chapter.

4.1 Global Optimisation

Previous work on finding an optimal sequence of compiler transformations for a specific program has been based upon different techniques such as iterative compilation (or uniform search); biased sampling methods including genetic algorithms and hill climbers; statistical methods including orthogonal arrays and fractional factorial designs; and static models of performance.

These approaches reveal not only that compiler optimisation is an area of active research but also that searching the optimisation space of compiler transformations can lead to signifi-

cant improvements in the performance of a program. This is surprising considering that most benchmarks reported in the literature have been extensively used for tuning a specific compiler.

The main difference from previous approaches (to the global optimisation problem) with the techniques proposed in this thesis is that the latter exploit *transference* across programs. Indeed, previous work has focused on searching the optimisation space of a single benchmark without using the knowledge gained from the optimisation of previous programs.

The related work on searching the optimisation space that is described below has been divided into two main approaches: those that have used biased random search and those that have used statistical techniques. However, it is necessary to explain first the simplest baseline of uniform random search.

4.1.1 Using Uniform Random Search

The simplest approach to global optimisation is to use iterative compilation (see section 2.6) based upon uniform random search. In this method the search is fully exploratory and there is no bias towards specific regions of the optimisation space in order to exploit the knowledge gained during the search process. This simple technique has been shown to provide significant improvements in performance, for example, over static heuristics that aim at determining a good set of transformations for a program (see for example Kisuki et al., 2000; Knijnenburg et al., 2002; Fursin et al., 2002; Fursin, 2004). However, the improvements obtained are usually at the expense of a large number of evaluations. More critically, this method does not exploit any existing structure in the space that can bias the search towards better optimisation regions.

4.1.2 Using Biased Random Search

One of the earliest contributions to this area has been given by Cooper et al. (1999), who used **genetic algorithms** (GAs) in order to optimise the size of the code produced by a compiler. This task is particularly important in the field of embedded systems, where one can afford long compilation times with the aim of producing small enough programs that, for example, fit into a read-only memory (ROM).

They implemented a GA that searched the space of compiler transformation sequences in order to produce the smallest version of a program. They worked with 10 program transformations that could be combined into sequences of length 12. These transformations were applied at an 'ILOC' level, which is an intermediate representation (IR) of a program. The GA generated different populations of transformation sequences (chromosomes) of length 12, which were evaluated according to the number of static ILOC operations (fitness value) of the transformed program. Ties were resolved by using the number of dynamic ILOC operations, which is a very crude measure of run-time performance.

The results reported on 8 Fortran programs and 6 C benchmarks showed code-size improvements of 20% to 75% and run-time improvements of 20% to 83%. Sometimes the resulting optimised programs were faster than a hand-tuned sequence and the GA was shown to converge to good solutions faster than uniform search. This latter result is important as it shows that there was some structure in the space of the transformation sequences considered. Interestingly, much shorter sequences than those obtained by the GA were found to have similar performance.

It is necessary to remark that this work was one of the earliest attempts to use artificial intelligence techniques in order to search the space of compiler transformations. However, given that GAs can take very long time to converge to a solution, it is unclear whether this approach can be applied to other scenarios besides embedded systems. Unfortunately, the experiments were not executed on an actual embedded architecture but only on a simulator. Furthermore, since different operations take different times to execute, the number of dynamic operations used as the secondary fitness function (for breaking ties) is not really a suitable measure of performance as it does not necessarily correlate with the actual execution time of a program.

The technique proposed in Cooper et al. (1999) is used as the basis of a prototype of an adaptive optimising compiler (Cooper et al., 2002). This prototype utilises biased random search algorithms such as **GAs** or **hill climbing** methods in order to find compilation sequences that optimise an objective function. This objective function can be for example, run time, code size or even power consumption. Thus, given a program, a target architecture, a set of transformations and an objective function, the compiler selects transformation sequences to be applied to the program and evaluates the objective function on the target architecture. These function values are then input to the so-called "steering algorithm", which selects new sequences (by using e.g. GAs) and applies them to the program. The process is repeated until some desired level of performance is achieved or until some constraints are satisfied.

Based upon this prototype of an adaptive compiler, Almagor et al. (2003, 2004) carried out a set of experiments that aimed at providing a better understanding of the search space and at assessing the performance of different search algorithms on the global optimisation problem. Two sets of experiments were performed: exhaustive experiments and exploratory experiments. The exhaustive experiments involved a set of 5 transformations combined into sequences of length 10, which were applied to two small benchmarks. The exploratory experiments included 16 different transformations forming sequences of length 10, which were applied to 10 programs. Their customised compiler was used as the base infrastructure for the experiments and the programs were executed on a simulated RISC architecture.

As a result of their analysis of the exhaustive experiments, two main conclusions were highlighted: firstly, that the space was not smooth nor continuous; and secondly, that there

were many local optima[1] in the space that were close to any randomly chosen point. The first conclusion does not bring any new information about the optimisation space. Furthermore, questions about the smoothness or continuity of the space are non-sensical given the discrete nature of the space. The second conclusion is interesting as it provides an indication that random biased sampling methods such as hill climbing with different starting points may perform well, given that most local optima in one of the spaces analysed were within 5% of the global optimum. Unfortunately, these results are very dependent on the definition of a local optimum and also on the very simple programs used for the experiments. The results for the exploratory experiments showed that GAs can perform slightly better than other techniques such as hill climbing or greedy search but at a very high cost in terms of the number of program evaluations.

There are several caveats about this work. In the first place, the programs were executed on a simulated RISC architecture instead of on a real embedded processor, which was the main motivation for such an approach. Additionally, as we shall see in Chapter 6, the effect of interactions among program transformations is a key issue in the characterisation of the compiler optimisation space. However, no analysis of these interactions was given. Finally, as in Cooper et al. (1999), the number of instructions executed was used as the measure of performance. As explained above, there is no direct correlation between this measure and the execution time, as different instructions may take different times to execute.

With a slightly different purpose but also based on the use of **genetic algorithms**, Kulkarni et al. (2003) developed additional modules for the VISTA framework. This framework is an interactive compilation software that allows the user to find good compiler transformation sequences for programs. The system asks the user to provide the optimisation phases[2] that should be tried on a program and it returns feedback with static and dynamic information about its performance. Therefore, in an interactive and iterative process involving expert-knowledge the user can find an effective optimisation sequence for a program. The software supports low-level transformations and the user can specify his own transformations. Additionally, the framework can automatically perform an exhaustive search for the best sequence in a small space; find the best permutation of transformations; and use GAs to search a bigger space. The system can optimise a program for code size and run time.

Unlike Cooper et al. (1999), the search for good transformation sequences in Kulkarni et al. (2003) is performed at a *function level* instead of at a program level. As expected, it is found that good sequences differ considerably within the same program. Besides this fact, it is not clear whether this approach really outperforms the work in Cooper et al. (1999, 2002) since

[1] They define a local optimum as a data point where all sequences that differ from it in one position have equal or worse performance.

[2] In general, an optimisation phase may involve the application of several program transformations. However, we will usually refer to phases as if they were single transformations.

their results are not comparable due to different optimisation specifics. As in Cooper et al. (1999, 2002), the experiments were not executed on a real embedded system and a systematic approach to tune the different parameters of the GA was not followed.

Triantafyllis et al. (2003, 2005) propose compiler "Optimization-Space Exploration" (OSE) as a technique to improve iterative compilation. This technique includes the **pruning of configurations** and the use of **performance estimators**.

A compiler configuration is simply a full assignment of parameters to compiler options. Pruning can be performed by using predictive heuristics that indicate for example, when a transformation is likely to be effective after others have been effective, and more importantly, by selecting a good set of k configurations.

The algorithm proposed starts with a seed of configurations and generates new configurations differing from one of the seeds in only one parameter. This is called the "expansion step". Afterwards, it determines the best k-element subset in the "selection step" and uses it as the new seed for the next iteration. The process is repeated until no improvement is observed or until some constraints (e.g. time) have been satisfied. Not surprisingly, the algorithm is very sensitive to different initialisations. An additional form of pruning is applied by organising the configurations in a tree-structure, which allows the selection of what should be tried next and what should be discarded.

The performance estimator of Triantafyllis et al. (2003, 2005) is specific to the Itanium architecture. Such estimator scores *segments of code* based on an analytical expression that involves indicators for ideal cycle counts, data cache performance, instruction cache performance and branch misprediction. Their results show a 5.3% of improvement on the training set (SPEC2000 benchmarks) and a 10% of overall improvement on the test set (SPEC95 and MediaBench benchmarks).

It is necessary to remark that the search method proposed in Triantafyllis et al. (2003, 2005) is simply a **hill-climber** (or local search algorithm) where the neighbourhood of a configuration is defined as all the configurations that differ from it in only one parameter. However, the idea of clustering in the transformation space is interesting. Additionally, as mentioned above, the performance estimator is very specific and therefore difficult to generalise to other architectures.

In a more restrictive approach, Stephenson et al. (2003) searched the space of *priority functions*. A priority function evaluates the importance of different parameters within a specific compiler heuristic and directly affects the efficacy of the heuristic. Their experiments included three different heuristics: hyper-block formation, register allocation and data prefetching; 20 benchmarks drawn from SPECfp, SPECint, MediaBench and other suites; Trimaran and the ORC compiler. **Genetic programming** was used to search the space of priority functions corresponding to each heuristic in order to find a solution on a specific program and a solution

on a set of different programs. The maximum average speed-up reported was 23% obtained with hyperblock formation on the training set and 9% when using cross-validation.

The greatest limitation of this work is the very long time taken by genetic programming to come up with a good priority function. Indeed, there is no clear evidence that the method outperforms random guessing for this specific problem. Furthermore, the performance obtained in a different set of programs demonstrates that *overfitting* is indeed an issue for this technique.

4.1.3 Using Statistical Techniques

A rather different approach to the global optimisation problem in compilers can be followed by using statistical techniques. An early work on this area was done by Chow and Wu (1999), who used **fractional factorial designs** (FFD), a technique from the *Design of Experiments (DOE)*, in order to trim down the search space of different compiler options.

Their main goal was to determine the best setting for a set of binary compiler options while using a reduced number of evaluations from a potentially large search space. These evaluations (or runs) where obtained by the FFD methodology. In addition to "efficiently" searching the space, the FFD technique allows the identification of interactions between the variables involved in the design, which in this case were the compiler transformation flags.

Starting with a small number of runs and iteratively adding more experiments in order to break ambiguities (or aliases) between different factors[3], they studied 9 different optimisation flags for the IA-64 micro-architecture simulator on one SPecInt95 benchmark: *compress*. The complete process involves the identification of interesting compiler switches, the generation of an initial design, the creation of different program versions corresponding to the settings of the design, the measurement of the performance of these settings, the analysis of the results and the resolution of ambiguities by adding more experiments. This is repeated until certain improvement has been achieved. Finally, the setting with the best performance throughout is selected.

The results suggested that from the 9 transformations used, only 6 flags should be turned on. Additionally, if a full factorial design (considering all the runs) had been applied to the final 6 factors, the performance obtained would have been similar. Finally, if no interactions had been considered, a decrease of 2% in the performance achieved would have occurred.

The adoption of a principled way to search the compiler optimisation space is worth highlighting as well as the analysis of these techniques in order to address a relevant compiler problem such as the identification of interactions between program transformations. However, although the factors under the constructed model were analysed in terms of their significance, the assumptions of the model itself were not validated. Indeed, as expressed in the original

[3]In fact, the generators and the aliasing structure used show that the initial design was a FFD of resolution *IV* (see e.g. Montgomery, 1997, pages 420–421). This means that aliasing between main factors and 3-factor interactions was allowed as well as aliasing between some 2-factor interactions and 2-factor interactions.

paper, the model fitted to the data is a linear-in-the-parameters model (see section 6.4 for an explanation of these models). There are several ways to evaluate the departures from this model. For example, a simple coefficient of determination that indicates how much variance is explained by the model could have been used. Additional weaknesses of this work are (from a compiler perspective): the relatively small search space investigated, the specificity of the results to only one benchmark and the marginal performance improvements obtained.

Using a similar technique, namely **orthogonal arrays**[4] (also from from the Design of Experiments (DOE)), Pinkers et al. (2004a,b) developed an iterative procedure to set up different compiler options. The process starts with a specific orthogonal array for which all the rows (settings) are executed. The relative effect of each option is evaluated according to its contribution to improvement in performance. If this contribution is over a certain threshold the corresponding option is turned on, otherwise the option is turned off. With the remaining options (discarding those already selected) a new orthogonal array is constructed and the process is repeated until all the final options are chosen.

The experiments were performed using GCC on a SimpleScalar simulator, 6 SPEC benchmarks and 19 compiler options grouped into 14 factors. The main difference with Chow and Wu (1999) is that Pinkers et al. (2004a,b) only consider the main-effect factors and neglect the interaction effects. Additionally, although the best improvements obtained outperform all the optimisation levels of the compiler, such improvements seem to be only marginal.

In an extension of the work in Pinkers et al. (2004a,b), Haneda et al. (2005b) address the slightly different problem of finding the best compiler setting for a collection of applications. This problem arises in supercomputing centres that continuously deal with a set of applications belonging to a specific domain.

In this approach, orthogonal arrays are used to find a good "representative" subset of the optimisation space. Subsequently, the execution times of this subspace are measured and utilised to approximate the effect of future compiler settings. More specifically, the search algorithm is divided into three steps: the first step takes the representative subset of the space given by an orthogonal array and determines those compiler options that have a large effect as well as those pairs that constructively interact; the second step greedily adds single optimisations to the previously formed sets while keeping those optimisations that do not have a negative interaction; finally, a small number of settings are tested (those with the largest number of options turned on) and the one with the best performance is selected.

Their experimental set-up included 50 optimisations from the GCC compiler, 7 benchmarks from SPECint95, and a Pentium IV architecture running at 2.8GHz. The orthogonal array used was given by a 400×50 matrix, where the columns correspond to the 50 compiler options

[4] An orthogonal array is an array in which all the possible patterns found by selecting k columns appear equally often (see e.g. Hedayat et al., 1999, page 2, for a formal definition). The parameter k is commonly known as the strength of the orthogonal array.

(flags) used and the rows correspond to the different settings of these options.

Unlike Pinkers et al. (2004a,b), Haneda et al. (2005b) aimed at modelling the effect of interactions between program transformations. However, it is unclear how effectively an orthogonal array involving 400 runs samples a space of $2^{50} \approx 10^{15}$ possible settings. Indeed, in addition to the fact that the strength of the orthogonal array used was not mentioned, a linear model that included main factors and 2-factor interactions would need to estimate 1275 parameters, which is much greater than the number of rows of the orthogonal array used. Additionally, there is a lot of criticism against using this methodology in the Design of Experiments literature (see for example Montgomery, 1997, pages 630–634). For example, the *aliasing* structure in orthogonal arrays may be quite complex and there may be cleaner and more effective alternative designs based on *fractional factorial designs*. A final drawback of this approach is the sensitivity of the method to several thresholds, which need to be specified beforehand.

In an additional extension to the work by Pinkers et al. (2004a,b) and Haneda et al. (2005b), Haneda et al. (2005a) use the **Mann-Whitney** statistical test in order to find a good set of compiler optimisation options by evaluating their positive or negative effect on the performance of a program. More specifically, their greedy algorithm takes an initial sample drawn from an orthogonal array and iteratively turns on/off optimisation flags by determining if their effect is significant (using the Mann-Whitney test) and by assessing if such effect is positive or negative.

Their experiments were executed using the GCC compiler, 10 programs from the SPEC2000 benchmark suite and a Pentium IV architecture. Their results show that, in general, their technique outperforms the standard -O*x* GCC flags. As in Haneda et al. (2005b), the efficacy of their method relies upon several threshold parameters, which are set up heuristically. Additionally, as each option is evaluated independently, the technique ignores the effect of interactions among optimisation flags.

Unlike Chow and Wu (1999); Pinkers et al. (2004a,b); Haneda et al. (2005b), Vuduc et al. (2004) propose a statistical-based approach to the global compiler optimisation problem that does not rely on the use of a Design-of-Experiments methodology. In particular, they develop a method that stops a potentially prohibitive exhaustive compilation search early. Additionally, they propose a technique that automatically selects a version (or implementation) of a program given a specific input data. The first problem is known as the **early stopping problem** and the second task is referred as the **run-time implementation problem**. The experiments are focused on a dense matrix multiplication kernel using the space of register and instruction-level optimisations.

Their analysis of the data shows that the search space changes from architecture to architecture and more importantly, that finding the best implementation of a program (even in the case of a simple kernel) is a "needle-in-a-haystack" problem.

The early stopping problem is the task of finding when to stop the search of a particular

implementation of a program (due to specific compiler settings) so that the performance obtained lies within some fraction of the optimal solution. This task is formulated as the problem of finding the probability of the maximum performance achieved at time t being over a certain threshold. If this probability is sufficiently high the search process can be stopped.

More formally, using the same notation as Vuduc et al. (2004), we wish to determine the value of :

$$P(M_t > 1-\varepsilon) > 1-\alpha, \tag{4.1}$$

where M_t is the random variable corresponding to the maximum performance obtained until time t, ε is a certain fraction of the optimal solution and α is a degree of uncertainty. Note that the probability statement in equation (4.1) is with respect to the performances obtained in the space of all possible implementations of a specific program (given a set of transformations) and that the performance measure is normalised with respect to the current best performance at time t.

Vuduc et al. (2004) argue that equation (4.1) can be estimated by using the **empirical cumulative distribution function** (cdf) of performances observed until time t. Thus, one can specify an ε fraction of the optimal solution, a degree of uncertainty α, and the algorithm will stop the search when these requirements are satisfied.

It is necessary to emphasise that this approach is essentially different from Almagor et al. (2004, 2003); Cooper et al. (2002) since there is no biased sampling involved in this technique. Therefore, as stated in Vuduc et al. (2004), their algorithm may serve as a complementary method to other search approaches. However, one should be particularly careful with the use of the empirical cdf for determining when to stop searching early as the empirical cdf may not be a good estimate of the actual cdf. This is an issue especially for needle-in-a-haystack type problems, where the goodness of the solutions at time t can be overestimated and therefore, the search will be stopped prematurely.

The run-time implementation problem has not been very much studied in the literature from a statistical or machine learning perspective. Vuduc et al. (2004) formulate this problem as a classification task, where one has to select the best implementation of a program given a specific input data. Thus, given input data that is described by some set of features, for example the matrix sizes in a matrix multiplication algorithm, one can try to predict the best implementation of the program (e.g. of matrix multiplication) so that the maximum performance is achieved. This is done by using a set of previously solved problems. Vuduc et al. (2004) built a training data set with three different implementations of dense matrix multiplication varying the sizes of the matrices between 1 and 800. They determined the best (out of 3) implementations for 1936 different inputs. The results were evaluated using 1436 training points and 500 test points. The best performance was obtained by using **support vector machines** and it was reported to have a 12% misclassification rate. This was one of the earliest attempts to learn across different

input data. However, in general, characterising more complex input data with a set of features can be a hurdle.

4.1.4 Other Approaches

Based upon the work in Cooper et al. (1999) and Chow and Wu (1999), Wu et al. (2006a) propose a methodology for "automatic exploration of compiler options". This methodology uses **Genetic algorithms** (GAs) and **fractional factorial designs** (FFDs) to trim down a potentially large space of compiler optimisations on a set of training benchmarks, which are different for each application *domain*. An application domain is a category which a particular benchmark belongs to, for example multimedia applications, network applications, etc. After applying GAs or FFDs one obtains a reduced space of optimisation settings which must be exhaustively enumerated and executed on the target architecture. The results are then stored in their corresponding database. Finally, given a new program, a user specification consisting of priority weights on the optimisation objective (which determine the priorities for optimisation on execution time, power consumption and code size) and an application domain, the compiler searches the stored databases to find the best set of optimisations to be applied.

Wu et al. (2006b) present a practical implementation of the above methodology on an Intel XScale 80200EVB architecture using 4 training programs and 4 test programs from a DSP test suite. Their results show very little improvement on execution time and power consumption, but significant improvements on the code size of the programs.

Their idea of clustering benchmarks in order to customise compiler optimisations is worth highlighting. However, this is done in an *ad hoc* manner. This differs from the approach adopted in the present thesis where features of a program are correlated with "good" compiler optimisations in an automatic and principled way. Additionally, we note that the approach followed by Wu et al. (2006a) makes predictions on a new program based upon a reduced optimisation space, which has been obtained by exploring a larger search space on a set of training programs. This may lead to very sub-optimal solutions.

Pan and Eigenmann (2006) propose the **combined elimination** (CE) algorithm with the goal of determining a "good" set of binary compiler options for a program while requiring a low number of evaluations. The algorithm initially evaluates the baseline setting corresponding to all the compiler optimisations being turned on, and it iteratively removes, i.e. turns off, one optimisation at a time so that the performance of the program is increased.

They investigate 38 GCC compiler options on 23 SPEC CPU2000 benchmarks and a Pentium IV and a SPARC II architectures. It is shown that the CE algorithm outperforms other techniques such as the ones presented in Pinkers et al. (2004a) and Triantafyllis et al. (2005) as it achieves solutions of comparable performance while requiring a significantly lower number of program evaluations.

An obvious limitation of the approach in Pan and Eigenmann (2006) is that it only addresses the problem of tuning a fixed-order set of transformations, i.e. it does not deal with the sequential (or phase-ordering) problem. Additionally, in a very recent approach which was based upon the work presented in Chapter 7, Cavazos et al. (2007) showed that the CE algorithm is outperformed by their technique and also by random uniform search.

Kulkarni et al. (2006) propose an alternative approach to efficiently search for optimisation phase orders of arbitrary length (or optimisation sequences). Rather than directly searching the space of phase orderings, their technique **prunes the space of all possible code versions** that can be obtained with such orderings. This pruning process is based upon two practical facts. Firstly, that there are compiler phases that do not change the code given some current state, i.e. there are idempotent phases (they refer to these phases as dormant phases). Secondly, that several different phase orders can lead to identical code versions even when all the phases are effective (they refer to effective phases as active phases). Thus, pruning can be performed by identifying idempotent transformations and by identifying identical versions of the code.

Clearly, their pruning methods can be used as a way of speeding up the data generation process for the application of techniques such as the ones presented in Chapter 7 and Chapter 8. Indeed, the first type of pruning has been applied during the generation of the small space of the SUIF data set described in section 5.3.2. Additionally, in general, the task of identifying identical code versions is hard. However, Kulkarni et al. (2006) suggest an efficient way of doing this without comparing the complete code generated by the compiler.

Kulkarni et al. (2006)'s experiments are performed at a function level with 15 optimisation phases, sequences of length 12 (on average), on 111 functions belonging to 6 benchmarks from the MiBench suite (Guthaus et al., 2001), using the very portable optimiser (VPO, Benitez and Davidson, 1988) and a StrongARM SA-100 processor running Linux as its operating system. Their exhaustive experiments, which were performed on 98% of all the functions by using the pruning techniques described above, showed an average potential reduction in code size of 37.8%.

Fursin et al. (2005) propose a practical method for *fast evaluation of program optimisations* that, instead of searching the space of optimisations in smarter ways, **scans more points within the same amount of time**. Their method is based on the fact that code sections (e.g. loops) exhibit periods of time where performance is stable. Thus, several optimisations can be evaluated during the same execution of the program every time a specific section is called. Such an approach is implemented by using **code instrumentation and code versioning** in a production compiler. For their experiments, five SpecFP2000 benchmarks were selected; the PathScale EKOPath compiler (PathScale, 2005) was used; and the programs were executed on an Intel Pentium IV architecture. Although their results show speed-up factors of 32 to 962, these speed-ups are very dependent on the behaviour of a particular application. Thus, there

is not much intuition on how this approach will scale to very large search spaces of program optimisations. Chapter 7 describes Predictive Search Distributions as a method that learns a distribution over good solutions across different benchmarks in order to speed up search on unseen problems. This has the advantage over Fursin et al. (2005)'s method in that when a new program is presented, it does not start search from scratch but it uses the knowledge gained from previously solved problems (programs). However, the method presented in Fursin et al. (2005) can be used as a complementary technique to machine learning methods in order to speed-up the data generation process.

In summary, most statistical methods proposed in the literature that attempt to intelligently search the compiler optimisation space have been limited to searching the space of compiler optimisation flags, where the order of the transformations has been fixed beforehand. This is a major limitation in comparison to biased-search algorithms as the order of transformations can play an important role in achieving maximum performance in compiler optimisation. The following section describes some previous work on using supervised learning in order to predict when and how to apply a single compiler transformation and also when/how to apply a set of program transformations.

4.2 Predictive Modelling

Most supervised learning approaches to compiler optimisation have focused on predicting when and how to apply a particular program transformation such as loop unrolling or instruction scheduling. This seems reasonable given the limited knowledge of informative program features for the predictive modelling task. However, this ignores the results from global optimisation indicating that the application of several transformations yields significant improvements in performance. Furthermore, it ignores the fact that the improvement provided by the application of a particular transformation when applied in isolation can be diminished when interacting with other transformations.

This section presents the related work on predictive modelling by dividing it into two approaches: those that attempted to predict a single transformation and those whose aim was to predict a set of transformations.

4.2.1 Predicting a Single Transformation

This section presents the work related to the use of supervised learning in order to predict the effect of a single compiler transformation (or optimisation). As we shall see below, given the importance of loop unrolling in compiler optimisation, there has been an active interest in predicting the effect of this transformation.

Branch Prediction

One of the earliest and successful attempts of using supervised learning in compiler optimisation was pursued by Calder et al. (1997). They addressed the problem of *branch prediction*, i.e. predicting whether a branch within a program is taken. Branch prediction is believed to be of great relevance to optimising compilers as well as to computer architectures. Calder et al. (1997) focused on **static branch prediction**, which (unlike *dynamic branch prediction*) does not require profiling information and is based solely on program structure information. In other words, the goal in static branch prediction is to foretell when a particular branch is taken before it is actually executed.

Their technique was called *Evidence-based static branch prediction (ESP)*, which aimed at predicting the direction of a branch given a set of features that identify the branch. The process starts by determining a set of static features that characterise branches. These features are associated with two dynamic measurements: the execution frequency *(normalised branch weight)* and the percentage of times the branch is taken *(branch probability)*. Once data has been collected over a sufficiently representative number of branches one could use any supervised learning algorithm for this task. Calder et al. (1997) use **artificial neural networks** and **decision trees**. The former was utilised with batch mode training in order to minimise the number of missed branches and the number of branches incorrectly taken. The latter method (decision trees) was employed to solve the binary classification problem of deciding whether a branch is taken.

The experiments were executed on 43 C and Fortran programs from SPEC92 and other suites. The programs were compiled with standard optimisation -O level on a DEC $3000-400$ using the Alpha AXP-21064 processor. The results reported indicate an overall miss rate of 20%, a significant improvement compared to the best existing heuristic at that time which had a 25% overall miss rate. This work can be considered as pioneering in the application of machine learning techniques to compiler optimisation. Indeed, as claimed in the original paper, the method is independent of the language, compiler or target architecture.

Since the publication of ESP (Calder et al., 1997), there has been a lot of interest in the application of machine learning techniques to the branch prediction problem, see e.g. Jiménez (2005); Singer et al. (2007) as recent references. However, branch prediction is a congested area of research and it is unclear if it is possible to outperform the highly tuned current state-of-the-art methods.

Instruction Scheduling

Contemporaneously to the work by Calder et al. (1997), Moss et al. (1998) proposed a machine learning approach to **learning to schedule straight-line code**. Unlike previous methods that used heuristics to schedule code, they formulated the problem of instruction scheduling as a

learning task. Their focus was on scheduling instructions within a basic block, i.e. straight-line code. An additional constraint to make the learning process feasible was to consider only greedy schedulers, in other words, those that from the beginning to the end of the basic block select the apparent best instruction from a set of possible instructions to be scheduled next.

The learning task was defined on the relation over triples (P, I_i, I_j), where P is a partial scheduling, I_i and I_j are instructions considered for scheduling and I is the set of candidate instructions to be scheduled next. A triple was said to belong to the relation if the first instruction is better to be scheduled than the second one. Otherwise, the triple was considered not to be a member of the relation. The search was additionally constrained to good schedules for blocks of 10 or less instructions. Although this may be seen as a very strong assumption, it is shown that the average number of instructions per basic block in their data is 4.9.

Their experiments involved the Digital Alpha 21064, 18 benchmarks from SPEC95 compiled with the vendor's tools and with the highest optimisation level. A total of 447,127 basic blocks with 2,205,466 instructions were collected. Five features characterising the instructions and the partial scheduling were selected and four different learning algorithms were used. Their results used cross-validation and targeted two different measures of performance: the number of times the learned scheduler made an optimal decision and the improvement in execution time as a result of the scheduling process.

Although their algorithm is found to provide better improvements than two production compilers' fixed strategies, it is outperformed by a heuristic greedy scheduler algorithm. Additionally, three possible weaknesses of this work are worth mentioning: the use of a simulator instead of a real processor; the difficulty of generalising this solution to global scheduling, and the considerably long time taken by the learned scheduler to execute. However, from a machine learning perspective, this work represents a great contribution to the field in the sense that it formulates a novel approach to a long-standing compiler optimisation problem.

Cavazos et al. (2004) followed a rather different approach to instruction scheduling with machine learning. Instead of developing an algorithm that can learn how to schedule straight line code, they raised the question if learning machinery can be used to decide **whether to schedule or not**. Their motivation arises from the fact that scheduling does not always provide improvements in performance. Furthermore, they experimentally showed that the improvement obtained from instruction scheduling can be very small compared to the overhead due to the application of the technique.

The formulation of the problem was straightforward: building a classifier able to predict when a basic block benefits from scheduling (which was called by the authors as a "filtering" technique). This is an important issue when dealing with just-in-time (JIT) compilers as an improvement in execution time may be diminished by an increase in the compilation time.

The data generation process involved seven benchmarks from SPECjvm98, *list scheduling*

as the algorithm of choice, and 14 "inexpensive" static features along with an estimate of cost for each basic block given by a simplified machine simulator. The labelling of the data relied on a threshold $t\%$ above which a block was considered to have benefited from the optimisation; blocks for which no improvement was achieved were labelled that they should not be scheduled, and those with an improvement above 0 but less than $t\%$ were discarded. The code was generated using the highest level of optimisation targeting an Apple Macintosh with two 533 MHz G4 processors model 7410.

The efficiency (in compilation time) and effectiveness (in execution time) of the technique when using a **rule set induction classifier** were evaluated across the benchmarks for different values of t. The method was shown to keep 90% of improvement while reducing the scheduling effort to 25% or less.

A major criticism to this work is that a systematic approach for setting the threshold t was not presented. Indeed, considering the sensitivity of the results to this parameter, a very simple methodology such as nested cross-validation could have proved practical for this purpose. Additionally, considering that the heuristic of "always scheduling" gave only 2.3% of overall improvement, the results seem to be practically insignificant. Due to this last difficulty, other benchmarks for which a greater improvement was found were included in the study. Nevertheless, as before, the threshold t was not systematically tuned.

Register Allocation

In Cavazos (2005) the use of classifier systems for instruction scheduling was extended to the construction of heuristics for register allocation. In this case, the task was to select a register allocator from two algorithms: *linear scan* and *graph colouring*. The method was called *hybrid register allocator*. This approach is very similar to the one presented by Cavazos et al. (2004) and indeed, both solutions were grouped into a more general methodology called 'LOCO' standing for "Learning for Optimizing Compilers". As in Cavazos et al. (2004), deciding the best setting for the thresholds involved in the technique was still an issue.

Loop Unrolling

In addition to instruction scheduling and register allocation, there has been a common interest in studying a particular compiler transformation such as loop unrolling. This interest is reasonably motivated by the fact that this transformation has a great impact on the execution time of programs. Additionally, it is an easy-to-apply and always-legal transformation. Indeed, loop unrolling is a very simple but powerful transformation that replicates the body of a loop several times. Although it has been studied for many years (see for example Dongarra and Hinds, 1979), deciding when and how much a loop should be unrolled remains a challenge for compiler writers and researchers. A reason for this is that loop unrolling may provide an

improvement or degradation in performance depending upon different circumstances. For example, it can enhance *instruction level parallelism (ILP)* but it can also degrade the cache due to an increase in the size of the loop body.

Monsifrot et al. (2002) proposed one of the earliest approaches to the development of automatic techniques to predict the applicability of loop unrolling. Their methodology was similar to previous approaches where a set of features characterising an instance, in this case a loop, were utilised to predict a goal: **when to unroll**. However, unlike Calder et al. (1997); Cavazos et al. (2004); Cavazos (2005), the transformation was applied at the source-code level (using the tool set for Fortran programs TSF, Bodin et al., 1998) instead of at the back-end of the compiler.

The set of features used to characterise a loop included memory accesses, arithmetic operations count, the size of the loop body, the presence of control operations and the number of iterations of the loop. Their experiments involved 1036 loops drawn from 20 benchmarks belonging to the SPEC95 suite and some computational kernels; the GNU Fortran Compiler; and two different architectures: an UltraSPARC and an IA-64. Each loop was executed twice on these platforms and categorised into four different groups. Thus, a loop was considered insignificant if its execution time was too small and therefore it was discarded; a loop was not improved if its benefit from unrolling was less than 10%; a loop was said to be improved if it experienced a benefit greater than 10%; and a loop was said to be degraded if its execution time under unrolling was greater than its baseline's. Finally, a feature vector representing a loop was labelled unroll/not unroll by computing a weighted sum of the number of loops that, being represented by such a vector, were degraded, unimproved or improved with unrolling.

Oblique **decision trees** were used for learning and the results reported an overall classification accuracy of 85%. However, the classification rate of positive examples (those loops improved by unrolling) was only 62.4% (on the UltraSPARC). There may be at least two reasons for this low performance. Firstly, their data was *noisy* by construction as loops with the same representation could belong to different classes. In other words, there were data points that, having the same representation, were labelled as improved by unrolling (positive) but also as degraded by the transformation (negative). Secondly, the generated data set was clearly imbalanced and very little was done to overcome this issue. Finally, the task of deciding whether a loop should be unrolled is unrealistic. Indeed, the performance achieved with loop unrolling is strongly dependent on the unroll factor used and transferring this decision to the built-in compiler heuristics leads to very suboptimal solutions.

This latter drawback was to be overcome by Stephenson and Amarasinghe (2004, 2005), who used supervised learning not only to investigate whether unrolling a loop could be beneficial but also to predict how much a loop should be unrolled, i.e. they aimed at predicting **the best unroll factor for loops**. Thus, the binary classification task proposed in Monsifrot et al.

(2002) was generalised to a multi-class problem where the goal was to determine the unroll factor for which a loop experienced its minimum execution time.

The experimental set-up included more than 2,500 loops from 72 benchmarks, the Open Research Compiler (ORC) with -O3 optimisation flag and an Itanium® 2 architecture. Unlike Monsifrot et al. (2002), loop unrolling was applied at the back-end of the compiler; a richer set of 38 features was used; and unroll factors from 1 to 8 were considered.

Nearest neighbours and **support vector machines** were used for learning and their performance was evaluated with a cross-validation procedure. A maximum of 65% of accuracy was obtained and a final overall speed-up of 5% was achieved over the ORC heuristic on the SPEC2000 benchmark suite. This speed-up decreased to 1% when software pipelining was enabled in the compiler.

Despite having a greater number of loops, a more numerous set of features and a more general approach to learning in loop unrolling, their results (as in Monsifrot et al., 2002) were not very encouraging. Certainly, although i) a lot of effort was invested in instrumenting the code; ii) loops with low execution time were discarded; and iii) the variability of the measurements was taken into consideration by calculating the median after 30 runs, very low accuracy was obtained and little improvement was observed with respect to the ORC heuristic.

This raised the question of whether a classification solution to the problem of predicting when and how unrolling should be applied is sufficient to capture the knowledge of the behaviour of loops under this transformation. Certainly, despite having measured the execution times for each loop, neither of these two previous approaches effectively used them. There is a lot to be gained by exploiting as much as possible all the data collected in a machine learning task. Consequently, Bonilla (2004) proposed a more general and smoother solution to learning in loop unrolling by using a **regression approach**.

A regression formulation of the problem aimed at predicting the improvement in performance that a loop could experience under different unroll factors. Therefore, a model $f(\mathbf{x}, u)$ was constructed by Bonilla (2004) in order to predict the speed-up or slow-down of a loop $\mathbf{x}$ when using unroll factor u. Given a new loop, the predicted unroll factor can be determined by by evaluating this function on the different unroll factors considered and selecting the one with the best predicted improvement.

A total of 248 loops drawn from eight benchmarks from the SPECfp95 suite and other variety of vectorial routines were collected. The GNU Fortran compiler with -O2 optimisation level and a dual Intel® XEON™ running at 2.00GHz with 512 KB in cache (level 2) and 4GB in RAM were used. Unroll factors from 1 to 8 were considered.

For learning, 11 static features describing the loops were used and two regression methods were utilised: **multiple linear regression** and **classification and regression trees (CART)**, which were evaluated under a cross-validation methodology. The results reported showed that

for some benchmarks the predictions were relatively accurate but for others the performance was poor. When determining the best factor for loops it was shown that the technique predicted the optimal or near-optimal solution most of the time resulting in a maximum of 18% of re-substitution improvement in performance and 2.5% of overall improvement.

Two major criticisms of this work should be pointed out. Firstly, it seems that 248 loops cannot be representative of a great variety of loops that can be characterised with 11 features. Secondly, and more importantly, the performance measure used to evaluate the technique was the so-called "re-substitution" improvement as real speed-ups obtained when using the predicted unroll factors were not evaluated. In other words, the improvements in performance obtained were computed by using the (original) collected data that ignored the interactions between loops and the effect of the instrumentation. Therefore, the speed-ups reported may well be an upper bound of the real improvements that could be achieved.

With three different and not very successful approaches to learning in loop unrolling it is not clear if the research community should continue focusing on predicting when this transformation should be applied in isolation. Since interactions among transformations do take place, predicting when a set of transformations should be applied seems a more interesting and realistic approach to using predictive modelling in compiler optimisation.

4.2.2 Predicting a Set of Transformations

With the aim of constructing an interactive tool in order to assist a user with finding a good set of optimisations for programs, Monsifrot and Bodin (2001) developed *Computer Aided Hand Tuning (CAHT)*. Their goal was to "learn expert knowledge associated with optimisation and parallelisation techniques". For this purpose, they used **case based reasoning**, an instance-based learning technique very similar to nearest neighbours. By adapting this method to the compiler optimisation problem, the following process was proposed. It starts by *identifying* those fragments of code candidates to be optimised, i.e. loops with significant execution times; afterwards, cases are *retrieved* according to some features of the loop, also called "indices". Once the cases are available, the solution can be *reused* when a new loop is presented. Finally, if new optimisations were applied to a specific loop the user should *retain* this new case. Thus, the cases were pairs composed by a loop represented by its features and a solution specifying the optimisations or parallelisation technique applied.

The experiments were performed on two benchmarks: one loop benchmark consisting of 64 loops and a more realistic Fortran program for Gaussian density computation. The programs were compiled using the -O3 optimisation level and executed on two different architectures: a 4-processor Sun Enterprise 450 and a 4-processor SGI Onyx.

Given the specificity of their results to a very small set of examples and the bias towards good performance implicit in their evaluation, it is unlikely that their results could generalise

to other experimental scenarios. However, it is necessary to highlight their endeavour in presenting a rich set of 31 features involving loop structure, arithmetic expressions, array accesses and data dependencies that can be used by other researchers. Finally, due to the fact that most cases were taken from tuning guides and that there is a lot of intervention from the user in the solution proposed, for example when checking for legality, the ultimate goal of generating automatic heuristics for compiler optimisation is not achieved.

In a very similar approach, Long and O'Boyle (2004) used **nearest neighbour** methods in order to predict good transformations for Java programs. Selecting 14 features from Monsifrot and Bodin (2001) and using the Unified Transformation Framework (UTF, Kelly and Pugh, 1993), programs were classified into categories, a set of transformations was applied to these programs and their performances were recorded.

Their experiments involved 16 programs that were executed on two different platforms running Windows and Linux. The speed-up obtained on Linux was 1.10 compared to 1.14 when applying uniform search with 1000 evaluations. The speed-up obtained on Windows was 1.09 compared to 1.10 when using uniform search. However, an analysis of the performance of the learning algorithm was not presented and therefore, the results may be only an indication that the set of benchmarks used, which was composed mostly by very simple kernels, was very easy to improve.

With the same goal of optimising Java programs, Cavazos and O'Boyle (2006) use **logistic regression** in order to predict good settings for compiler options on *program methods* (i.e. at function level). More specifically, their method learns a logistic regression function for each optimisation option independently, and it uses such functions to make predictions on new methods that have not been seen before.

Their experiments were executed using the Jikes RVM framework (Burke et al., 1999), 7 SPECjvm98 benchmarks and 7 additional benchmarks from other suites, and a Pentium IV architecture. The results were compared with the default baseline of all optimisations being turned on at different optimisation levels O0, O1 and O2. While marginal improvements were achieved at O0 and O1 optimisation levels, the reduction in total time (compilation time plus execution time) was 29%. However, when considering the adaptive scenario, which is the standard model used by the Java JIT compiler, total time reductions of only 1% for the SPECjvm98 and 4% for the rest of the benchmarks were obtained.

A method-specific approach such as the one followed by Cavazos and O'Boyle (2006) seems to be a flexible way of tackling predictive modelling in compiler optimisation. Indeed, rather than predicting the same set of transformations for a complete program, different predictions can be made for different methods. However, this neglects the effect of interactions between different parts of the program. Additionally, Cavazos and O'Boyle (2006)'s approach ignores the effect of interactions between different program transformations as a logistic regres-

sion function is learnt independently for each optimisation flag. Finally, as the experiments were executed on a Pentium IV architecture and the instrumentation was done at a function level, one may expect to observe some variability in the measured execution times. However, this variability was not reported, which makes difficult the assessment of the statistical significance of their results.

Similar to Cavazos and O'Boyle (2006), Wu et al. (2007) follow a fine-level-of-granularity approach with the goal of determining a good set of optimisations for program segments, e.g. loops. Their idea is to create a database of program segments that can be characterised by a set of static features (i.e. "syntax structure") and dynamic features (i.e. "architecture-dependent behaviour"), which are stored with their corresponding set of good compiler transformations. When a new program is presented, it is processed and divided into segments which are matched with the previously stored segments and their corresponding optimisation sets are utilised, i.e. a **nearest-neighbour-like algorithm** is adopted.

Their experiments considered a set of 20 training programs containing a large number of loops, 30 test programs from embedded suites such as MediaBench and DSP kernel, the GCC compiler and an Intel embedded XScale PXA255 architecture. Their results show that from the 30 test programs considered, only 9 benchmarks achieved an improvement over the -O3 default optimisation level. As in Cavazos and O'Boyle (2006), the approach in Wu et al. (2007) neglects the effect of interactions between different code segments, which could be the reason for the modest improvements achieved.

In this section we have described the related work on using supervised learning in compiler optimisation focused on the formulation of the problem as a classification task (although this is not the case for the work presented in Bonilla, 2004). As explained in section 3.3.2, the compiler optimisation problem can be formulated as a performance prediction task by adopting a regression approach. The related work on this area is presented in the following section.

4.3 Performance Prediction

Several authors have addressed the problem of building performance estimators. For example, Karkhanis and Smith (2004) use data dependency information, cache miss rates and branch misprediction rates in order to formulate a *Superscalar Processor Model.* In principle, this model could be utilised to evaluate different program optimisations. However, the process of constructing the model is extremely complex and made in an *ad hoc* manner, which makes it difficult to implement and impractical for the purpose of driving compiler optimisation.

In a simpler approach, Triantafyllis et al. (2003, 2005) develop a **static performance estimator** for the *Itanium architecture* within their compiler Optimisation-Space Exploration (OSE) framework. This estimator provides a relative performance predictor between two code

segments. Their main goal is to be able to discriminate between two code segments by determining which one runs faster. Hence, the accuracy of the predictor is not as relevant as its (relative) discriminatory power.

Such a performance predictor is based on estimators of the ideal cycle count, data cache performance, instruction cache performance and branch misprediction. The general idea of making predictions of relative performance is somewhat similar to the approach presented in Chapter 8, where regression models are learnt in order to predict the performance speed-up of a program relative to its baseline's execution time. Clearly, if our ultimate goal is to find good transformation sequences for programs, there is no need to develop very accurate predictive models as long as these models correctly determine when a version of a program or a code segment executes faster than another one. However, Triantafyllis et al. (2003, 2005) do not make use of the information provided by other programs' behaviour, i.e. there is no notion of learning (or *transference*) across different programs. In fact, the models proposed in Triantafyllis et al. (2003, 2005) are based upon analytic expressions that are architecture-specific and therefore difficult to generalise to new and more complex architectures.

Zhao et al. (2003) also propose an **analytical approach** to predicting program performance. Their general framework combines optimisation models, code models and resource models in order to predict the *impact on cache performance for embedded processors*, which is measured as an estimate of the number of cache misses. Loop optimisations for data locality were used. The optimisation model includes the characteristics of an optimisation (code transformation) and their impact on a specific objective (e.g. execution time); the resource model characterises the target machine configuration, i.e. the cache; and the code model is an abstraction of the application code. These models are integrated in order to predict a so-called "benefit" value, i.e. the benefit of applying an optimisation. As in Triantafyllis et al. (2003, 2005), this approach is architecture-specific and it relies on a specific cache configuration, which diminishes its generalisation power.

İpek et al. (2006) proposed a rather different approach to modelling performance. Their goal is to speed-up architecture simulation times by "efficiently exploring the architectural design spaces". They used **artificial neural networks** (ANN) in order to predict the performance of a target architecture based on several hardware parameters such as cache size and memory latency. Their results show that by training on less than 2% of a design space, their predictions achieve less than 2% error. One of the major differences between the work in İpek et al. (2006) and the technique proposed in Chapter 8 is that the former builds a different model for each benchmark, i.e. no transference across benchmarks occurs. Additionally, such an approach is not suitable to model the space of compiler transformations due to its specificity to a program binary, as a modification resulting from the application of a program transformation would require a re-training of the model on a new set of simulation samples.

4.4 Optimisation Space Characterisation

It has been claimed in the compiler literature for many years that the effect of interactions between program transformations represents one of the major issues in compiler optimisation. However, very little work has been done in order to provide a qualitative and quantitative characterisation of such interactions. Indeed, although Pinkers et al. (2004a,b) make use of statistical models such as orthogonal arrays in order to trim down the search space of compiler optimisations, an analysis of the interactions between transformations is not presented. Additionally, as we have described in section 4.1.3, Chow and Wu (1999) present a quantification of the interactions between some program transformations for a fixed set of optimisation flags. However, the goodness of fit of their model is not evaluated.

From a rather different perspective, Lee et al. (2004, 2006) present an empirical study of the impact of different optimisations when they are applied in isolation and when they are applied in conjunction with other subsets of optimisations. Their experiments were executed within the Jikes RVM framework (Burke et al., 1999) using 20 Java optimisations and 9 benchmarks (from SPECjvm98, SPECjbb2000 and an XML database) on an IA32 platform and a PowerPC architecture.

It is shown in Lee et al. (2004, 2006) that there was very little interaction between the program optimisations used. However, the experiments underlying this study were very restrictive as only interactions of transformations with fixed subsets (e.g.-O3) of optimisations were considered. Furthermore, unlike Chow and Wu (1999), there is no statistical analysis or mathematical quantification of such interactions. Finally, it is clear that their results are strongly affected by the fact that (in most cases) a single transformation, namely *inlining*, dominated the performance obtained in the experiments.

Based upon exhaustive data generated from 15 optimisation phases and sequences of length 12 (on average) on 111 functions belonging to 6 benchmarks from the MiBench suite (Guthaus et al., 2001), Kulkarni et al. (2006) provide an analysis of interactions between optimisation phases within the very portable optimizer (VPO, Benitez and Davidson, 1988). Their focus is on enabling/disabling interactions and order dependencies. While the former occurs when an optimisation phase is enabled or disabled by the prior application of another optimisation phase, the latter relates to the production of identical or different code when applying two optimisation phases in distinct orders. The details of their experimental set-up have been described in section 4.1.4.

A quantification of the enabling interactions is provided by using the so-called *enabling probabilities*. The probability of transformation a being enabled by transformation b was estimated by computing the ratio of the number of times transformation a became active (i.e. effective) to the number of times a became active plus the number of times a remained dormant (i.e. inactive or ineffective), given that a was dormant and b had been applied before. Likewise,

disabling interactions were provided by estimating *disabling probabilities*. The probability of transformation a being disabled by transformation b was estimated by computing the ratio of the number of times transformation a became dormant to the number of times a became dormant plus the number of times a remained active, given that a was active and b had been applied before.

It is shown that many optimisations have a very low probability of being enabled by other optimisations. Additionally, and not surprisingly, it is shown that a transformation is disabled mainly due to the application of the same transformation. The order-dependency between optimisation phases is determined by estimating the probability of two consecutively active phases producing different code. It is shown that most optimisation phases are independent of the order in which they are applied.

The estimated probabilities are then used to reduce the compilation time during the exploration of the optimisation space of the benchmarks under study[5]. This is achieved by selecting the phase that should be applied next to be that with the highest probability of being active. It is shown that the compilation times can be considerably reduced compared to the case of the interaction results being ignored. However, it is found that there was a slight degradation in performance in terms of execution time since the constructed distributions do not take the performance of the sequences into consideration.

We note that the major difference from Kulkarni et al. (2006)'s approach to the technique presented in section 6.5 is that we consider the *performance* of the transformation sequences. Indeed, when two transformations are applicable, there may or may not be interactions that affect the run-time behaviour of a program. Furthermore, if our aim is to use such characterisation of interactions for optimisation then our models are more effective than the approach proposed in Kulkarni et al. (2006), as the latter ignores the effect of interactions and order-dependencies on the speed-up of the program (or function).

Sections 4.1 to 4.4 have described previous work that is related to the main objectives of this thesis. Although the use of unsupervised learning methods in compiler optimisation is not addressed in this project, for completeness, the following section presents some related work on this area.

4.5 Unsupervised Learning

Eeckhout et al. (2002) address the problem of *benchmark selection* and *input data selection* during the design phase of a microprocessor. This problem also known as *workload composition* is relevant to the computer architecture community given that selecting a representative set of benchmark-input pairs from a larger set can significantly reduce simulation times.

[5]Note that by using a heuristic based on the probabilities of interactions or dependencies between transformations, one may well ignore relevant regions of the optimisation space.

They apply standard **hierarchical clustering** on 79 program-input pairs selected from SPECint95 and an additional database workload. In order to characterise each program, a set of 20 microarchitecture-dependent and microarchitecture-independent features are used and their dimensionality is reduced to 4 features by using Principal Components Analysis (PCA) while retaining 93.1% of the variance. It is shown (for three benchmarks) that program-input pairs that are close in the feature space have similar behaviour.

The approach in Eeckhout et al. (2002) is interesting as it proposes a set of program features that can be considered for program characterisation. Furthermore, it is shown experimentally that a subset of benchmarks can be selected as representative of a larger set, which can significantly reduce the number of training examples needed for predictive modelling. However, it is possible to replace heuristic-based clustering (such as hierarchical clustering) by other techniques such as Gaussian Mixture Models (GMM). Additionally, it is also possible to utilise other methods that allow clustering without requiring the direct specification of the number of clusters beforehand, such as the Chinese Restaurant Process Mixture (see e.g. Aldous, 1985, pages 90–91).

Eeckhout et al. (2005a) extend the work in Eeckhout et al. (2002) by increasing the number of benchmark-input pairs to 63 (taken from SPEC CPU2000 integer benchmarks) and applying **k-means** clustering. PCA and Independent Component Analysis (ICA) are used as dimensionality-reduction techniques. This approach was evaluated on the task of predicting the average number of cycles per instruction (CPI). However, as recognised by the authors in a later correction note (Eeckhout et al., 2005b) and contrarily to the results shown in the paper, these two dimensionality-reduction techniques should perform similarly given that the specific ICA algorithm used for the results did apply PCA as a pre-processing stage.

4.6 Summary and Discussion

This chapter has described the related work on applying machine learning or artificial intelligence techniques to compiler optimisation. The review has been focused on *global optimisation*, *predictive modelling*, *performance prediction* and *characterisation* of the optimisation space.

The global optimisation problem, i.e. the task of searching for a "good" sequence or a good set of program transformations, has been addressed in the literature from two different perspectives: using biased random search methods and using statistical techniques. Biased random search methods such as genetic algorithms have been used in previous approaches to tackle the general problem of searching for "good" transformation sequences (see e.g. Cooper et al., 1999; Almagor et al., 2004; Kulkarni et al., 2003). However, the major caveats of these approaches are the very long time needed to achieve a good solution and the dependency of

the methods on specific parameters that are tuned in a heuristic manner. Statistical techniques such as fractional factorial designs or orthogonal arrays can be seen as a more principled way of dealing with the global optimisation problem. Nevertheless, previous approaches (such as Chow and Wu, 1999; Pinkers et al., 2004a; Haneda et al., 2005a) have been limited to dealing with fixed binary sets of optimisation flags.

Most previous work that has addressed the predictive modelling problem, i.e. the use of supervised learning to *learn* good compiler transformations for programs, has been limited to predicting when/how a single program transformation such as loop unrolling or instruction scheduling (e.g. Monsifrot et al., 2002; Stephenson and Amarasinghe, 2005; Cavazos et al., 2004) should be applied. Clearly, such an approach is unrealistic as it ignores the fact that the effect of a particular transformation strongly depends upon the application of other transformations. Although some other approaches have proposed techniques to predict a set of transformations (e.g. Cavazos and O'Boyle, 2006), they have not dealt with the more general sequential prediction problem.

The techniques proposed in chapters 7 and 8 can be seen as more **general** and **efficient** approaches to the compiler optimisation problem. They are more general as, in principle, they can deal with the sequential problem. They are more efficient, as they use the knowledge from previously solved problems in order to improve search. Indeed, these techniques do not consider the global optimisation problem and the predictive modelling problem as unrelated tasks but they rather exploit *transference* across programs (i.e. predictive modelling) in order to speed up search on a new problem that has not been seen before (i.e. global optimisation).

The performance prediction problem, i.e. the task of predicting the performance (e.g. execution time) of a program when applying a set of transformations has also been addressed in the literature. This problem can bee seen as a separate (but related) area of research as it can be applied not only to compiler optimisation (as in Chapter 8) but also to computer architecture design. Most previous work has approached this problem by proposing models which are built in an *ad hoc* manner using architecture-specific characteristics (e.g. Karkhanis and Smith, 2004; Zhao et al., 2003). These approaches are difficult to generalise and to scale to complex architectures. Unlike previous work, the methods proposed in Chapter 8 do not rely upon expert-knowledge and they can be learnt entirely from data. As stated above, these techniques learn across different programs and can be used to tackle the general compiler optimisation problem. Furthermore, these methods can be easily generalised to learn across different architectures by using a higher level of transference (or *multi-task* learning).

Finally, we are not aware of much previous work on the characterisation of the optimisation space. The identification of interactions between program transformations has been tackled by previous approaches but they have been limited to fixed sets of transformations (see e.g. Chow and Wu, 1999; Pinkers et al., 2004a; Haneda et al., 2005a; Lee et al., 2006). Furthermore, a

quantification of such interactions has not been previously investigated (with the exception of Chow and Wu, 1999). Although Kulkarni et al. (2006) do present such quantification, they only consider interactions at compilation time, neglecting the ultimate effect of the transformations on the speed-up of programs. The technique proposed in section 6.5 not only deals with transformation sequences but it also takes the dynamic behaviour of a program into consideration.

Chapter 5

Experimental Set-Up

Chapter 3 described a machine learning approach to the compiler optimisation problem. The main goal of this approach is to improve search on the optimisation space of a new program by achieving *transference* across programs, i.e. by exploiting the knowledge that has been gained from previously solved problems (or optimised programs).

This machine learning approach is based upon a supervised learning scenario, where a set of **features** that characterise programs is required along with a set of **targets** to be learnt from these features. For the compiler optimisation problem, the features can be (for example) characteristics of the code of a program believed to be informative about the targets. In general, these features may well be different for different machine learning techniques used. This is indeed the case for the methods proposed in this thesis and, therefore, we will postpone the description of the features to Chapter 7 and Chapter 8.

Recalling that our ultimate goal is to find a good set of compiler transformations (or a transformation sequence) that makes a program run fastest, the targets for the compiler optimisation application are "good" **transformation sequences**. The "goodness" of a transformation sequence is evaluated according to a measure of performance based on the **execution time** of the program after the application of such a transformation sequence. As we shall see in chapters 6, 7 and 8, one can use a transformed version of the execution times, e.g. the performance speed-ups, in order to make the targets more suitable for learning.

This chapter explains the details of the experiments that gave rise to the target values that can be used for learning. This chapter is important not only because it shows that the experimental design underlying the generation of the data is sound, but also because it allows the reproducibility of the results by other researchers. We start by looking at the general characteristics of the created data set in section 5.1. Afterwards, the benchmarks, the optimisations and the architectures used are described in sections 5.2, 5.3 and 5.4 respectively. Finally, the details of the instrumentation of the programs, i.e. how the execution times have been measured, are given in section 5.5 and the chapter is summarised in section 5.6.

5.1 The SUIF Data Set

In many applications of machine learning and data mining most of the time is spent on data generation, data cleaning[1] and data pre-processing. The compiler optimisation task is not an exception to this rule and, as pointed out in Bonilla (2004), the process of generating clean and reliable data for this particular application is a very time-consuming activity. Therefore, we have spent a lot of effort in generating clean data that represents opportunities for learning.

The data generated for this project is based on the application of **source-to-source** transformations to C programs by using the restructuring compiler framework SUIF 1 (Hall et al., 1996). Henceforth this data set will be called the SUIF data set, and it will be used for the application of the techniques proposed in chapters 6, 7 and 8.

We have selected source-to-source transformations motivated by recent work on *iterative compilation* in the area of embedded systems. It has been shown that searching the optimisation space of source-to-source transformation sequences can lead to significant improvements on the performance of embedded processors. However, this comes at the cost of a large number of evaluations of a program (see e.g. Franke et al., 2005, as a recent reference). Therefore, our aim is to use machine learning techniques in order to improve iterative compilation on embedded systems. Thus, the setting under which the experiments have been carried out includes benchmarks that target embedded systems as well as embedded architectures.

Throughout this thesis we will make use of data that has been generated by the application of length-5 sequences (drawn from 14 code transformations) to 12 different benchmarks. This space of transformation sequences has been exhaustively enumerated and it will be referred to as the small space of the SUIF data set. We have generated this data with the purpose of facilitating the analysis of the results from applying machine learning to the compiler optimisation problem. Indeed, by having an exhaustively enumerated space of transformation sequences one can always determine the global maximum improvement achieved on each benchmark in order to assess the performance of a search algorithm. Moreover, one can make further analyses by using the complete data without relying on samples of an unknown population.

In addition to the small space, a large space has been studied within the context of predictive search distributions (see Chapter 7 for details). The details of this data set, which we call the large space of the SUIF data set, are also presented in this chapter.

5.2 Benchmarks

The UTDSP benchmark suite (Lee, 1997) was created to "evaluate the quality of code generated by a high-level language (such as C) compiler targeting a programmable digital signal

[1] In data mining, the term *data cleaning* refers to the process of detecting and removing errors, inaccuracies and inconsistencies from the data.

processor (DSP)" (Lee, 1997). Twelve different benchmarks from this suite have been used for the experiments. This set of C programs contains small kernels as well as larger applications. As described in Lee (1997), kernels are small fragments of code that do crucial calculations within digital signal processing applications. The applications are complete programs that would execute on a DSP in a commercial product.

The code size of these programs ranges from 20 lines to 500 lines. However, these benchmarks are regarded as compute-intensive programs by the DSP community and they are continuously used in stream-processing applications.

The UTDSP benchmarks are written in different versions according to their "coding style". This coding style corresponds to the use of pointers, array notation and software pipelining. Of the four coding styles that are available in UTDSP, we have selected the non-software-pipelined array-based coding wherever applicable. This style can be considered as the most "natural" implementation of these benchmarks and, in general, it is free of specific heuristics that attempt to hand-craft optimisations, which may turn into misleading results on certain architectures. Table 5.1 presents a brief description of the benchmarks used for the experiments.

5.3 Optimisations

We have considered **source-to-source** transformations applicable to C programs by using the restructuring compiler framework SUIF 1 (Hall et al., 1996). However, many of these transformations are also implemented within the optimisation phases of a native compiler (see for example Almagor et al., 2004).

These transformations have been applied at a program level, i.e. at a global-level of granularity. This has the advantage over finer levels of granularity such as function-level or loop-level in that it makes the effect of the instrumentation negligible. Additionally, it facilitates the implementation of transformations that are not included in SUIF and it reduces the variability (if any) of the measurements. However, it assumes the same optimisation strategy for all code fragments within a program and, therefore, it misses potential optimisation opportunities with respect to the scenario that considers transformations applied at a local level of granularity. This is not much of a problem for the benchmarks in the UTDSP suite since these programs are relatively small. See section 9.2.2 for a discussion of global optimisation versus local optimisation.

Using these transformations, we have investigated two different spaces of transformation sequences: a **large space** with 90 code transformations forming sequences of length 20 and a **small space** composed of 14 transformations (selected by compiler experts) combined into sequences of up to length 5. These spaces are briefly described below.

Kernels	
Benchmark	**Description**
fft_256	256-point complex FFT (Fast Fourier Transform).
fir_256_64	256-tap FIR (Finite Impulse Response) filter processing 64 points.
iir_4_64	4-cascaded IIR (Infinite Impulse Response) biquad filter processing 64 points.
latnrm_32_64	32nd-order Normalised Lattice filter processing 64 points.
lmsfir_32_64	32-tap LMS (Least-mean-squared) adaptive FIR filter processing 64 points.
mult_10_10	$[10 \times 10] \times [10 \times 10]$ matrix multiplication.
Applications	
Benchmark	**Description**
adpcm	An Adaptive Differential Pulse-Coded Modulation encoder.
compress	Compress a 128 x 128 pixel image using the Discrete Cosine Transform.
edge_detect	Detects the edges in a 256 gray-level 128×128 pixel image using 2D convolution and Sobel Operators.
histogram	Enhances a 128×128 pixel image by applying global histogram equalisation.
lpc	Implements a Linear Predictive Coding (LPC) encoder.
spectral_estimation	Calculates the power spectral estimate of an input sample of speech using periodogram averaging.

Table 5.1: UTDSP benchmarks used for the experiments that generated the SUIF data set.

5.3.1 Large Space

The large space of the SUIF data set has been generated within the context of predictive search distributions (see Chapter 7 for more details). It is composed by 90 code transformations forming sequences of length 20, which represents a space of 90^{20} sequences. This space has been sampled by using a PBIL-like algorithm (Baluja, 1994; Baluja and Caruana, 1995), obtaining around 2000 samples per program. The transformations used for this space are listed in Appendix A, Table A.1.

Symbol	Name	Transformation
'1'	$UNRO_1$	Loop unrolling with unroll factor 1
'2'	$UNRO_2$	Loop unrolling with unroll factor 2
'3'	$UNRO_3$	Loop unrolling with unroll factor 3
'4'	$UNRO_4$	Loop unrolling with unroll factor 4
'f'	FLAT	Loop flattening
'n'	NORM	Loop normalisation
't'	TURN	Turn imperfectly nested loop conversion
'k'	BREAK	Break load constant instructions
's'	CSE	Common subexpression elimination
'd'	DEAD	Dead code elimination
'h'	HOIST	Hoisting of loop invariants
'i'	IFH	If hoisting
'm'	MOVE	Move loop-invariant conditionals
'c'	COPY	Copy propagation

Table 5.2: Transformations used for the small space of the SUIF data set.

5.3.2 Small Space

Most of the results presented in this thesis are based on the small space of the SUIF data set. This space is composed by 14 transformations combined into sequences of up to length 5. These transformations are a subset of the transformations used for the large space and their selection was based upon two criteria: i) evidence of their impact on the performance of the programs and ii) compiler experts' knowledge on what transformations were expected to affect the code of the programs under study. For the former, exploratory experiments of the 90 transformations used in the large space were executed and their frequency of ocurrance in sequences that improved the code was taken into consideration. The transformations used for this space are listed in Table 5.2.

The small space of the SUIF data set represents a space of 14^5 sequences. However, We have not considered sequences that include repeated transformations or sequences that include loop unrolling more than once, which has reduced the number of sequences to 149584. We have exhaustively evaluated the performance of all the 149584 sequences. Collecting the data for this task is a time-consuming activity as every sample corresponds to a compilation and an execution of a program. It takes around 3 days to run one benchmark over all the sequences.

It is necessary to remark that all program versions generated by the application of a transformation sequence have been checked for correctness. Therefore, the ultimate code generated

by the compiler after applying such transformation sequences did maintain the meaning of the original programs.

5.4 Architectures

The experiments have been executed on a Texas Instruments C6713 board, a high end floating point DSP running at 225MHz with 256kB of internal memory. The programs were compiled using the Texas Instruments' Code Composer Studio Tools Version 2.21 with the highest -O3 optimisation level. We will refer to this platform henceforth as the **TI** board.

Additional experiments were executed on an AMD Alchemy Au1500 processor running at 500MHz with 16KB instruction cache and 16kB data cache. The programs were compiled using GCC 3.2.1 with -O3 flag, which according to the manufacturer provided the best performance. This platform will be called henceforth the **AMD** architecture. The data on this architecture will be mainly used for the analysis of the the performance of predictive search distributions (see chapter 7 for more details) and for the characterisation of the optimisation space (Chapter 6).

5.5 Instrumentation

As in the application of the code transformations, the instrumentation of the code in order to measure the execution times has also been implemented at a program level. On the TI board, this has been done using the API to the on-chip timer of the board with the function `TIMER_RGET` (Texas Instruments, 2003). This is included in the Code Composer Studio Tools Version 2.21. On the AMD architecture, this has been implemented using the `clock_gettime` function. On both architectures, it has been found experimentally that the variability of the measurements was negligible (less than 1%). This is important as it was not necessary to replicate the experiments several times.

5.6 Summary and Discussion

This chapter has presented the details of the experimental set-up that has given rise to the data used for the application of machine learning techniques to compiler optimisation. Motivated by recent work on *iterative compilation* (see e.g. Franke and O'Boyle, 2001; Franke et al., 2005), we have considered source-to-source transformations implemented in the SUIF infrastructure (Hall et al., 1996), which have been applied to 12 different benchmarks from the UTDSP suite (Lee, 1997).

The experiments targeted an exhaustively enumerated space of transformation sequences drawn from 14 code transformations combined into sequences of up to length 5. The pro-

grams have been executed on a Texas Instruments board (TI). Additional experiments within the context of predictive search distributions (Chapter 7) have been executed. These experiments include a larger optimisation space of transformation sequences drawn from 90 code transformations combined into sequences of length 20 and an AMD architecture.

It should be noted that the source-to-source transformations were applied on top of native compiler-specific transformations. As we shall see in Chapter 6, this approach led to significant improvements in performance on most benchmarks. However, the interpretability of the effect of the transformations is made difficult. Indeed, there may be a direct effect on the ultimate code generated by the compiler or an indirect effect due to the fact that some source-to-source transformations may just enable optimisations that are applied at the low level (by the native compiler).

Chapter 6

Characterisation of the Optimisation Space

This chapter presents a characterisation of the optimisation space of the SUIF data set. The goal here is not only to provide a summary of the data but also to analyse the effect of the transformation sequences used on the execution time of the programs under study. With this purpose, this chapter proposes an extension of the well-known analysis of variance (ANOVA) methodology to deal with sequence data. This technique is used to explain the main effect of the program transformations (on the small space of the SUIF data set) and their interactions when they are applied to a program in a sequential manner.

The organisation of this chapter is as follows. Section 6.1 presents the general issues involved in the exploratory analysis of a compiler optimisation space. Section 6.2 describes the performance speed-ups achieved with the experiments. Section 6.3 gives a preliminary analysis of the optimisation spaces under study by discussing the difficulty of searching these spaces and by providing the most important transformations belonging to the best sequences found. Finally, section 6.4 describes an extension of the analysis of variance methodology (ANOVA) that deals with sequence data and section 6.5 presents the results of applying such a technique to the small space of the SUIF data set in order to understand the effect of single transformations and their interactions.

6.1 Exploratory Data Analysis

The types of problems we address in this thesis are essentially search (or optimisation) problems. Therefore, we are interested in analysing the performance speed-ups achieved for each program, the relevance of the different compiler transformations used and the effect of these transformations on the execution time of the programs. In particular, we will attempt to tackle the following issues regarding the optimisation space of the SUIF data set:

1. Is the search problem a 'needle in a haystack'?
2. What are the most relevant transformations?
3. Does the order of the transformations really matter?
4. Do long sequences produce better results than short sequences?
5. Are the effects of the transformations linearly added?
6. Do the transformations interact? If so, which transformations interact with each other and up to what extent?.

Addressing question 1 to 4 aims at understanding the optimisation space of the problems that will be used for the application of the methods explained in chapters 7 and 8. Addressing questions 5 to 6 aims at tackling the problem of identifying and quantifying *interactions* between program transformations.

6.2 Speed-ups Achieved

Let us start with a preliminary analysis of the maximum performance obtained for each benchmark. Certainly, if no significant improvements were found for any of the benchmarks used there would be no reason to continue with the analysis.

Table 6.1 shows the best speed-ups (as defined in section 2.7) obtained for the small and large spaces on both architectures: the TI and the AMD. We see that significant speed-ups have been obtained on average for both platforms and that most benchmarks could be improved with the experiments. On the small space of the TI, although benchmarks such as *latnrm*, *lmsfir*, *mult* and *histogram* did not experience any reduction in their execution times, other benchmarks such as *fir*, *adpcm*, *compress* and *edge* were dramatically improved. On the small space of the AMD, all but one benchmark (*adpcm*) experienced some improvement in performance. The maximum speed-ups achieved on the small space were 1.84 on the TI board and 2.0 on the AMD architecture. In summary, eight (out of twelve benchmarks) on the TI and eleven (out of twelve benchmarks) on the AMD were improved by some transformation sequence in the small space.

We also see that the speed-ups obtained for the AMD are greater than those obtained for the TI, showing that the compiler in the former (GCC) is easier to improve than the commercial compiler in the latter. Nevertheless, the speed-ups obtained on the TI board are more than encouraging in the compiler community, as the compiler used on this board is believed to produce high quality code for these types of applications.

Finally, these results show that searching a large space such as the one considered in our experiments yields further improvements. These results are important as they show that good

	SPEED-UP TI		SPEED-UP AMD	
PROGRAM	SMALL	LARGE	SMALL	LARGE
FFT	1.04	1.84	1.05	1.08
FIR	1.84	1.86	1.36	1.61
IIR	1.19	1.19	1.42	1.42
LATNRM	1.00	1.02	1.37	1.54
LMSFIR	1.00	1.00	1.43	1.43
MULT	1.00	1.06	1.81	2.00
ADPCM	1.32	1.33	1.01	1.01
COMPRESS	1.64	1.65	1.79	1.89
EDGE	1.30	1.52	1.45	1.39
HISTOGRAM	1.00	1.01	1.33	1.41
LPC	1.12	1.16	1.06	1.07
SPECTRAL	1.08	1.19	1.09	1.36
AVERAGE	1.21	1.32	1.35	1.43

Table 6.1: Maximum speed-ups obtained with the experiments on the SUIF data set.

improvements can be obtained with iterative compilation and that the data generated presents opportunities for learning. Thus, it makes sense to use machine learning techniques such as the ones presented in Chapter 7 and Chapter 8 in order to focus search over good subspaces of transformation sequences.

6.3 Analysis of the Optimisation Space

In this section we provide an analysis of the small (optimisation) space of the SUIF data set (which is described in section 5.3.2). In particular, we are interested in addressing questions 1 to 4 raised in Section 6.1. Let us start by looking at how difficult it is to search the optimisation space of the benchmarks under study. The second column of Table 6.2 shows the percentage of *effective* sequences for each benchmark. We define an effective sequence as a sequence for which all the transformations do change the code. In other words, an effective sequence does not contain idempotent transformations[1]. Note that the number of effective sequences is

[1] We define an idempotent transformation as a transformation that does not change the code.

		Best Sequence	
Program	**Effective (%)**	**TI**	**AMD**
FFT	3.71	{3nm}	{4hns}
FIR	0.69	{h4m}	{3}
IIR	1.80	{3f}	{h4}
LATNRM	2.88	{}	{4mndc}
LMSFIR	0.97	{}	{s3}
MULT	0.48	{}	{4mf}
ADPCM	10.82	{1ish}	{}
COMPRESS	0.85	{4s}	{3mshf}
EDGE	4.56	{f2h}	{fhms4}
HISTOGRAM	0.11	{}	{4}
LPC	7.96	{sn2}	{fcmd4}
SPECTRAL	10.19	{scf}	{3nmdc}

Table 6.2: Percentage of effective sequences and the shortest best sequence for each benchmark on the small space of the SUIF data set. Empty sequences {} mean that there is no transformation sequence that improves performance over the baseline program. (See Table 5.2 for the list of transformations used and their corresponding symbols.)

architecture-independent given that the transformations have been applied at the source level.

We see that, in general, a very small portion of the sequences are effective. Therefore, if a search algorithm does not have a strategy to detect ineffective transformations it could spend a considerable amount of time sampling sequences that have already been executed before, i.e. sequences that include idempotent transformations. Indeed, although for benchmarks such as *adpcm* and *spectral* the number of effective sequences is greater than 10% of the total number of sequences, for benchmarks such as *fir*, *iir*, *lmsfir*, *compress* and *histogram* the number of effective sequences is less than 2%.

6.3.1 Difficulty of the Search Spaces

Only knowing the percentage of effective sequences is not sufficient to ascertain how difficult it is to search the optimisation space of a particular benchmark. Henceforth we will associate the *difficulty* of a search space to the number of samples needed to obtain "good" performance

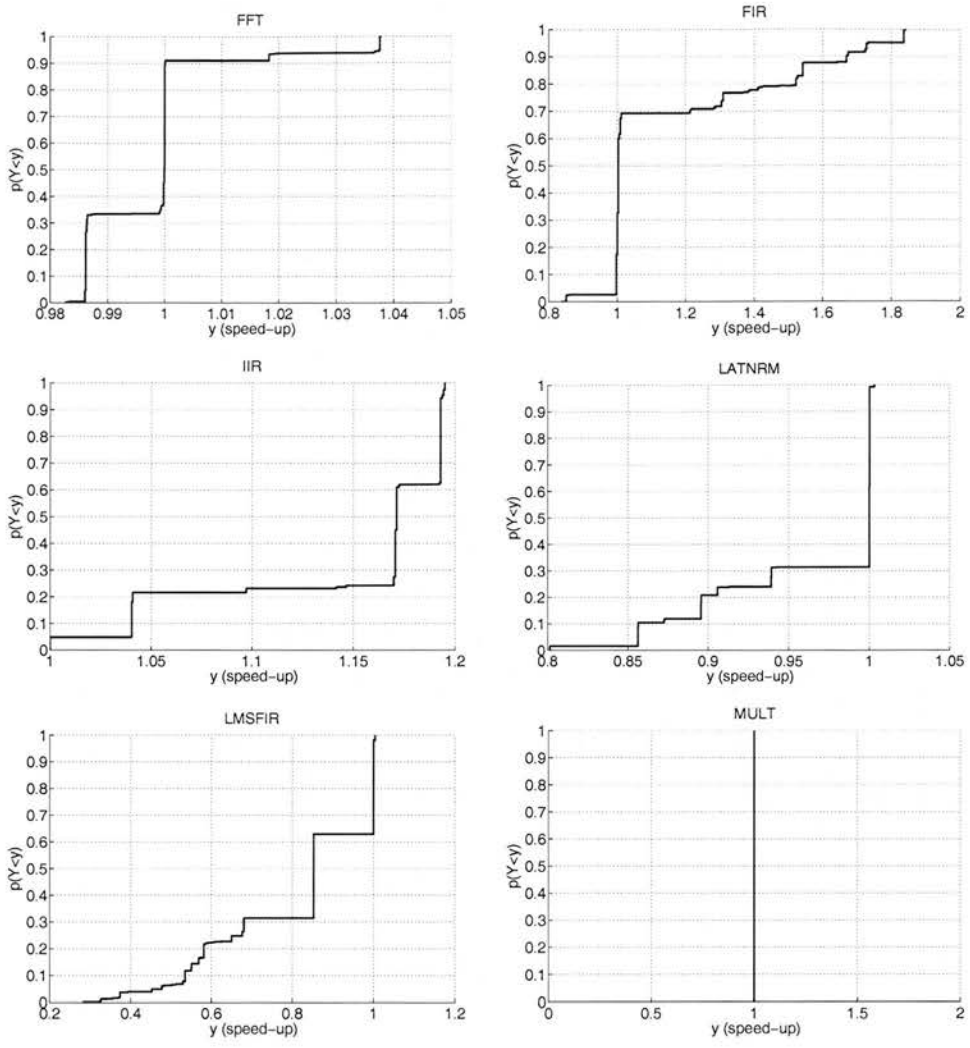

Figure 6.1: The cumulative distribution function of performance speed-ups for the small space of the kernel benchmarks of the SUIF data set on the TI board.

when, for example, using uniform search. Therefore, if we define a good transformation sequence as that providing at least 95% of the maximum possible improvement in performance, then those spaces for which the proportion of good sequences is very small will be more difficult to search than those for which there is a larger amount of transformations sequences that achieve good performance. Hence, we are also interested in analysing the distribution of the performance speed-ups (or the distribution of the execution times). Figures 6.1 and 6.2 show the empirical *cumulative distribution function (cdf)* of performance speed-ups on the TI board for the kernel benchmarks and application benchmarks respectively. Similarly, Figures 6.3 and 6.4 show the cdfs for the kernels and applications on the AMD architecture. Note the differences on the scale of the horizontal axis (speed-up) for distinct benchmarks.

We see that the difficulty of searching the space of transformation sequences is *program-*

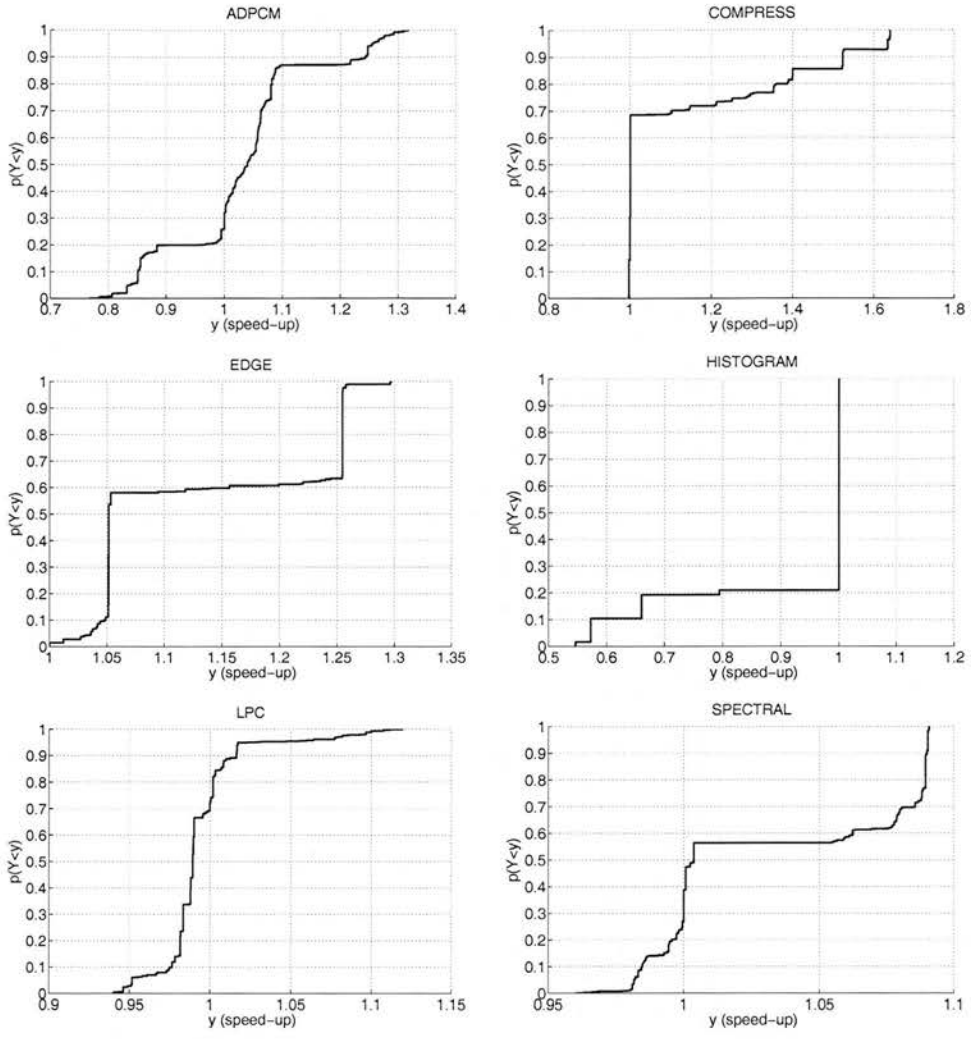

Figure 6.2: The cumulative distribution function of performance speed-ups for the small space of the application benchmarks of the SUIF data set on the TI board.

dependent and *architecture-dependent*. For example, let us consider relatively easy search spaces such as those corresponding to the benchmarks *iir* and *edge* on the TI board. In the case of *iir* (middle left of Figure 6.1), almost 40% of the sequences yield a performance speed-up that is very close to the maximum speed-up of 1.19. A similar case occurs for *edge* (middle left of Figure 6.2), where almost 40% of the sequences achieve an improvement that is at least 95% of the maximum speed-up available. Interestingly, there is no transformation sequence that slows down these programs.

On the TI, more difficult-to-search spaces are found (for example) for *adpcm* and *lpc*. For *adpcm* (top left of Figure 6.2), roughly 20% of the sequences cause a detriment in performace; about 35% of the sequences yield no improvement or degrade performance; almost 88% of the sequences achieve a suboptimal speed-up of 1.2 and less than 2% of the sequences have a speed-up of at least 95% of the maximum improvement. For *lpc* (bottom left of Figure 6.2),

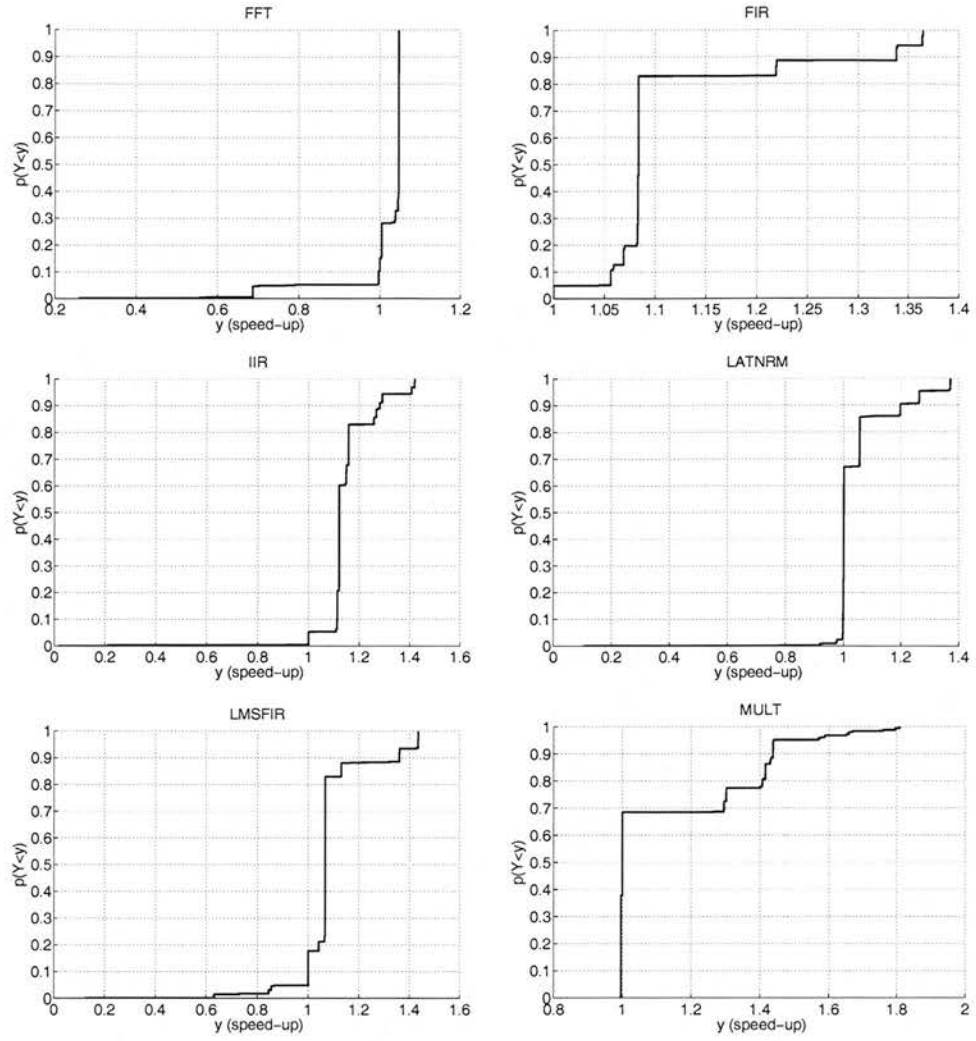

Figure 6.3: The cumulative distribution function of performance speed-ups for the small space of the kernel benchmarks of the SUIF data set on the AMD architecture.

70% of the sequences degrade performance. There are also many sequences that achieve a (very poor) suboptimal performance, i.e. about 25% of the sequences have a speed-up greater than 1 but less than 1.02. Additionally, less than 2% of the sequences have a speed-up of at least 95% of the maximum improvement.

We also see that the easy-to-search spaces found on the TI board, namely *iir* and *edge* are difficult spaces to search on the AMD architecture. On this latter architecture, a very interesting benchmark is *edge* (middle left of Figure 6.4). We note that for this benchmark only a very few number of sequences achieve at least 95% of the maximum speed-up available. Indeed, the search space of this benchmark on the AMD can be considered as a "needle in a haystack".

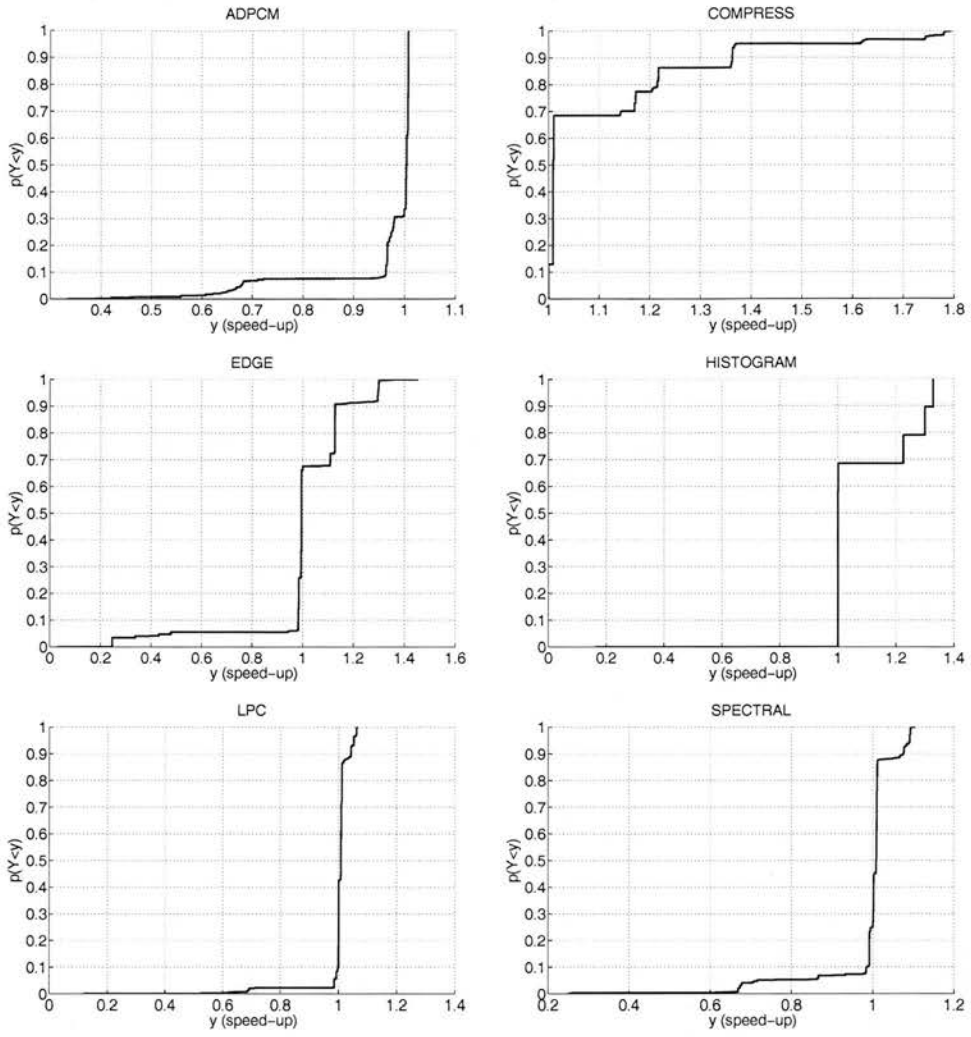

Figure 6.4: The cumulative distribution function of performance speed-ups for the small space of the application benchmarks of the SUIF data set on the AMD architecture.

6.3.2 Important Transformations

From the third column and fourth column of Table 6.2 it is possible to deal (partially) with question 2 and question 4 raised in section 6.1. These questions enquire about the relevant transformations and the minimum length of good sequences respectively.

Here we highlight the transformations that appear in the best sequence found for each benchmark. A deeper analysis of the impact of the transformations on the performance of the programs is presented in section 6.5.

It is evident that unrolling ('1','2','3', or '4') does have an impact on these benchmarks as it always appears in the best sequence (except for the benchmark *spectral* on the TI). This might seem rather obvious for a compiler expert who already knows that loop unrolling is a very important transformation. However, it is surprising that the compilers, which have probably

been tuned for these types of benchmarks, do not have a more aggressive strategy for unrolling and still can be improved by this transformation applied at the source-code level.

Other important transformations are, for example, common sub-expression elimination ('s') that appear four times in the best sequences of improved benchmarks on both architectures, loop flattening ('f'), move loop-invariant conditionals ('m') (especially on the AMD), and hoisting of loop invariants ('h').

It is also possible to note that two transformations, namely turn imperfectly nested loops into perfectly nested loops ('t') and break load constant instructions ('k') do not appear in any of the best sequences. We also see in Table 6.2 that the best speed-ups can be achieved with short and longer sequences on both architectures. In principle, one could design search algorithms that would prefer shorter sequences in order to reduce the compilation time of a program. However, as we have seen in Table 6.1, exploring longer sequences on larger spaces can lead to significantly greater improvements.

We have presented so far an analysis of the optimisation spaces of the SUIF data set with a focus on the characterisation of the speed-ups achieved and the difficulty of searching such spaces. The following sections describe how the standard statistical ANOVA technique can be extended to deal with sequence data and used to analyse the effect of code transformations and their interactions on the execution time of programs.

6.4 Analysis of Variance (ANOVA) of Sequence Data

Analysis of variance (ANOVA) models are widespread in the analysis of experiments and in modelling real-life applications. The main reason for this is their simplicity and easy interpretability. Here we aim to extend the standard ANOVA methodology to deal with sequence data. Our main goal is to construct linear models that are a good fit to sequence data and that allow us to understand how this data is related to some target measure y. Note that, unlike the general machine learning problem, here we are not concerned about generalisation but we are rather interested in having an explanatory model of the data. We start this section by explaining the standard ANOVA model. Afterwards, we extend this model to deal with sequence data and propose different model classes that can be suitable for fitting this data. Such models are then applied to the small space of the SUIF data set in order to characterise the effect of program transformations, their interactions and the relevance of the transformation order in the compiler optimisation problem.

6.4.1 Standard ANOVA

ANOVA-type models are statistical techniques useful to investigate the relationship between a response variable and a set of predictor variables (see e.g. Neter et al., 1996). One advantage

of ANOVA models over regression models is that the predictor variables are not necessarily restricted to quantitative values. However, as we shall see in section 6.4.2, when working with qualitative predictor variables an ANOVA model is equivalent to a linear (in the parameters) regression model, where these variables are represented with indicator variables. Nonetheless, it is useful to look at the ANOVA view of these models in order to understand the effect of the predictor variables on the response values.

The terminology in ANOVA models refers to the predictor variables as *factors* and the possible values these factors can take as *factor levels*. For simplicity, let us explain an ANOVA model that includes only two factors as the generalisation to more factors is straightforward.

The main equation for a two-factor ANOVA model is given by:

$$\mu_{ij} \stackrel{\text{def}}{=} \mu_{..} + \alpha_i + \beta_j + (\alpha\beta)_{ij}, \tag{6.1}$$

where μ_{ij} is the value of the response variable when factor A takes on factor level i and factor B takes on factor level j; $\mu_{..}$ is the overall (or common) mean; α_i is the **main effect** of factor A at level i; β_j is the **main effect** of factor B at level j; and $(\alpha\beta)_{ij}$ is the **interaction effect** between factors A and B at levels i and j respectively. The factor levels for A vary as $i = 1, \ldots, a$ and for B as $j = 1, \ldots, b$.

The parameters of the model in equation (6.1) are defined as follows:

$$\begin{aligned}
\mu_{..} &= \frac{1}{ab}\sum_i\sum_j \mu_{ij}, && (6.2)\\
\alpha_i &= \mu_{i.} - \mu_{..}, && (6.3)\\
\beta_j &= \mu_{.j} - \mu_{..}, && (6.4)\\
(\alpha\beta)_{ij} &= \mu_{ij} - (\mu_{..} + \alpha_i + \beta_i) && (6.5)\\
&= \mu_{..} + \mu_{ij} - \mu_{i.} - \mu_{.j}, && (6.6)
\end{aligned}$$

where: $\mu_{i.} = \frac{1}{b}\sum_j \mu_{ij}$ and $\mu_{.j} = \frac{1}{a}\sum_i \mu_{ij}$. We see in equations (6.3) and (6.4) that the main effects are a measure of how much a factor level mean deviates from the overall mean (Neter et al., 1996, page 801). Equation (6.5) states that there is an interaction between two factor levels if the corresponding mean (μ_{ij}) cannot be explained as an **additive effect** of the overall mean ($\mu_{..}$) plus the mean for each factor level independently ($\alpha_i + \beta_i$). Two factors do not interact if all the interaction values are zero. In this case the factors are said to be additive. Additionally, it is clear that $\sum_i \alpha_i = \sum_j \beta_i = \sum_i (\alpha\beta)_{ij} = \sum_j (\alpha\beta)_{ij} = 0$.

Estimation of ANOVA parameters

Given some noisy observations y_{ijk}, and assuming that:

$$\begin{aligned}
y_{ijk} &= \mu_{ij} + \varepsilon_{ijk} && (6.7)\\
&= \mu_{..} + \alpha_i + \beta_j + (\alpha\beta)_{ij} + \varepsilon_{ijk}, && (6.8)
\end{aligned}$$

where y_{ijk} is the k^{th} observed value of the response variable and ε_{ijk} is the noise level, we wish to estimate the parameters of the model $\mu_{..}$, α_i, β_j and $(\alpha\beta)_{ij}$ for $i = 1,\ldots,a$ and $j = 1,\ldots,b$.

If we further assume that the noise values are independent and distributed according to a normal distribution with zero mean and variance σ^2, i.e. $\varepsilon_{ijk} \sim \mathcal{N}(0,\sigma^2)$ then, given equation (6.8), the response values are also independent and distributed according to: $y_{ijk} \sim N(\mu_{..} + \alpha_i + \beta_j + (\alpha\beta)_{ij}, \sigma^2)$. Thus, we can use maximum likelihood estimation in order to find these parameters. This is equivalent to minimising:

$$\begin{aligned} \mathcal{E} &= \sum_i \sum_j \sum_k (y_{ijk} - \mu_{..} - \alpha_i - \beta_j - (\alpha\beta)_{ij})^2, \\ \text{subject to: } & \sum_i \alpha_i = \sum_j \beta_i = \sum_i (\alpha\beta)_{ij} = \sum_j (\alpha\beta)_{ij} = 0. \end{aligned} \tag{6.9}$$

Solving (6.9) is straightforward and it leads to the following (least-squares) estimates:

$$\widehat{\mu}_{..} = \bar{y}_{...} \tag{6.10}$$

$$\widehat{\alpha}_i = \bar{y}_{i..} - \bar{y}_{...} \tag{6.11}$$

$$\widehat{\beta}_j = \bar{y}_{.j.} - \bar{y}_{...} \tag{6.12}$$

$$\widehat{(\alpha\beta)}_{ij} = \bar{y}_{...} + \bar{y}_{ij.} - \bar{y}_{i..} - \bar{y}_{.j.}, \tag{6.13}$$

where the $\cdot$ notation, as before, means that the average has been taken over the corresponding factor levels and/or replications. Thus, we see the correspondence between equations (6.10) to (6.13) and equations (6.2) to (6.6).

6.4.2 From ANOVA Models to Regression Models

It is clear from equation (6.8) that an ANOVA model can be seen as a linear (in the parameters) regression model (see also e.g. Neter et al., 1996, sections 16.11 and 19.7). A linear (in the parameters) regression model is given by:

$$f(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}), \tag{6.14}$$

$$y = f(\mathbf{x}) + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0,\sigma^2), \tag{6.15}$$

where $\boldsymbol{\phi}(\mathbf{x}) = (\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \ldots, \phi_D(\mathbf{x}))^{\mathrm{T}}$ is the design function corresponding to indicator variables that represent the factors (and their levels) and possibly the interactions between these factors to be included in the design. For convenience in the notation we will assume that the first component of $\boldsymbol{\phi}(\mathbf{x})$ is 1 so that a bias term is included in equation (6.14). The factor effects (i.e. the main effects and the interaction effects) are given by the vector of parameters $\mathbf{w}$.

As an example, let us take an experiment with two factors x_1 and x_2 and 2 levels per factor, e.g. low and high. A suitable design function for a 2-factor interaction model is $\boldsymbol{\phi}(x_1,x_2) = (1, I[x_1], I[x_2], I[x_1]I[x_2])^{\mathrm{T}}$, where $I[x_i]$ is an indicator variable such that $I[x_i] = 1$ if factor x_i takes on the level high and $I[x_i] = 0$ otherwise.

Given a complete set of observations (i.e. a factorial design) $\{\mathbf{x}_i\}_{i=1}^N$ (which include repeated measurements at the same location $\mathbf{x}$) and their corresponding response (or target) values y_i, the parameter vector $\mathbf{w}$ can be estimated from the observed data in order to minimise an objective function such as the mean square error (MSE). In fact, the least squares solution to equation (6.14) is given by:

$$\widehat{\mathbf{w}} = (\Phi\Phi^{\mathrm{T}} + \lambda \mathrm{I})^{-1}\Phi\mathbf{y}, \tag{6.16}$$

where $\Phi = (\boldsymbol{\phi}(\mathbf{x}^1), \ldots \boldsymbol{\phi}(\mathbf{x}^N))$ is the design matrix, $\mathbf{y} = (y_1, \ldots, y_N)^{\mathrm{T}}$ is the vector of responses, I is the identity matrix and λ is a regularisation parameter. The solution given by equation (6.16) is known in the statistics literature as "ridge regression" (Hoerl and Kennard, 1970). We note that for a full factorial design, i.e. when we have a complete set of observations, the matrix $\Phi\Phi^{\mathrm{T}}$ has full rank and therefore $(\Phi\Phi^{\mathrm{T}})^{-1}$ exists. In this case we can use $\lambda = 0$ for the computation of equation (6.16). However, for an incomplete set of observations, or a fractional factorial design, regularisation is required ($\lambda > 0$) if all the parameters of the model need to be estimated as $\Phi\Phi^{\mathrm{T}}$ would be rank-deficient.

Additionally, an unbiased estimator[2] of σ^2 is given by:

$$\widehat{\sigma}^2 = \frac{1}{N-D}\sum_{i=1}^{N}(y_i - \widehat{y}_i)^2, \tag{6.17}$$

where N is the total number of observations (or data-points); D is the number of parameters of the model; y_i is the observed value of the response variable for data-point $\mathbf{x}_i$; and $\widehat{y}_i$ is the response value predicted by the model, i.e. $\widehat{y}_i = \widehat{\mathbf{w}}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_i)$.

It can also be shown (see e.g. Montgomery, 1997, pages 542 and 561) that:

$$\widehat{\mathbf{w}} \sim \mathcal{N}(\mathbf{w}, \sigma^2(\Phi\Phi^{\mathrm{T}} + \lambda \mathrm{I})^{-1}). \tag{6.18}$$

Thus, in order to measure the variability of the regression coefficients we can use the *standard error* $\xi(w_i)$ of each regression coefficient w_i, which is defined as follows:

$$\xi(w_i) = \sqrt{\widehat{\sigma}^2(\Phi\Phi^{\mathrm{T}} + \lambda \mathrm{I})^{-1}_{ii}}, \tag{6.19}$$

where $(\Phi\Phi^{\mathrm{T}} + \lambda \mathrm{I})^{-1}_{ii}$ is the i^{th} element on the diagonal of $(\Phi\Phi^{\mathrm{T}} + \lambda \mathrm{I})^{-1}$.

Goodness-of-fit of the Model

The **goodness-of-fit** of the model can be determined by using the so-called **coefficient of determination**:

$$r^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \widehat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}, \tag{6.20}$$

where N is the total number of observations (or data-points); y_i is the observed value of the response variable for data-point $\mathbf{x}_i$; $\widehat{y}_i$ is the response value predicted by the model; and $\bar{y}$ is the

[2]Note that the maximum likelihood estimator $\widehat{\sigma}^2_{\mathrm{ML}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \widehat{y}_i)^2$ is not an unbiased estimator of σ^2.

mean response value throughout all the data-points. Note that, for simplicity in the notation, we have omitted the dependency of y on the k^{th} observation at the same representation $\mathbf{x}$.

As the coefficient of determination measures the predictive power of the model with respect to the case of always predicting the mean, and the mean is the best least-squares fit when no predictor variables are present, the coefficient of determination measures the predictive power of the features. Additionally, the coefficient of determination can be directly understood as the amount of *variance explained* by the model. Therefore, numbers closer to one are preferred.

Understanding the Regression Parameters

In order to understand the meaning of the parameter vector $\mathbf{w}$ in equation (6.14) let us take the particular example (described above) of an experiment with two factors x_1 and x_2, 2 levels per factor (e.g. low and high) and the response variable y. As above, a suitable design function for a 2-factor interaction model is $\boldsymbol{\phi}(x_1,x_2) = (1, I[x_1], I[x_2], I[x_1]I[x_2])^{\mathrm{T}}$, where $I[x_i]$ is an indicator variable such that $I[x_i] = 1$ if factor x_i takes on the level high and $I[x_i] = 0$ otherwise. Thus, the two-factor interaction regression model is given by:

$$f(\mathbf{x}) = w_0 + w_1 I[x_1] + w_2 I[x_2] + w_{12} I[x_1] I[x_2]. \tag{6.21}$$

Using equations (6.10) to (6.13) and equation (6.8) we see that the expected value of y_{ijk} is $\bar{y}_{ij\cdot}$. In other words, $\widehat{y}_{ij} = f(x_1 = i, x_2 = j) = \mathbb{E}[y_{ijk}] = \bar{y}_{ij\cdot}$. Replacing this value in equation (6.21) for the 4 possible settings of x_1 and x_2 we have that:

$$\widehat{w}_0 = \bar{y}_{00\cdot} \tag{6.22}$$

$$\widehat{w}_1 = \bar{y}_{10\cdot} - \bar{y}_{00\cdot} \tag{6.23}$$

$$\widehat{w}_2 = \bar{y}_{01\cdot} - \bar{y}_{00\cdot} \tag{6.24}$$

$$\widehat{w}_{12} = \bar{y}_{00\cdot} + \bar{y}_{11\cdot} - \bar{y}_{10\cdot} - \bar{y}_{01\cdot}, \tag{6.25}$$

where $\bar{y}_{00\cdot}$ is the mean of the response variable of those examples in which both factors are at the low level (or level zero); $\bar{y}_{10\cdot}$ is the mean of the response variable of those examples in which the factor x_1 is at the high level (or level one) and the factor x_2 is at the low level; $\bar{y}_{01\cdot}$ is the mean of the response variable of those examples in which the factor x_1 is at the low level and the factor x_2 is at the high level; and $\bar{y}_{11\cdot}$ is the mean of the response variable of those examples in which both factors are at the high level. These parameter estimates provide a measure of the main effects and interaction effects of the factors x_1 and x_2.

The advantage of the regression view of ANOVA models is that we can extend this technique in order to deal with sequence data. We are particularly interested in extracting a set of informative features from sequences that allow us to analyse e.g. the main effects of different literals within a sequence and their pairwise interactions.

Thus, we aim at evaluating "good" representations for sequences by looking at the coefficients of determination obtained when applying different models corresponding to distinct representations. Furthermore, we will analyse the strength of the effects of such features and their interactions by looking at the parameter vector $\mathbf{w}$.

6.4.3 Sequence ANOVA

Let us define the alphabet $\mathcal{A} = \{x_1, \ldots x_{|\mathcal{A}|}\}$, from which we can draw sequences of arbitrary length L, i.e. $\mathbf{x} = (x_{i_1}, x_{i_2}, \ldots, x_{i_L})$ with $i_j \in \{1, \ldots, |\mathcal{A}|\}$. For each of these sequences we measure a target value y so that our observed data is given by $\mathcal{D} = \{(\mathbf{x}^i, y^i)\}_{i=1}^N$, where N is the number of observations.

Given the data $\mathcal{D}$, we want to analyse the effects of each element (or literal) of the alphabet $\mathcal{A}$ on the target values y. We are also interested in the effect of pairwise interactions and (if any) in the effect of higher order interactions. The model classes we will consider for this purpose are presented below.

Bag-of-characters Model

In this model class we only account for the presence or absence of a particular literal within a sequence, ignoring any positional or order information. For example, a bag-of-characters model that includes main effects and pairwise interactions is given by:

$$f(\mathbf{x}) = w_0 + \sum_i w_i I[x_i] + \sum_{i,j} w_{ij} I[x_i] I[x_j], \tag{6.26}$$

where $I[x_i]$ is an indicator variable such that $I[x_i] = 1$ if the literal x_i appears within a sequence and $I[x_i] = 0$ otherwise.

Positional Model

In this model class we consider not only the presence or absence of a literal within a sequence but also the position where that literal appears. For example, a positional model that includes second order interactions is given by:

$$f(\mathbf{x}) = w_0 + \sum_{i,j} w_{ij} I[x_i^j] + \sum_{i,j} \sum_{k,l} w_{ijkl} I[x_i^k] I[x_j^l], \tag{6.27}$$

where $I[x_i^k]$ is an indicator function so that $I[x_i^k] = 1$ if the literal x_i appears in position k and $I[x_i^k] = 0$ otherwise.

Order-based Model

In this model class we include a main-effect model as given by the bag-of-characters model along with higher order interactions that depend on the order of the literals applied within a

sequence. In other words, the positional information is neglected and only the order information is considered. For example, an order-based model that considers three-factor interactions is given by:

$$f(\mathbf{x}) = w_0 + \sum_i w_i I[x_i] + \sum_{i,j} w_{ij} I[x_i, x_j] + \sum_{i,j,k} w_{ijk} I[x_i, x_j, x_k], \tag{6.28}$$

where $I[x_i]$, $I[x_i, x_j]$ and $I[x_i, x_j, x_k]$ are indicator functions so that $I[x_i] = 1$ if the literal x_i appears within the sequence $\mathbf{x}$ and $I[x_i] = 0$ otherwise; $I[x_i, x_j] = 1$ if the literal x_j appears after the literal x_i and $I[x_i, x_j] = 0$ otherwise; and $I[x_i, x_j, x_k] = 1$ if the literal x_k appears after the literal x_j, which also appears after the literal x_i and $I[x_i, x_j, x_k] = 0$ otherwise. It is important to emphasise that in the definition of $I[x_i, x_j]$ and $I[x_i, x_j, x_k]$ the literals are not necessarily contiguous.

Note that the order-based model can be seen as a general ANOVA model class for sequences in the sense that the sequences under study are not required to have the same length and the positional information is not specifically encoded. As we shall see in section 6.5, this latter characteristic of the model may be very useful for the analysis of interactions between program transformations in the compiler optimisation problem. We also note that the features extracted in the *bag-of-characters* model and the *order-based model* are related to string kernels (see e.g. Rasmussen and Williams, 2006, section 4.4.1 and references therein).

Understanding the Regression Parameters in the Sequence Model

The parameters of this model have similar definitions to those provided in equations (6.22) to (6.25). The only difference here is that the order of the transformations is taken into consideration. In particular, let us take an example of an order-based model for sequences with two literals x_1 and x_2 and pairwise interactions so that:

$$f(\mathbf{x}) = w_0 + w_1 I[x_1] + w_2 I[x_2] + w_{12} I[x_1, x_2] + w_{21} I[x_2, x_1], \tag{6.29}$$

where $I[x_i]$ and $I[x_i, x_j]$ are defined as above. The least-squares estimates of the model parameters can be obtained by using:

$$\widehat{w}_0 = \overline{y}_{\neg x_1, \neg x_2} \tag{6.30}$$

$$\widehat{w}_1 = \overline{y}_{x_1, \neg x_2} - \overline{y}_{\neg x_1, \neg x_2} \tag{6.31}$$

$$\widehat{w}_2 = \overline{y}_{\neg x_1, x_2} - \overline{y}_{\neg x_1, \neg x_2} \tag{6.32}$$

$$\widehat{w}_{12} = \overline{y}_{\neg x_1, \neg x_2} + \overline{y}_{x_1, x_2} - \overline{y}_{x_1, \neg x_2} - \overline{y}_{\neg x_1, x_2} \tag{6.33}$$

$$\widehat{w}_{21} = \overline{y}_{\neg x_1, \neg x_2} + \overline{y}_{x_2, x_1} - \overline{y}_{x_1, \neg x_2} - \overline{y}_{\neg x_1, x_2}, \tag{6.34}$$

where $\overline{y}_{\neg x_1, \neg x_2}$ is the mean of the response variable for those sequences in which neither x_1 or x_2 appear; $\overline{y}_{x_1, \neg x_2}$ is the mean of the response variable for those sequences in which only x_1 appears; $\overline{y}_{\neg x_1, x_2}$ is the mean of the response variable for those sequences in which only x_2 appears; $\overline{y}_{x_1, x_2}$

is the mean of the response variable for those sequences in which x_2 appears after x_1; and $\overline{y}_{x_2,x_1}$ is the mean of the response variable for those sequences in which x_1 appears after x_2.

The above expressions for the regression parameters hold for the *ord-two* model only if it is fitted to all sequences of up to length 2. If one uses longer sequences then the parameters need to be estimated using the general equation (6.16).

6.5 Sequence ANOVA for Compiler Optimisation

In order to understand the effect of compiler transformations and their interactions on the small space of the SUIF data set, we can readily fit the model classes described in section 6.4.3 with different levels of interactions. We identify a transformation sequence as a sequence of singleton transformations x_i drawn from our alphabet of 14 transformations (given in Table 5.2).

The models that will be applied to all (149584) sequences of up to length 5 in the small space of the SUIF data set are given in Table 6.3. We see that (as explained in section 6.4.2) given the complete data for the small space of the SUIF data set, the matrix $\Phi\Phi^{\mathrm{T}}$ has full rank for the models that use a bag-of-characters representation or an order-based representation. Thus, we have used $\lambda = 0$ for the computation of equation (6.16) on these models. However, we note that, by definition, the matrix $\Phi\Phi^{\mathrm{T}}$ is rank-deficient on the models that are based on a positional representation. Thus, for these models we have used regularisation (i.e. $\lambda > 0$) so that equation (6.16) can be computed numerically. It has been found that the results reported for these models are insensitive to a wide range of values of the regularisation parameter, i.e. $10^{-8} < \lambda < 1$.

6.5.1 Results on The Small Space of the SUIF Data Set

Table 6.4 shows the explained variance for the different models (described above) on the small space of the SUIF data set for the TI board. We see that for some benchmarks (*fir*, *latnrm*, *lmsfir*, *edge*, *histogram*, *spectral*) a simple main effects model with a bag-of-characters representation seems to be sufficient to explain most of the variance of the data. Additionally, considering 2-factor interactions with a bag-of-characters representation provides a better fit, especially for program *iir*, which is an indication that interactions between program transformations do take place. This is seen more clearly in the results obtained by the **ord:two** model, where for all the programs but *adpcm* almost all the data variance is explained. This reveals that the ordering information, i.e. considering interactions of transformations depending on when they have been applied (after/before other transformation) is important, in particular for benchmarks *fft*, *iir* and *lpc*. Positional information does not seem to provide any improvement over the equivalent order-based model in terms of variance explained. We also see that a three-

Model	Description	D	rank($\Phi\Phi^T$)
bag:main	*Main effects* model using a *bag-of-characters* representation. It only considers the presence/absence of a particular transformation within a sequence.	15	Full
bag:two	Main effects and *2-factor interactions* using a *bag-of-characters* representation.	100	Full
pos:main	*Main effects* using a *positional* representation.	71	70
pos:two	Main effects and *2-factor interactions* using a *positional* representation.	1771	1630
ord:two	Main effects and *2-factor interactions* using an *order-based* representation.	185	Full
ord:three	Main effects, 2-factor interactions and *3-factor interactions* using an *order-based* representation.	1985	Full

Table 6.3: The ANOVA models considered for the analysis of the small space of the SUIF data set. From left to right we have the name of the model, the description, the number of parameters of the model (D) and the rank of the matrix resulting from multiplying the design matrix times its transpose.

factor interaction model provides a slight improvement on benchmarks *iir* and *adpcm*. This latter program seems to be a complex benchmark in the sense that interactions involving more that three transformations are required to explain most of the data variance.

Table 6.5 shows the explained variance for the ANOVA models considered on the small space of the SUIF data set for the AMD board. We see that, unlike on the TI board, on this architecture the main-effect model with a bag-of-characters representation only explains most of the data variance on two benchmarks: *mult* and *histogram*. Having a two-factor interaction model with a bag-of-characters representation does not provide significant improvements and it is necessary to model two-factor interactions with an order-based representation in order to explain most of the variance on most benchmarks. We also see, as on the TI, that the positional model does not provide significant improvements over the equivalent order-based model. However, even the order-based model that includes two-factor interactions is a poor fit for benchmarks such as *adpcm*, *lpc* and *spectral*. Finally, we see that (on the AMD architecture) the benchmarks *adpcm* and *spectral* are not well explained by a three-factor interaction model and they require the modelling of higher order interactions.

Benchmark	$r^2_{bag:main}$	$r^2_{bag:two}$	$r^2_{pos:main}$	$r^2_{pos:two}$	$r^2_{ord:two}$	$r^2_{ord:three}$
FFT	0.44	0.61	0.56	0.99	0.99	0.99
FIR	0.88	0.92	0.89	0.97	0.96	0.99
IIR	0.33	0.78	0.35	0.85	0.83	0.98
LATNRM	0.93	0.96	0.94	0.99	0.99	0.99
LMSFIR	0.94	0.96	0.94	0.99	0.98	0.99
MULT	-	-	-	-	-	-
ADPCM	0.04	0.16	0.06	0.32	0.23	0.52
COMPRESS	0.82	0.86	0.88	0.99	0.99	0.99
EDGE	0.92	0.95	0.93	0.99	0.99	0.99
HISTOGRAM	0.99	0.99	0.99	1.00	1.00	1.00
LPC	0.39	0.51	0.54	0.98	0.97	0.99
SPECTRAL	0.89	0.94	0.90	0.99	0.99	0.99

Table 6.4: Explained variance for different ANOVA models fitted to the small space of the SUIF data set on the TI board. The models differ in their coding representation and the order of interactions modelled: "bag", "pos" and "ord" stand for bag-of-characters, positional and order-based representation respectively. "main", "two" and "three" stand for main effects, two-factor interactions and three-factor interactions respectively.

6.5.2 Analysis of Main Effects and Two-Factor Interactions

In order to analyse the main effects of the program transformations and their pairwise interactions we report the values $w_i/\|\mathbf{w}\|_\infty$ and $w_{ij}/\|\mathbf{w}\|_\infty$, in the *ord:two* model, where w_i and w_{ij} are the values of the coefficients of the main factors and pairwise factors as given by equation (6.28) and $\|\mathbf{w}\|_\infty$ is the infinity norm of the whole vector of coefficients, i.e. $\|\mathbf{w}\|_\infty = \max(|w_1|,\ldots|w_{|\mathcal{A}|}|,|w_{11}|,\ldots|w_{|\mathcal{A}||\mathcal{A}|}|)$ with $|\mathcal{A}|$ being the total number of transformations. Note that we could report the p-values resulting from assessing the hypothesis that these coefficients are different from zero. However, even if such coefficients are statistically significantly different from zero, they may well be irrelevant in practice.

Nonetheless, only those coefficients that are statistically significant are reported. Given the test statistic $\tau(w_i) = w_i/\xi(w_i)$, where $\xi(w_i)$ is the standard error of the i^{th} regression coefficient w_i, as defined in equation (6.19), the coefficient w_i (or w_{ij}) is said to be statistically significantly different from zero at the 0.05 significance level if $|\tau(w_i)| > t_{0.05/2,N-D-1}$, where $t_{0.05/2,N-D-1}$ is the $1-0.05/2$ percentile of the *t-student* distribution with $N-D-1$ degrees of freedom, N is the total number of sequences and D is the number of parameters in the model (see e.g. Montgomery, 1997, pages 557 and 558).

Figures 6.5 to 6.8 show the main effects of the program transformations and their inter-

Benchmark	$r^2_{bag:main}$	$r^2_{bag:two}$	$r^2_{pos:main}$	$r^2_{pos:two}$	$r^2_{ord:two}$	$r^2_{ord:three}$
FFT	0.13	0.23	0.23	0.56	0.44	0.78
FIR	0.43	0.51	0.54	0.78	0.62	0.92
IIR	0.28	0.38	0.48	0.75	0.69	0.93
LATNRM	0.30	0.35	0.60	0.82	0.71	0.89
LMSFIR	0.41	0.58	0.51	0.87	0.85	0.99
MULT	0.91	0.93	0.93	0.99	0.99	0.99
ADPCM	0.02	0.05	0.08	0.23	0.13	0.38
COMPRESS	0.72	0.79	0.79	0.99	0.99	1.00
EDGE	0.15	0.33	0.21	0.48	0.44	0.77
HISTOGRAM	0.99	0.99	0.99	0.99	0.99	0.99
LPC	0.07	0.12	0.11	0.28	0.20	0.44
SPECTRAL	0.06	0.12	0.15	0.37	0.31	0.67

Table 6.5: Explained variance for different ANOVA models fitted to the small space of the SUIF data set on the AMD architecture. The models differ in their coding representation and the order of interactions modelled: "bag", "pos" and "ord" stand for bag-of-characters, positional and order-based representation respectively. "main", "two" and "three" stand for main effects, two-factor interactions and three-factor interactions respectively.

actions on each benchmark for the small space of the SUIF data set for the TI board and the AMD architecture using the *ord:two* model. The first column of each plot corresponds to the main effects and the other columns correspond to the two-factor interaction effects of applying a program transformation on the vertical axis followed by a program transformation on the horizontal axis. The results for the benchmarks on which the model explains less than 60% of the variance are not shown given that higher-order interactions are needed to provide a good fit for these programs.

One common feature across all the graphs is the absence of values for the interactions between a program transformation that is followed by the same transformation. This is due to the experimental design adopted which did not include repeated transformations (see section 5.3.2).

Main Effects

On the TI board (figures 6.5 and 6.6) we see that *unrolling* (with factors 2,3,4) has a significant impact on most benchmarks with the exception of *fft*. Similarly, loop *flattening* has an important main effect especially on benchmarks *iir*, *edge* and *spectral*. Other transformations with significant main effects are *cse*, *hoist* and *copy*. Transformations such as *turn*, *break* and

dead do not have a significant main effect on this architecture.

Similarly, on the AMD architecture (figures 6.7 and 6.8) we see that *unrolling* has a significant and consistent main effect across benchmarks. Other transformations worth mentioning are *flattening*, *cse*, *hoist*, and *copy*. As on the TI, *break*, *turn* and *dead* do not have a significant main effect on the benchmarks used.

Interactions

A word of caution: Before describing the pairwise interactions between program transformations on the small space of the SUIF data set, it is important to recall the meaning of these interactions. As explained throughout section 6.4 and exemplified in equations (6.30) to (6.34), there is an interaction between two factors if their joint effect on the response variable cannot be explained as an additive effect of both factors when they are applied independently. In the compiler optimisation problem the identification and quantification of these interactions with the *ord:two* model helps to identify enabling/disabling transformations and to recognise when the order of these transformations has an impact on the speed-up of a program. However, caution must be exercised when interpreting the sign of the interaction effect. In particular, a negative interaction value for two transformations does not necessarily mean that the joint application of the transformations has a negative impact on the speed-up of the program, as it only indicates that the expected speed-up of the program when both transformations are jointly applied is less than the sum of the individual expected speed-ups when such transformations are applied in isolation.

Figures 6.5 and 6.6 show the two-factor interaction effects of applying a program transformation on the vertical axis followed by a program transformation on the horizontal axis for the TI board. In general, *unrolling* has a big impact when interacting with other transformations, especially when interacting with *norm*, *flattening* and *cse*. Interestingly, *norm* has very little impact when applied in isolation. Additionally, such interactions can be positive as in the case of the benchmark *lmsfir* (Figure 6.5) when unrolling interacts with *cse*, or they can be negative as in the case of the benchmark *fir* (Figure 6.5) when unrolling interacts with *norm*. Other interactions worth mentioning are *norm–move* and *cse–hoist* (on *fft*); *hoist–cse* (on *iir*); and *move-cse* (on *lmsfir*).

The interactions between transformations can be order-dependent or order-independent, which can be seen in the symmetric/asymmetric properties of the plots in figures 6.5 and 6.6. For example, we see that for the benchmark *iir* there are significant interactions *unroll–flat* or *unroll–cse*, which do not rely on the order these transformations have been applied. A similar case occurs on the benchmark *lmsfir* for the interaction *unroll–cse*. Contrarily, on the benchmark *lpc* there are significant interactions that rely on applying *unrolling* (e.g. with unroll factor 4) after *norm* but the reverse interaction *unroll–norm* (with unroll factor 4) is rather weak.

A closer analysis of the data shows that while the mean speed-ups of those sequences where $unro_4$ appears or where *norm* appears are 0.9737 and 1.0099 respectively, the mean speed-up of those sequences where $unro_4$ appears (not necessarily contiguously) after *norm* is 1.0557. However, the reverse interaction leads to a slow down of the program as shown by the average speed-up of those sequences where *norm* appears after $unro_4$ being 0.9476. Furthermore, the sequence {n4} yields a speed-up of 1.0774, which is over 96% of the maximum speed-up available for the program *lpc* (shown in Table 6.1).

As a final observation regarding the main effects and interactions of the program transformations on the TI, we can conclude from Table 6.4 and Figure 6.6 that the benchmark histogram is completely dominated by the effect of *unrolling* and *normalisation*.

Figures 6.7 and 6.8 show the two-factor interaction effects of applying the program transformations on the vertical axis followed by the program transformation on the horizontal axis for the AMD architecture. As in the case of the TI board, we see that unrolling has a significant impact when interacting with other transformations such as *norm*, *cse*, *hoist*, *flat* and *copy*. Other interactions worth mentioning are *cse–hoist* and *hoist–cse* on *fir*, *iir*, *latnrm* and *lmsfir*; *cse–norm*, *cse–copy*, *copy–cse*, *hoist–norm*, *hoist–copy*, *copy–hoist* and *copy–norm* on *latnrm*; and *flat–cse* and *cse–flat* on *iir*.

From Table 6.5 and Figure 6.8 we can conclude that the benchmark *histogram* is almost completely dominated by the effect of unrolling, and the benchmark *mult* is almost completely dominated by the effects of *unrolling* and *flattening*. Additionally, we note that for some benchmarks such as *iir* and *lmsfir*, *unrolling* has a positive main effect when applied in isolation but a negative impact when applied in conjunction with other transformations such as *cse* and *hoist*.

6.5.3 Related Work on the Effect of Compiler Transformations

A final remark regarding the main effects of program transformations and their interactions within transformation sequences is that the impact of these sequences on the execution time of a program is crucial for modelling such effects. We note this by looking at the different patterns of main effects and interactions for the same program on the two architectures (see e.g. the benchmark *latnrm* on figures 6.5 and 6.7). Indeed, approaches such as the one adopted in Kulkarni et al. (2006, see section 4.4 of this thesis for details) that neglect the effect of a transformation sequence on the final execution time of the program, would fail at effectively characterising the main effects of program transformations and their interactions.

Other authors have used statistical techniques from the design of experiments in compiler optimisation (see e.g. Chow and Wu, 1999; Pinkers et al., 2004a,b). The main difference of the technique presented in section 6.5 with respect to previous approaches is that it deals with the general problem of analysing sequence data rather than a fixed set of compiler transformations.

For example, Pinkers et al. (2004a,b) use *orthogonal arrays* in order to effectively sample

a large set of compiler optimisation flags. However, their approach only considers the main effects of compiler options neglecting their interaction effects. Chow and Wu (1999) present a quantification of the main effects of a fixed set of optimisation flags and their interactions using *fractional factorial designs*. Although their analysis considers the uncertainty of such effects, the goodness of fit of the final model used is not reported. The reader is referred to section 4.4 for more details on the work related to this area.

6.5.4 Optimisation with the Sequence ANOVA Model

The regression view of the sequence ANOVA model as given by the *ord:two* model (i.e. the two factor interaction model with an order-based representation) allows us to use such a model for optimisation. The following analysis relies upon fitting this model to the complete data of each benchmark corresponding to the small space of the SUIF data set. Thus, for each benchmark, the predictions given by the *ord:two* model on all the sequences can be used in order to ascertain the sequence that is expected to provide the best performance[3].

Let us call the actual speed-up achieved by such a sequence y^{anova} and the maximum speed-up that can be obtained y^{best}. Table 6.6 shows the ratios $y^{\text{anova}}/y^{\text{best}}$ obtained on the small space of the SUIF data set. We see that the model achieves almost the best possible performance on most benchmarks on both architectures (with the exception of *latnrm* on the AMD). In fact, for *edge* on the TI and *fir* and *histogram* on the AMD, the sequence predicted by using the sequence ANOVA model yields the best possible performance that can be obtained in the transformation space considered.

6.6 Summary

This chapter has presented a characterisation of the optimisation space of the SUIF data set. In particular, the speed-ups achieved with the experiments have been analysed and have been shown to be relevant for the application of machine learning techniques to the different optimisation problems. Additionally, the difficulty of the search spaces has been studied by using the cumulative distribution function of performance speed-ups on each benchmark. It has been shown that this difficulty is program-dependent and architecture-dependent and that some benchmarks are more interesting than others in terms of their optimisation space.

With the goal of providing an understanding of the effect of interactions between program transformations, the technique of analysis of variance has been extended in order to deal with sequence data. This extension has been applied to the data generated for the small space of the SUIF data set. It has been shown that although some benchmarks are well explained by

[3]Obviously, if there was exhaustive data for a specific benchmark it would not be necessary to search for a "good" optimisation sequence and, in practice, such models must be fitted to a (non-exhaustive) sample of transformations sequences.

	y^{anova}/y^{best}	
Benchmark	**TI**	**AMD**
FFT	0.99	0.99
FIR	0.99	1.00
IIR	0.99	0.98
LATNRM	-	0.79
LMSFIR	-	0.99
MULT	-	0.99
COMPRESS	0.99	0.99
EDGE	1.00	-
HISTOGRAM	-	1.00
LPC	0.97	-
SPECTRAL	0.99	-

Table 6.6: The performance of the sequence ANOVA model (*two-factor interaction model with an order-based representation*) when used for optimisation on the small space of the SUIF data set for the TI and the AMD. The ratio between the speed-up obtained with the sequence predicted by the model (y^{anova}) and the maximum speed-up (y^{best}) on each benchmark is shown. The results are reported only on those benchmarks on which some improvement can be achieved (i.e. $y^{best} > 1$) and on which the model explains at least 60% of the data variance.

a *main effects model*, other benchmarks require the inclusion of two-factor *interaction*s and higher-order interactions in order to explain most of the data variance. An important finding of applying the technique proposed is that the information provided by the position of a program transformation within a sequence is irrelevant and, therefore, one can rely on models that only take the order information into account. This observation can be used, for example, to drive optimisation algorithms without the need for expensive representations of a transformation sequence.

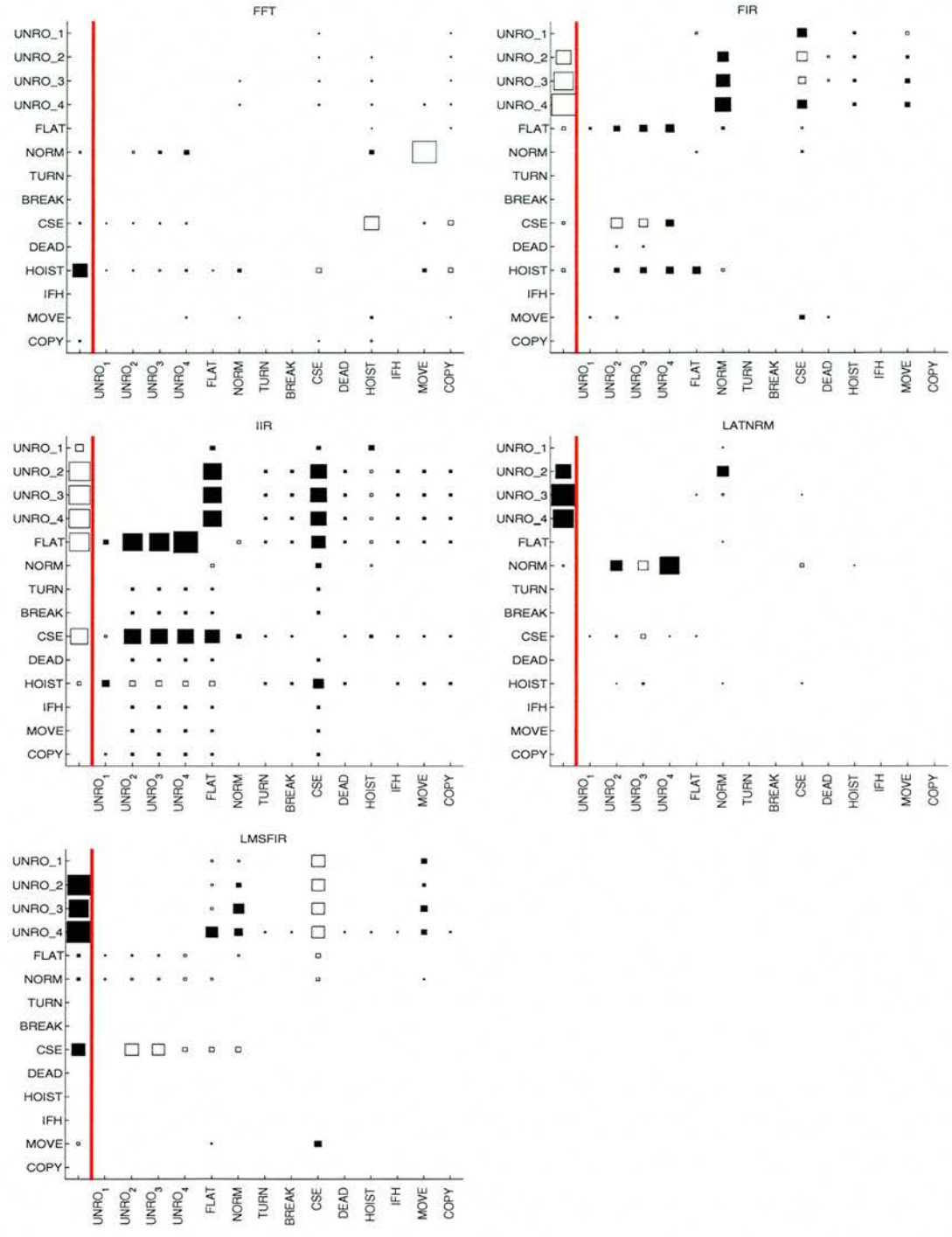

Figure 6.5: Significant main effects (first column of each graph) and two-factor interactions between transformations on the vertical axis followed by the transformations on the horizontal axis for the small space of the kernel benchmarks of the SUIF data set on the TI board. The area of each region in the Hinton diagrams is proportional to its corresponding normalised coefficient ($w_i/\|\mathbf{w}\|_\infty$ or $w_{ij}/\|\mathbf{w}\|_\infty$) in the *two-factor interaction model with an order-based representation*. Black regions correspond to negative values and white regions correspond to positive values. The program *mult* is not reported as there is not variability on its speed-ups.

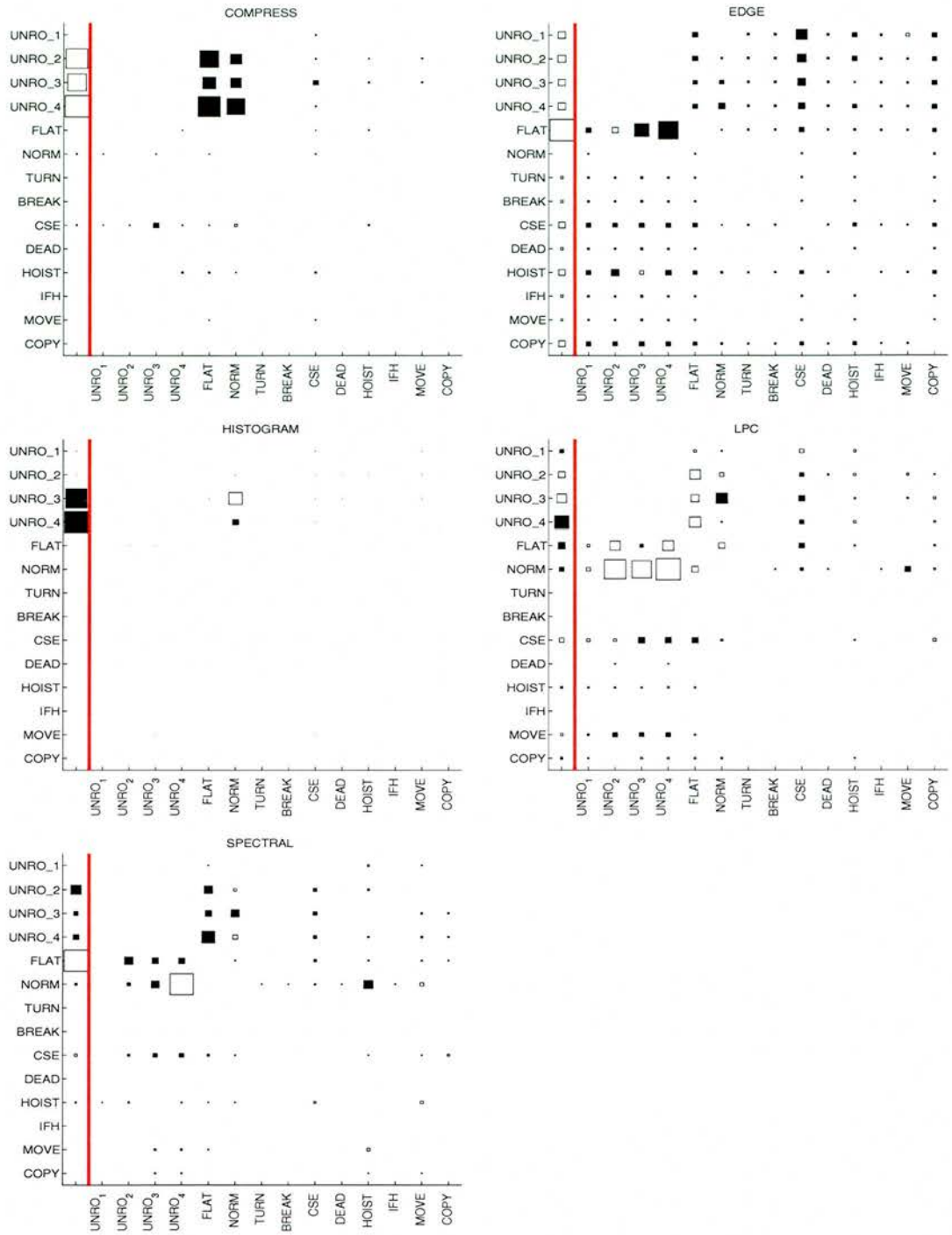

Figure 6.6: Significant main effects (first column of each graph) and two-factor interactions between transformations on the vertical axis followed by the transformations on the horizontal axis for the small space of the application benchmarks of the SUIF data set on the TI board. The area of each region in the Hinton diagrams is proportional to its corresponding normalised coefficient ($w_i/\|\mathbf{w}\|_\infty$ or $w_{ij}/\|\mathbf{w}\|_\infty$) in the *two-factor interaction model with an order-based representation*. Black regions correspond to negative values and white regions correspond to positive values. Benchmarks for which the model explains less than 60% of the variance are not reported.

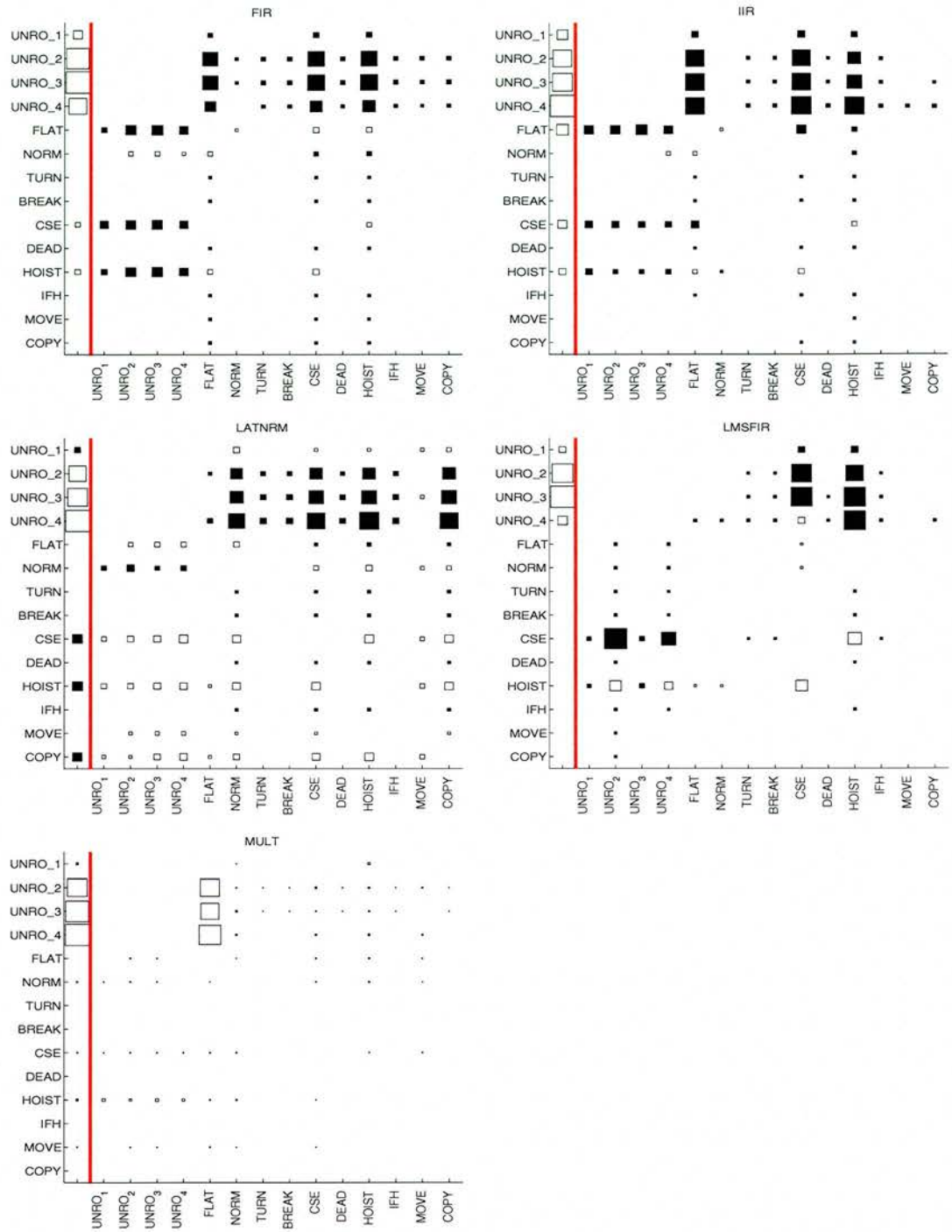

Figure 6.7: Significant main effects (first column of each graph) and two-factor interactions between transformations on the vertical axis followed by the transformations on the horizontal axis for the small space of the kernel benchmarks of the SUIF data set on the AMD. The area of each region in the Hinton diagrams is proportional to its corresponding normalised coefficient ($w_i/\|\mathbf{w}\|_\infty$ or $w_{ij}/\|\mathbf{w}\|_\infty$) in the *two-factor interaction model with an order-based representation*. Black regions correspond to negative values and white regions correspond to positive values. Benchmarks for which the model explains less than 60% of the variance are not reported.

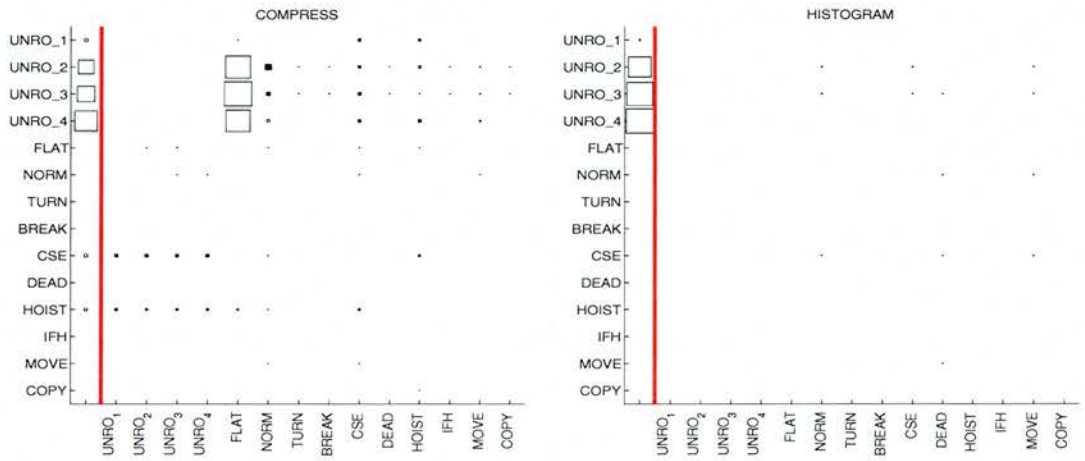

Figure 6.8: Significant main effects (first column of each graph) and two-factor interactions between transformations on the vertical axis followed by the transformations on the horizontal axis for the small space of the application benchmarks of the SUIF data set on the AMD. The area of each region in the Hinton diagrams is proportional to its corresponding normalised coefficient ($w_i/\|\mathbf{w}\|_\infty$ or $w_{ij}/\|\mathbf{w}\|_\infty$) in the *two-factor interaction model with an order-based representation*. Black regions correspond to negative values and white regions correspond to positive values. Benchmarks for which the model explains less than 60% of the variance are not reported.

Chapter 7

Predictive Search Distributions

Estimation of Distribution Algorithms (EDAs) are a popular approach to learn a probability distribution over the "good" solutions to a combinatorial optimisation problem. Here we consider the case where there is a collection of such optimisation problems with learned distributions, and where each problem can be characterised by some vector of features. Now we can define a machine learning problem to predict the distribution of good solutions $q(\mathbf{x}|\mathbf{t})$ for a new problem with features $\mathbf{t}$, where $\mathbf{x}$ denotes a solution. This predictive distribution is then used to focus the search.

The utility of this method is demonstrated on the compiler optimisation task where the goal is to find a sequence of code transformations to make the code run fastest. Results on the SUIF data set of 12 different benchmarks on two distinct architectures show that this approach consistently leads to significant improvements in performance.

This chapter is organised as follows. In section 7.1, the general formulation of a combinatorial optimisation problem is presented. In section 7.2, Estimation of Distribution Algorithms are explained as a common approach for solving combinatorial optimisation problems. In section 7.3, inspired by the general EDA idea, the technique of Predictive Search Distributions is proposed and described as a method for speeding up search by exploiting transference across different optimisation tasks. Work related to predictive search distributions is briefly reviewed in section 7.4. The application of predictive search distributions to the compiler optimisation problem and its results on the SUIF data set are presented in section 7.5. Finally, a summary and a discussion on the use of predictive search distributions are given in section 7.6.

7.1 Motivation: Combinatorial Optimisation

In this chapter we consider optimisation problems and their solution. As input we are given a description T of the optimisation problem, for example the edge weights between all vertices

of a graph for a minimum balanced cut (MBC) problem (see e.g. Andreev and Räcke, 2004)[1]. For any input T there is a set $X(T)$ of valid solutions; for MBC this is the set of bisections that are balanced. We also require an evaluation function f which takes as input a problem description T and a valid solution $\mathbf{x} \in X(T)$, and outputs the quality of the solution. For the MBC problem f is the negative sum of the edge weights in the cut. Our goal is then to find the optimal solution $\mathbf{x}^{\text{opt}}(T)$ such that

$$\mathbf{x}^{\text{opt}}(T) = \text{argmax}_{\mathbf{x} \in X(T)} f(T, \mathbf{x}). \tag{7.1}$$

Another example of an optimisation problem, indeed the one that motivated this work, is in compiler optimisation. For a given benchmark T we can apply a sequence of transformations to the code so as to produce the same input-output behaviour but different run times. As seen in section 2.5, examples of transformations are loop unrolling and common sub-expression elimination. Our goal is to find the sequence of transformations that makes the code run fastest. See section 2.4 for more details on this problem.

7.2 Estimation of Distribution Algorithms: EDAs

Many combinatorial optimisation problems are NP-hard and for such problems it is common to use heuristic optimisation methods, for example those based on population search. Such methods include genetic algorithms which explore the search space by evolving populations of candidate solutions. Estimation of distribution algorithms (EDAs) may be viewed as a way of evolving a probabilistic graphical model $g(\mathbf{x}) \in \mathcal{G}$ describing a distribution of good candidates, rather than evolving a specific population (Pelikan et al., 1999). These methods are particularly popular for addressing combinatorial optimisation problems, although they have also been applied in continuous domains typically, in situations when numerical optimisation was difficult. In the rest of this chapter we will focus specifically on the case when $\{\mathbf{x}\}$ is a space of strings of up to length L defined over finite alphabets.

EDAs are a relatively new idea and one of the first algorithms known as *population-based incremental learning (PBIL)* was proposed by Baluja (1994) as a technique that combined genetic algorithms with competitive learning. This gave rise to different approaches that attempted to model the probability distribution of a search space iteratively within a population-based framework. The concrete concept of estimation of distribution algorithm as a unified framework is presented in Mühlenbein and Paass (1996).

[1] The minimum cut of a graph can be found in polynomial time using network flow methods. However, this algorithm does not guarantee that the two halves are balanced, i.e. that there are equal numbers of nodes in each half. The MBC problem is NP-hard.

7.2.1 General Algorithm

In an EDA, the initial population of size N of the candidate strings $\mathcal{X}^{(0)} \stackrel{\text{def}}{=} \{\mathbf{x}_i|_{i=1}^N\}$ is generated according to some prior distribution (which is usually a uniform distribution). Then at each j^{th} iteration, the algorithm performs the *evaluation* step by computing the objectives $f(\mathbf{x}_i)\ \forall \mathbf{x}_i \in \mathcal{X}^{(j)}$. At the *selection* step, the EDA generates an intermediate subset $\widetilde{\mathcal{X}}^{(j)} \subseteq \mathcal{X}^{(j)}$ of the improved candidate strings, so that candidates $\mathbf{x}_i \in \mathcal{X}^{(j)}$ with higher values of $f(\mathbf{x}_i)$ are more likely to be members of $\widetilde{\mathcal{X}}^{(j)}$. At the *learning* step, the algorithm uses density estimation methods for fitting a distribution $g \in \mathcal{G}$ to the selected set of candidates $\widetilde{\mathcal{X}}^{(j)}$, which gives rise to a new distribution $g^{(j+1)}$. This is followed by *sampling* of the new population $\mathcal{X}^{(j+1)}$ from the learned graphical model (typically it is assumed that the population size is fixed, so that $|\mathcal{X}^{(j+1)}| = |\mathcal{X}^{(j)}| = N$), and the procedure is repeated until a termination criterion is met. After termination, the optimal solution $\mathbf{x}^{\text{opt}}$ is approximated as the $\arg\max_{\mathbf{x}\in\mathcal{X}^{(\infty)}} f(\mathbf{x})$, where $\mathcal{X}^{(\infty)}$ is the final population.

The algorithm for a typical EDA is summarised below. Here N is the size of the population, assumed to be the same at each iteration, and $\mathcal{U}$ denotes the uniform distribution.

1. *initialisation*: constrain the family of the EDA search distributions $\mathcal{G}$; set $j = 0$, N, etc.
2. *generate initial population*: $\mathcal{X}^{(0)} = \{\mathbf{x}_i|_{i=1}^N \sim \mathcal{U}\}$;
3. *evaluation*: $\forall \mathbf{x}_i \in \mathcal{X}^{(j)}$ compute $f(\mathbf{x}_i)$;
4. *selection*: choose a subset $\widetilde{\mathcal{X}}^{(j)} \subseteq \mathcal{X}^{(j)}$ biased towards better-performing solutions; define the empirical distribution $\tilde{p}^{(j)}$ on the subset;
5. *learning*: learn $g^{(j+1)}$ by optimising a discrepancy measure $D(\tilde{p}^{(j)}; g \in \mathcal{G})$ (e.g. KL-divergence);
6. *sampling*: generate $\mathcal{X}^{(j+1)} \stackrel{\text{def}}{=} \{\mathbf{x}_i|_{i=1}^N \sim g^{(j+1)}\}$;
7. iterate steps 3–6 until a termination criterion is met.

After termination, the optimal solution $\mathbf{x}^{\text{opt}}$ is approximated as the $\arg\max_{\mathbf{x}\in\mathcal{X}^{(\infty)}} f(\mathbf{x})$, where $\mathcal{X}^{(\infty)}$ is the final population. We note that the specific EDA that uses the KL-divergence as the discrepancy measure $D(\tilde{p}^{(j)}; g \in \mathcal{G})$ (in step 5) has been referred in the literature as the *cross-entropy* method (Rubinstein and Kroese, 2004).

Usually the models $g \in \mathcal{G}$ are constrained to lie in tractable parametric families and there are a large number of EDAs which are based on specific parameterisations. For example, the population-based incremental learning (PBIL) algorithm assumes that $\mathcal{G}$ is a family of factorised distributions; mutual information-maximizing input clustering (MIMIC) methods constrain $\mathcal{G}$ to be a family of Markov chains, etc. See section 7.2.3 for an overview of specific algorithms. Figure 7.1 illustrates the general idea of an EDA.

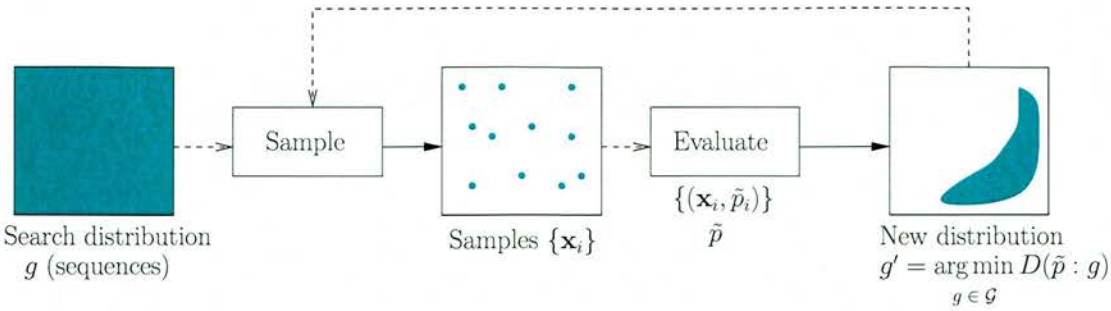

Figure 7.1: A schematic illustration of an Estimation of Distribution Algorithm (EDA).

7.2.2 Defining the Empirical Distribution on Good Solutions

As explained above, an EDA requires the construction of the empirical distribution $\tilde{p}(\mathbf{x})$, which can be obtained for example by placing a uniform distribution over the subset $\widetilde{\mathcal{X}} \subseteq \mathcal{X}$. This subset can be selected (for instance) by thresholding the minimum performance of the observed solutions. For example, one can select those solutions that provide at least 95% of the maximum performance achieved, as given by the maximum observed value of the evaluation function $f(X)$.

Alternatively, We can construct an empirical distribution on good solutions by using a Boltzmann distribution:

$$w^k = \tilde{p}(\mathbf{x}^k) = \frac{e^{\beta f(\mathbf{x}^k)}}{\sum_{u=1}^{|\widetilde{\mathcal{X}}|} e^{\beta f(\mathbf{x}^u)}}, \tag{7.2}$$

where β is the inverse temperature ($\beta = 1/\tau$) and $f(\mathbf{x})$ is our evaluation function (e.g. the speed-up measure in the compiler optimisation problem). This way of constructing the empirical distribution has the feature that the good solutions are weighted according to their evaluation function values.

7.2.3 Families of EDA search Distributions

A key issue in an EDA is the selection of the model distribution $g(\mathbf{x}|\widetilde{\mathcal{X}})$. Obviously, the family of distributions $\mathcal{G}$ to be considered will depend on the complexity of the optimisation problem. Several choices for such distribution have been proposed in the literature depending on the order of interactions that are modelled. Some of these approaches are briefly explained below. The reader is referred to Larrañaga and Lozano (2001, Chapter 2) and Pelikan et al. (1999) for a more comprehensive review of EDAs. For simplicity in the notation we will omit the dependency of the model distribution on the selected subset of solutions $\widetilde{\mathcal{X}}$ by simply writing $g(\mathbf{x})$.

Identically Factorised Distributions

One of the simplest choices of the model distribution for an EDA is a factorised distribution $g(\mathbf{x}) = \prod_i g(x_i)$. Fitting such distribution is straightforward (by using counts). This has been proposed in Mühlenbein (1997) under the name of the Univariate Marginal Distribution Algorithm (UMDA).

Similar in spirit to the UMDA, the well-known Population-Based Incremental Learning algorithm (PBIL; Baluja, 1994; Baluja and Caruana, 1995) characterises the solution of an optimisation problem with a set of binary strings and the population of solutions is represented with a probability vector $g(\mathbf{x}) = (g(x_1), \ldots, g(x_L))$. Initially, $g(x_i) = 0.5, i = 1, \ldots, L$ and the probability vector is updated according to the rule $g^{j+1}(\mathbf{x}) = (1-\alpha)g^j(\mathbf{x}) + \alpha\frac{1}{|\widetilde{\mathcal{X}}|}\sum_{\mathbf{x}_i \in \widetilde{\mathcal{X}}} \mathbf{x}_i$, where α is the learning rate and $|\widetilde{\mathcal{X}}|$ is the number of solutions selected.

Markov-chain Distributions

Identically factorised distributions may be a very limited approach to modelling the distribution of good solutions on an optimisation problem as they assume that the variables that characterise these solutions are statistically independent. Therefore, one can also consider pairwise interactions as proposed by de Bonet et al. (1997) in their Mutual-Information-Maximizing Input Clustering (MIMIC) algorithm. Here the model distribution is a Markov chain: $g(\mathbf{x}) = g(x_1)\prod_{i=2}^{L} g(x_i|x_{i-1})$, which is learnt in order to minimise the KL-divergence between the chain and the empirical distribution of the selected points. In order to find the optimal variable permutation to construct the chain, de Bonet et al. (1997) propose a greedy search algorithm that adds one variable at a time so as to maximise the conditional entropy with respect to the previously selected variables.

More Complex Distributions

Baluja and Davies (1997) propose the use of dependency *trees* in order to characterise higher-order interactions in the model distribution of an EDA. Their approach is based upon the algorithm proposed by Chow and Liu (1968), which is used to find the optimal model within this class of distributions. The work in Baluja and Davies (1997) is extended in Baluja and Davies (1998), where the COMIT (Combining Optimizers with Mutual Information Trees) algorithm is proposed as a way of combining the tree-based EDA approach with multiple local optimisers.

A different approach to modelling high-order dependencies is given in the Factorized Distribution Algorithm (FDA) proposed by Mühlenbein and Mahnig (1999). Here the model distribution factorises as the product of k multivariate conditional distributions. A major caveat of this approach is that such factorisation must be provided by an expert requiring knowledge of the structure of the problem. This latter drawback is overcome by Pelikan et al. (2000),

Algorithm	Structure	Author(s)
UMDA	iid variables	Mühlenbein (1997)
PBIL	iid variables	Baluja (1994)
MIMIC	Markov chain	de Bonet et al. (1997)
COMIT	Trees	Baluja and Davies (1997)
FDA	Product of distributions	Mühlenbein and Mahnig (1999)
BOA	Bayesian Network (BN)	Pelikan et al. (2000)
hBOA	hierarchical BN	Pelikan and Goldberg (2001)

Table 7.1: Examples of Estimation of Distribution Algorithms.

who propose their Bayesian Optimization Algorithm (BOA) in which the model distribution is a Bayesian network. In order to measure the quality of the networks they use the Bayesian Dirichlet (BD) metric (Heckerman et al., 1995). The work in Pelikan et al. (2000) is further extended to handle local structures in the Bayesian network, more specifically decision trees. Such algorithm has been called the hierarchical BOA (Pelikan and Goldberg, 2001). For a real-life application of this algorithm see e.g. Pelikan and Goldberg (2003). Table 7.1 summarises some of the EDAs proposed in the literature according to the model distribution used.

7.2.4 Learning an EDA Search Distribution

Here we want to learn a distribution $g(\mathbf{x}) \in \mathcal{G}$ that is as close as possible to the empirical distribution $\tilde{p}(\mathbf{x})$. We can achieve this for example by minimising the KL-divergence:

$$\begin{aligned} \mathrm{KL}(\tilde{p}(\mathbf{x}), g(\mathbf{x})) &= \left\langle \log \frac{\tilde{p}(\mathbf{x})}{g(\mathbf{x})} \right\rangle_{\tilde{p}(\mathbf{x})} && (7.3) \\ &= \sum_u \tilde{p}(\mathbf{x}^u) \log \tilde{p}(\mathbf{x}^u) - \sum_u \tilde{p}(\mathbf{x}^u) \log g(\mathbf{x}^u) && (7.4) \\ &= -H(\tilde{p}(\mathbf{x})) + H(\tilde{p}(\mathbf{x}), g(\mathbf{x})), && (7.5) \end{aligned}$$

where $H(\tilde{p}(\mathbf{x}))$ is the entropy of $\tilde{p}(\mathbf{x})$ and $H(\tilde{p}(\mathbf{x}), g(\mathbf{x}))$ is the cross-entropy of $\tilde{p}(\mathbf{x})$ and $g(\mathbf{x})$. Clearly, in order to minimise the above equation we focus on minimising the cross-entropy term since $H(\tilde{p}(\mathbf{x}))$ is a constant.

By minimising the cross-entropy term (which is equivalent to maximising the log-likelihood) we have:

$$\mathcal{L} = -H(\tilde{p}(\mathbf{x}), g(\mathbf{x})) = \sum_u \tilde{p}(\mathbf{x}^u) \log g(\mathbf{x}^u). \quad (7.6)$$

An iid Distribution

For the iid distribution $g(\mathbf{x}) = \prod_{i=1}^{L} g(x_i)$, equation (7.6) can be written as:

$$\mathcal{L} = \sum_u \tilde{p}(\mathbf{x}^u) \sum_{i=1}^{L} \log g(x_i^u), \tag{7.7}$$

with $g(x_i)$ being a multinomial distribution: $g(x_i) = (\theta_i^1)^{I[x_i=a_1]}(\theta_i^2)^{I[x_i=a_2]} \ldots (\theta_i^{|\mathcal{A}|})^{I[x_i=a_{|\mathcal{A}|}]}$, where $\theta_i^j = p(x_i = a_j)$; $I[x_i = a_j]$ is an indicator function that is 1 when $x_i = a_j$ and zero otherwise; $|\mathcal{A}|$ is the total number of possible states; and $\sum_j \theta_i^j = 1$.

Hence, we have that:

$$\mathcal{L} = \sum_u \tilde{p}(\mathbf{x}^u) \sum_{i=1}^{L} \sum_{j=1}^{|\mathcal{A}|} I[x_i^u = a_j] \log \theta_i^j. \tag{7.8}$$

Maximising equation (7.8) with respect to some parameter θ_k^n subject to the constraints $\sum_j \theta_i^j = 1$ is straightforward (by using Lagrange multipliers). Thus, we obtain:

$$\theta_k^n \sum_u \tilde{p}(\mathbf{x}^u) \sum_j I[x_k^u = a_j] = \sum_u \tilde{p}(\mathbf{x}^u) I[x_k^u = a_n], \tag{7.9}$$

where it is clear that $\sum_j I[x_k^u = a_j] = 1$ and consequently $\sum_u \tilde{p}(\mathbf{x}^u) \sum_j I[x_k^u = a_j] = 1$. Therefore:

$$\theta_k^n = \sum_u \tilde{p}(\mathbf{x}^u) I[x_k^u = a_n]. \tag{7.10}$$

This result, known as the maximum likelihood estimator, is intuitive and the particular case of the empirical distribution $\tilde{p}(\mathbf{x})$ being the uniform distribution yields the estimation of the parameters of the iid distribution θ_k^n as the number of (selected) solutions in which $x_k = a_n$ over the total number of (selected) solutions.

However, in order to provide smoother estimates and avoid the zero frequency problem, i.e. obtaining a zero value for $g(\mathbf{x})$ when the pattern $x_k = a_n$ does not appear in the data, it is customary to add *pseudocounts* α_n to the number of data-points that have been actually seen. These pseudocounts can be seen as the (hyper-)parameters of a Dirichlet prior when following a Bayesian approach.

If we consider a *stationary* iid distribution $g(\mathbf{x})$ for which the same set of parameters is used across all the positions of the sequence, i.e. $g(x_i) = \prod_{j=1}^{|\mathcal{A}|} (\theta^j)^{I[x_i=a_j]}$ for $i = 1, \ldots, L$, we have that the parameters $\theta^n = p(x_i = a_n)$ for $i = 1, \ldots, L$ can be estimated by using:

$$\theta^n = \frac{\sum_u \tilde{p}(\mathbf{x}^u) \sum_{i=1}^{L} I[x_i^u = a_n]}{\sum_u \tilde{p}(\mathbf{x}^u) \sum_{i=1}^{L} \sum_{j=1}^{|\mathcal{A}|} I[x_i^u = a_j]} \tag{7.11}$$

$$= \frac{1}{L} \sum_u \tilde{p}(\mathbf{x}^u) \sum_{i=1}^{L} I[x_i^u = a_n]. \tag{7.12}$$

As above, this result is intuitive when $\tilde{p}(\mathbf{x})$ is the uniform distribution since such estimator corresponds to the proportion of times the literal a_n appears in the selected solutions.

A Markov Distribution

For a Markov chain distribution $g(\mathbf{x}) = g(x_1)\prod_{i=2}^{L} g(x_i|x_{i-1})$ we have that

$$\mathcal{L} = \underbrace{\sum_u \tilde{p}(\mathbf{x}^u)\log g(x_1^u)}_{\mathcal{L}_1} + \underbrace{\sum_u \tilde{p}(\mathbf{x}^u)\sum_{i=2}^{L}\log g(x_i^u|x_{i-1}^u)}_{\mathcal{L}_2}, \tag{7.13}$$

with $g(x_1) = \prod_{j=1}^{|\mathcal{A}|}(\theta_1^j)^{I[x_1=a_j]}$ and $g(x_i|x_{i-1}) = \prod_{j=1}^{|\mathcal{A}|}\prod_{k=1}^{|\mathcal{A}|}(\theta_i^{jk})^{I[x_{i-1}=a_j]I[x_i=a_k]}$, with parameters $\theta_1^j = p(x_1 = a_j)$ and $\theta_i^{jk} = p(x_i = a_k|x_{i-1} = a_j)$. As above, $I[x_i = a_j]$ is an indicator function that is 1 when $x_i = a_j$ and zero otherwise and $|\mathcal{A}|$ is the total number of possible states. Additionally, $\sum_j \theta_1^j = 1$ and $\sum_k \theta_i^{jk} = 1$.

Maximising equation (7.13) with respect to the parameters θ_1^n and θ_ℓ^{mn} can be done by independently maximising $\mathcal{L}_1$ with respect to θ_1^n and maximising $\mathcal{L}_2$ with respect to θ_ℓ^{mn}. By following the same procedure as the one adopted for the iid distribution, the parameters that maximise $\mathcal{L}_1$ are given by:

$$\theta_1^n = \sum_u \tilde{p}(\mathbf{x}^u)I[x_1^u = a_n];\ \mathrm{n} = 1,\ldots|\mathcal{A}|. \tag{7.14}$$

Now we want to maximise:

$$\mathcal{L}_2 = \sum_u \tilde{p}(\mathbf{x}^u)\sum_{i=2}^{L}\sum_{j=1}^{|\mathcal{A}|}\sum_{k=1}^{|\mathcal{A}|} I[x_{i-1}^u = a_j]I[x_i^u = a_k]\log\theta_i^{jk}, \tag{7.15}$$

with respect to some parameter θ_ℓ^{mn} subject to the constraints $\sum_k \theta_i^{jk} = 1$. Thus, (by using Lagrange multipliers) we obtain:

$$\theta_\ell^{mn}\sum_u \tilde{p}(\mathbf{x}^u)\sum_k I[x_{\ell-1}^u = a_m]I[x_\ell^u = a_k] = \sum_u \tilde{p}(\mathbf{x}^u)I[x_{\ell-1}^u = a_m]I[x_\ell^u = a_n]. \tag{7.16}$$

Rearranging terms and considering that $\sum_k I[x_\ell^u = a_k] = 1$, we finally obtain:

$$\theta_\ell^{mn} = \frac{\sum_u \tilde{p}(\mathbf{x}^u)I[x_{\ell-1}^u = a_m]I[x_\ell^u = a_n]}{\sum_u \tilde{p}(\mathbf{x}^u)I[x_{\ell-1}^u = a_m]}. \tag{7.17}$$

By considering the particular case of $\tilde{p}(\mathbf{x})$ being the uniform distribution we obtain the intuitive result that the parameters of the Markov chain distribution $\theta_\ell^{mn} = p(x_\ell = a_n|x_{\ell-1} = a_m)$ can be estimated by dividing the number of solutions for which $x_{\ell-1} = a_m$ and $x_\ell = a_n$ over the number of solutions for which $x_{\ell-1} = a_m$. As on the iid case, one can add pseudocounts to the number of solutions in which such patterns have been observed.

If we consider a *stationary* Markov chain for which the same transition probabilities are used across all the positions of the sequence, i.e. $g(x_i|x_{i-1}) = \prod_{j=1}^{|\mathcal{A}|}\prod_{j=1}^{|\mathcal{A}|}(\theta^{jk})^{I[x_{i-1}=a_j]I[x_i=a_k]}$, we can estimate the parameters $\theta^{mn} = p(x_i = a_n|x_{i-1} = a_m)$ for $i = 2,\ldots,L$ by using:

$$\theta^{mn} = \frac{\sum_u \tilde{p}(\mathbf{x}^u)\sum_{i=2}^{L} I[x_{i-1}^u = a_m]I[x_i^u = a_n]}{\sum_u \tilde{p}(\mathbf{x}^u)\sum_{i=2}^{L} I[x_{i-1}^u = a_m]\sum_{k=1}^{|\mathcal{A}|} I[x_i^u = a_k]} \tag{7.18}$$

$$\theta^{mn} = \frac{\sum_u \tilde{p}(\mathbf{x}^u)\sum_{i=2}^{L} I[x_{i-1}^u = a_m]I[x_i^u = a_n]}{\sum_u \tilde{p}(\mathbf{x}^u)\sum_{i=2}^{L} I[x_{i-1}^u = a_m]}. \tag{7.19}$$

We note that we estimate θ_1^n using equation (7.14). As above, the estimation of the transition probabilities, i.e. θ^{mn}, is intuitive as it relies upon counting the number of transitions $a_m \rightarrow a_n$ in the selected solutions over the number of times that a_m appears on the first $L-1$ positions.

7.2.5 Improving the Performance of EDAs

While there is some empirical evidence that more complex EDAs outperform simpler EDAs for some tasks, there are studies reporting no improvements in other situations (see e.g. Johnson and Shapiro, 2001). Guiding search with more complex distribution families may be justified in cases where such families provide an exact match to the search problems. (However, the fact that good solutions may be well explained by some distribution $g \in \mathcal{G}$ does not necessarily mean that such distributions will be easy to find, especially for very large search problems.) Nevertheless, in practice it is rarely the case that we know exact characterisations of good distributions, i.e. formally arguing for a specific choice of the search distribution G for a specific task may in general be very difficult.

In addition to increasing the representational complexity of the distributions, it is important to address other ways of improving the performance of EDA methods. Shapiro (2003, 2005) suggests that a fundamental flaw of EDAs is the lack of ergodicity, which biases the search towards previously seen parts of the search space even when there is no fitness-supported evidence that this could help to generate better solutions. Intuitively, this happens because EDAs do not typically compensate for the finite sampling effects. Indeed, if there were no selection, the sample variance would eventually decay to zero (this is analogous to the drift effects in population genetics). Without selection, any changes in the resulting candidates throughout the iterations of the algorithm will essentially be caused by random fluctuations, independently of the specifics of a search problem. In a real EDA (when selection does take place), there will be a trade-off between the random drift and the bias towards better solutions, which will depend on a specific task and optimisation heuristics.

7.3 Predictive Search Distributions: PSD

We have seen how EDAs can be used to solve a single combinatorial optimisation problem. Imagine now that we have a collection of examples of the same kind of problem, but with different descriptions T drawn from a space $\mathcal{T}$. In this case if we have solved the optimisation problems we will have a set of solutions $(T_1, \mathbf{x}^{\text{opt}}(T_1)), (T_2, \mathbf{x}^{\text{opt}}(T_2)), \ldots, (T_M, \mathbf{x}^{\text{opt}}(T_M))$. Here $\mathbf{x}^{\text{opt}}(T_i)$ could be an approximate solution to the problem rather than the true global optimum. One idea to apply machine learning to this optimisation problem is to learn the mapping between T and $\mathbf{x}^{\text{opt}}(T)$ based on examples. However, this may be a very difficult mapping to learn; one problem is that small changes to the input T of a combinatorial optimisation problem

may lead to very different solutions.

We will take a rather different approach, mapping from a problem description T to a probability distribution $q(\mathbf{x}|T)$ over good solutions. Our motivation is that it may be very hard to predict the optimal solution $\mathbf{x}^{\text{opt}}(T)$ for an instance T, but that it may well be easier to define a search distribution which gives high probability to $\mathbf{x}^{\text{opt}}(T)$.

An inspiration for this idea is recent work on EDAs where for a given problem T a search over solutions is conducted by repeatedly re-estimating a probability distribution over good solutions (see section 7.2 for more details on EDAs). However, note that EDAs are concerned with finding an optimal solution $\mathbf{x}^{\text{opt}}(T)$ for a given problem T, while our goal is to make use of what has been learned from previous problems in predicting the search distribution for a new instance.

Our goal of mapping from T to a predictive distribution $q(\mathbf{x}|T)$ is actually similar to the standard probabilistic machine learning set up where the output is a predictive distribution, e.g. a Bernoulli probability or a univariate Gaussian distribution. However, we note two special aspects of modelling search distributions:

- The distribution $q(\mathbf{x}|T)$ is meant to *focus* our search. Thus we might well make multiple draws from it in the hope of finding better solutions.
- The predictive distribution $q(\mathbf{x}|T)$ might be complicated (e.g. it might be a graphical model); this kind of complexity is not commonly used in standard machine learning situations, although it does arise, e.g. in conditional random fields (Lafferty et al., 2001).

The use of multiple EDA runs in order to learn a single search distribution across different optimisation problems is explained in Section 7.3.1. Learning and predictions with PSD are explained in sections 7.3.2 and 7.3.3 respectively. Finally, a brief discussion on model selection in PSD is given in section 7.3.4.

7.3.1 From Multiple EDAs to a Single Predictive Distribution

As explained in section 7.2.1, at the end of the EDA run on problem T_i we have $\tilde{p}^{(\infty)}(\mathbf{x}|T_i)$, which is the empirical distribution over solutions. Additionally, we have $g^{(\infty)}(\mathbf{x}|T_i)$, which is the model (output) distribution. We can use either of these distributions in order to learn a single distribution across a set of optimisation problems $T_1, \ldots, T_M$.

The Need for a Common Representation

The key issue that we now address is how to make predictions on a new problem instance T given a training set of problems and their corresponding search (or empirical) distributions. We assume that for any problem description $T \in \mathcal{T}$ we can extract a number of features $\mathbf{t}$

that (partially) characterise the problem. Extracting a common representation for different optimisation problem instances can be a hurdle since there may be a lot of variation within an optimisation problem class. However, for tasks such as the compiler optimisation problem these features can appear naturally (e.g. code features of a program). Additionally, there is some evidence that even for common and challenging problems in optimisation, it is possible to extract a set of meaningful features across different instances, for example by computing statistics of different characteristics of the problems, see e.g. Zhang and Dietterich (1995).

7.3.2 Learning PSD

In PSD we seek to learn a predictive distribution $q(\mathbf{x}|\mathbf{t},\boldsymbol{\theta})$ that outputs a distribution over solutions $\mathbf{x}$ given an input $\mathbf{t}$ and parameters $\boldsymbol{\theta}$. We give two approaches to learning the predictive distribution based on (a) memory-based methods and (b) maximisation of the conditional likelihood.

Memory-Based Methods

A possible approach to learning PSD is to use memory-based methods such as k nearest-neighbours (k-NN). For example, using 1-NN we can set the predictive search distribution to be the EDA distribution of the training problem whose features are the closest to the new problem. For $k > 1$ we could use a mixture of distributions from the k closest neighbours. This method may be particularly useful if the number of training problems M is small. In our experiments on the compiler optimisation problem, given the limited amount of training programs in the SUIF data set, we have learned PSD using a 1-NN approach (see section 7.5.5 for more details).

Maximisation of the Conditional Likelihood

The conditional likelihood of the parameters $\boldsymbol{\theta}$ given the data can be expressed as:

$$\mathcal{L}_{\mathbf{x}|T} \stackrel{\text{def}}{=} \sum_{i=1}^{M} \sum_{\mathbf{x}} \tilde{p}^{(\infty)}(\mathbf{x}|T_i) \log q(\mathbf{x}|\mathbf{t}_i,\boldsymbol{\theta}), \tag{7.20}$$

where $\tilde{p}^{(\infty)}(\mathbf{x}|T_i)$ is the empirical distribution over solutions output at the end of the EDA run on problem T_i. An alternative objective function would be

$$\sum_{i=1}^{M} \sum_{\mathbf{x}} g^{(\infty)}(\mathbf{x}|T_i) \log q(\mathbf{x}|\mathbf{t}_i,\boldsymbol{\theta}). \tag{7.21}$$

This is similar to (7.20) but uses the output EDA distribution $g^{(\infty)}(\mathbf{x}|T_i)$ for each problem rather than the empirical distribution. However, in cases where the family of distributions $\mathcal{G}$ in the EDA has hidden variables (e.g. a HMM) then (7.20) should be easy to evaluate while (7.21) may not be.

In practice we can consider a number of parametric and/or structural constraints on the family of predictive distributions $q(\mathbf{x}|\mathbf{t},\boldsymbol{\theta})$. Since in our case the distribution is defined over strings, choices for $q(\mathbf{x}|\mathbf{t},\boldsymbol{\theta})$ include logistic regression and conditional random field models (Lafferty et al., 2001).

7.3.3 Predictions with PSD

Having learnt the distribution over good solutions $q(\mathbf{x}|\mathbf{t},\boldsymbol{\theta})$ across different optimisation problems, making predictions on a new problem T is straightforward as we will be using $q(\mathbf{x}|\mathbf{t},\boldsymbol{\theta})$ to guide search on this problem. In other words, we can make multiple draws from this distribution with the hope of finding better solutions. Additionally, note that as we see more data points on the new problem T we can gradually update our model towards these solutions in an EDA-based fashion.

It is also worth remarking that the predictive search distribution methodology may not always lead to speed-ups, and could in principle degrade performance relative to uniform search. This would be the case if the features extracted did not provide useful information about the search distribution, or if $g(\mathbf{x}|T)$ varies very rapidly with changes in the input T so that a very large amount of training data would be needed to characterise the problem class. However, for the compiler optimisation problems studied, we have found that the technique leads to significant improvements over uniform search. See section 7.5 for more details.

7.3.4 Choosing a Family of PSD: Model Selection

Learning predictive search distributions can be seen as a standard machine learning problem where we aim to learn a distribution over distributions on good solutions to different instances of an optimisation task. Therefore, choosing a family of PSD to which $q(\mathbf{x}|\mathbf{t},\boldsymbol{\theta})$ should be constrained is a model selection problem. Several approaches have been proposed in the machine learning and statistics literatures for model selection or model comparison. Given two models $\mathcal{M}_0$ and $\mathcal{M}_1$, we want to select one model on the basis of some previously seen data $\mathcal{D}$. Furthermore, we wish our selected model to perform well not only on the training data $\mathcal{D}$ but also on data that we have not seen before.

One possible solution to the model selection problem is to use cross-validation. However, this may be a very time-consuming activity as we may require a lot of runs (e.g. in leave-one-out cross-validation or in k-fold cross-validation) and our models can be quite expensive to evaluate. Other frequentist approaches such as the AIC criterion (Akaike, 1974), the Bayesian information criterion (BIC, Schwarz, 1978) and the likelihood ratio test (see e.g. Lindgren, 1993, section 10.2) can be used. These approaches select a model based upon the maximisation of the conditional likelihood and they may penalise (e.g. AIC or BIC) overly complex models .

Alternatively, one can adopt a fully Bayesian approach such as the Bayes factors (Jeffreys, 1966; Kass and Raftery, 1995), in which one marginalises out (or average over) the parameters of the model instead of using their maximum likelihood estimates as in the AIC or BIC criteria. This can be seen as a way of avoiding overfitting. A possible drawback of the Bayesian approach is that one often needs to compute intractable integrals and therefore numerical approximations are required. Additionally, a key issue in Bayesian model selection is the choice of the prior distribution over the parameters $p(\boldsymbol{\theta}^{(k)}|\mathcal{M}_k)$. As the marginal likelihood (or evidence) is sensitive to such prior, one should be careful with its selection in order to avoid misleading results.

The reader is referred to e.g. Bishop (2006, section 3.4) for an overview of model selection criteria in machine learning.

7.4 Related Work

We are not aware of much previous work in this area. One common way to help speed up search problems is through memoization, i.e. remembering the answers to previous problems (or partial problems) so as to eliminate search if they are encountered again. The proposed method goes beyond this in that it affords inductive generalisation to new search problems rather than simply storing earlier results.

There has also been a lot of work on learning search-control knowledge, see e.g. Langley (1996, chapter 10) for an overview. This work focuses on planning in sequential decision problems (SDPs), i.e. the task of reaching a goal state from a start state; this general area is addressed by reinforcement learning. There has also been work on speeding up search using explanation-based learning (EBL) using solution traces and a domain theory. However, the search problems we are addressing are not SDPs and so this work does not apply.

One possible alternative approach is to learn a regression function $\hat{f}$ that approximates $f(T,\mathbf{x})$ from data samples over the $\mathcal{T} \times \mathcal{X}$ input space. Such an approach will be described in chapter 8 when using Gaussian process predictors. However, if the space of solutions $\mathcal{X}$ is very large then even if this proxy function can be learned accurately it would be very time consuming to find the string $\mathbf{x}$ that optimises $\hat{f}(T,\mathbf{x})$; thus in many situation we may prefer an approach that directly outputs a distribution of "good" solutions.

Over the past few years, there has been a lot of interest in the topic of *inductive transfer*, see e.g. Thrun and O'Sullivan (1996) and Caruana (1997) as some of the earlier references. In these (and later) papers a common set up is that there are multiple, related supervised learning problems and that the goal is to avoid tabula rasa learning for a new problem by extracting information from the problems seen before. Thus the learning is at a higher level, e.g. by using the previously seen problems to define priors on parameters for the new problem. Our sugges-

tion for learning search distributions is actually more like the standard probabilistic machine learning set up at the lower level, where the prediction is a probability distribution.

In a general framework for dealing with combinatorial optimisation problems, Horvitz et al. (2001) propose *a Bayesian approach to tackling hard combinatorial optimization problems.* They employ Bayesian learning procedures to build probabilistic models that capture the dependencies between the observations in a given problem and their probability distributions over run time. In other words, they build models that can predict the run time of problem solvers. More specifically, features that can capture patterns and dynamics of the state of problem solvers are formulated. These features are used in a Bayesian-structure learning framework in order to infer predictive models able to forecast whether the run time to reach a solution on a problem instance will be "short" or "long". This labelling is performed with respect to the median of all the training run times.

Based upon this latter work, Kautz et al. (2002) formulate *Dynamic Restart Policies* as a solution for *speeding-up backtracking search.* They tackle the problem of deciding whether a search algorithm should stop a particular run and restart the execution after some randomisation. They show the benefits of their approach on a set of hard combinatorial optimisation problems. In general, it seems appealing to apply such an idea to combinatorial optimisation problems. However, for applications where the evaluation function is very costly to execute such as the compiler optimisation problem, directly modelling distributions over good solutions seems to be a more practical way of speeding up search.

Zhang and Dietterich (1995) apply reinforcement learning to the Job-shop Scheduling problem. In this work, a value function is learnt over a set of problems and it is then used in a "one-step look ahead search procedure" to find solutions on new scheduling problems. This is closer in spirit to our idea of achieving transference in search. However, in general, reinforcement learning is concerned with maximising the *expected* reward on the long run. This is not the case for many optimisation problems such as the compiler optimisation problem, where given a search path on states (or problem description) which is obtained by a sequential application of actions, one aims at finding the *maximum* possible value along such path.

7.5 Compiler Optimisation with PSD

This section describes the application of predictive search distributions (PSD) to the problem of compiler optimisation and presents the experiments that were carried out with the goal of improving iterative compiler optimisation. The results presented in this section are obtained on the SUIF data set. See chapter 5 for details on the experimental set up under which this data set has been generated.

Sections 7.5.1 to 7.5.5 explain how the problem of compiler optimisation is addressed

with predictive search distributions and provide details of the evaluation function, distributions, features and evaluation set-up used. Sections 7.5.6 and 7.5.7 give the results for the small space and the large space of the SUIF data set respectively, when evaluating the PSD technique with respect to uniform search. Section 7.5.8 presents the results for other baselines and shows that the PSD method outperforms these baselines. Finally, Section 7.5.9 analyses the distributions learned for each program and section 7.5.10 shows the performance of the PSD technique for different values of training samples.

7.5.1 Formulation of the Problem

As described in section 2.4, *Compiler optimisation* deals with the problem of making a compiler produce better code, i.e. code that runs fast. Numerous program transformations have been proposed in the literature and implemented in commercial and research compilers for this purpose. However, it is difficult to know when or how a compiler should apply these transformations to a specific program. Additionally, the effect of interactions between program transformations makes the problem of producing optimal code even harder.

An interesting scenario in compiler optimisation is *iterative compilation*, where one can afford several program executions in order to determine a set of program transformations that significantly increase performance. This task can be formulated as a combinatorial optimisation problem where a set of transformations can be combined into sequences of arbitrary length. This approach of searching the space of transformation sequences has been shown to provide excellent performance at the cost of a large number of evaluations of a program (Franke et al., 2005).

Considering that similar programs may have similar behaviour under the application of several code transformations, we propose Predictive Search Distributions for improving iterative optimisation. Making explicit the notation used throughout this chapter, $\mathbf{x}$ represents a sequence of code transformations; $\mathbf{t}$ is a set of features extracted from a program; and $q(\mathbf{x}|\mathbf{t})$ is the predictive distribution over good transformation sequences given some program features. Thus, our goal is to learn $q(\mathbf{x}|\mathbf{t})$ based on distributions over good solutions on training programs $T_1, \ldots, T_M$, and use the predictive distribution to guide search on a new program that has not been seen before.

7.5.2 Evaluation Function

In order to evaluate the quality of a transformation sequence we use the speed-up (see section 2.7) as a measure of performance. As shown in Table 6.1, significant speed-ups have been obtained on average for both platforms and most benchmarks can be improved with the experiments. These results are important as they show that good improvements can be obtained with iterative compilation and that the data generated presents opportunities for learning. Thus, it

makes sense to use techniques such as Predictive Search Distributions in order to focus search over good subspaces of transformation sequences.

7.5.3 Distributions Used

Two classes of distributions have been fitted to the set of good transformation sequences on each program. For the results presented in this section we have defined a good transformation sequence as a sequence that has an improvement in performance of at least 95% of the maximum improvement achieved.

The first distribution class that has been used is an **iid** distribution, where the transformations within a sequence are considered independent, so that

$$g(x_1, x_2, \ldots, x_L) = \prod_{i=1}^{L} g(x_i), \tag{7.22}$$

where L is the length of the sequence. This iid model neglects the effect of interactions among transformations, which can be very restrictive as some transformations enable the applicability of others, and there are transformations that yield good performance only when others have been previously applied.

Bearing in mind that more complex models can involve a much greater number of parameters, a stationary **Markov** chain has been used as the second model to focus search. In this model, the probability of a transformation being applied in a specific position of a sequence depends on the transformation that has been previously applied, so that

$$g(x_1, x_2, \ldots, x_L) = g(x_1) \prod_{i=2}^{L} g(x_i | x_{i-1}). \tag{7.23}$$

Note that searching with this Markov distribution differs from the MIMIC algorithm (de Bonet et al., 1997), which uses a non-stationary Markov chain.

Both distributions have been fitted by maximum likelihood estimation when the parameters are shared across the different positions of the sequence (i.e. we have used stationary distributions) as explained in section 7.2.4 and as shown in equation (7.12) for the iid distribution and equations (7.14) and (7.19) for the Markov distribution. Additionally, we have used pseudocounts of 0.001[2].

7.5.4 Program Features

As in any other machine learning task, a significant difficulty for applying PSD to the compiler optimisation problem is to extract relevant **features** for learning. For this purpose we have relied on the knowledge of compiler experts who have identified thirty-four program features

[2]In our experiments we have noticed that the results presented for the small space are insensitive to pseudocounts between 10^{-7} and 1.

Program Features	
Binary	**Integer**
All loop indices are constants?	Loop nest depth
Array is accessed in a non-linear manner?	Loop step within for loop
Both int and floats used in loop?	No. of other instructions in loop
Loop strides on leading array dimensions only?	No. of store instructions in loop
For loop has constant lower bound?	No. of array instructions in loop
For loop has constant stride?	No. of branch instructions in loop
For loop has constant upper bound?	No. of call instructions in loop
For loop has unit stride?	No. of array references within loop
For loop is nested?	No. of compare instructions in loop
For loop is perfectly nested?	No. of divide instructions in loop
For loop is simple?	No. of float variables in loop
Loop contains an if statement in for-construct?	No. of generic instructions in loop
Loop contains an if-construct?	No. of instructions in loop
Loop has branches?	No. of integer variables in loop
Loop has calls?	No. of iterations in for loop
Loop has regular control flow?	No. of load instructions in loop
Loop iterator is an array index?	No. of memory copy instructions in loop

Table 7.2: Program features used on the application of Predictive Search Distributions to the compiler optimisation problem. For each program, binary features are counted (added) and integer features are averaged across loops.

believed to describe the characteristics of a program well and to be relevant for our specific task. The number of instructions in loops and the number of array references in loops are examples of such features. A complete list of the features is shown in Table 7.2. These features have been aggregated (added or averaged) across loops.

7.5.5 Learning Methods and Evaluation Set Up

Given the high-dimensional search space and the limited amount of training data (only twelve benchmarks), it seems rather difficult to learn search distributions for our set of programs. Therefore, we have reduced the dimensionality of the input to five features by using PCA (while retaining 99% of the data variance) and tested our approach with **1-nearest neighbour** predictor in a Leave-One-Out Cross-Validation (LOOCV) procedure. Thus, we have predicted the search distribution for a program by simply using the distribution of its nearest neighbour.

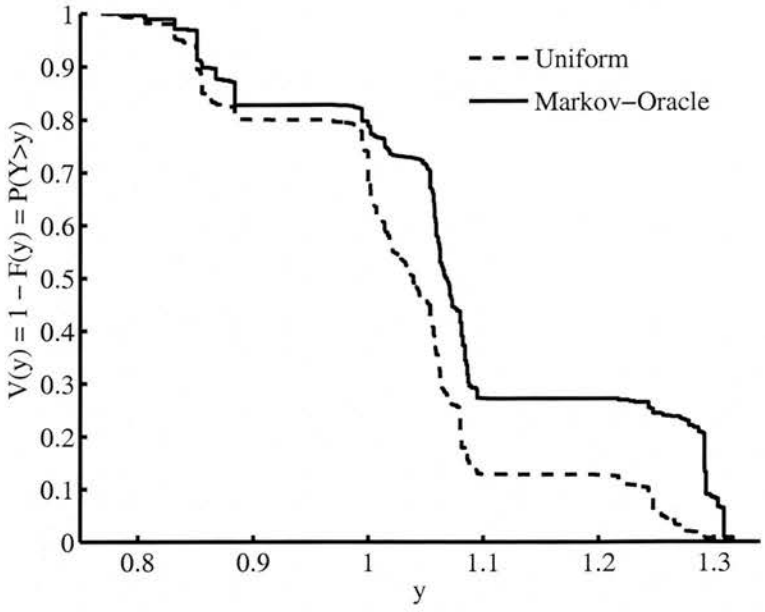

Figure 7.2: Survival function of performance speed-ups for uniform distribution and Markov-oracle distribution for the benchmark *adpcm* on the TI architecture for the small space of the SUIF data set of 14^5 sequences.

7.5.6 Evaluation on the Small Space of the SUIF Data Set

Before presenting the results for the predictive search distributions we want to evaluate the potential of the method by fitting the IID and the Markov models to the true distribution of "good" solutions. Let us call these distributions the *IID-oracle* and the *Markov-oracle*. They are oracles in the sense that they provide an upper bound on our expectations of speeding up search by our learned models. Therefore, our first experiment aims to show that these oracles do improve search. Clearly, if this is not the case, it is not worth expending effort trying to learn these distributions.

Let $V(y) = 1 - F(y)$ be the survival function (or complementary cumulative distribution function) of performance speed-ups y achieved with a given search distribution, where $F(y)$ is the cumulative distribution function of speed-ups y. By definition, $V(y)$ is the probability of a speed-up being greater than y, i.e. $P(Y > y)$. Therefore, greater values of $V(y)$ for near-optimal speed-ups translate into higher probabilities of finding good transformation sequences. Indeed, the ideal case would occur if all the density mass was concentrated on the maximum speed-up $y^{\text{opt}} = \max_{y \in \mathcal{Y}}(y)$ so $V(y)$ would be one for all $y < y^{\text{opt}}$ but zero for $y = y^{\text{opt}}$.

Given a set of target values (e.g. speed-ups) $\{y^i\}_{i=1}^N$ corresponding to the sequences $\{\mathbf{x}^i\}_{i=1}^N$, each associated with a probability of being sampled p^i (with $\sum_i p^i = 1$), the cumulative distribution function $F(y)$ can be computed by sorting the target values $\{y^i\}_{i=1}^N$ in ascending order and calculating the cumulative sum of the probabilities corresponding to such order. In other

words, for each (sorted) speed-up y^i one has a corresponding value $F(y^i) = \sum_{j=1}^{i} p^j$. For a general search distribution $g(\mathbf{x})$ we use $p^i = g(\mathbf{x}^i)/\sum_{j=1}^{N} g(\mathbf{x}^j)$. For the uniform search distribution $g(\mathbf{x}^i) = 1/N$. For the iid and Markov distributions $g(\mathbf{x}^i)$ is computed by using equations (7.22) and (7.23) respectively.

Figure 7.2 shows an example of a survival function for the speed-ups y achieved by uniform search, and search guided by the Markov-oracle distribution. This has been obtained for the benchmark *adpcm* on the TI board. We can see that the Markov-oracle $V(y)$ outperforms the uniform $V(y)$ for all speed-ups greater than one. This means that if we use the Markov-oracle distribution to guide search we will have a greater chance of obtaining a good transformation sequence, and that the expected number of samples needed to achieve good performance will be reduced. This behaviour was consistent for all the benchmarks using both oracle distributions on both architectures.

The Search Improvement Factor

It is possible to compute the expected number of samples $E[n]$ needed to reach a good data-point (first success) when assuming random sampling. If we define a good data-point in the performance space as the one for which the speed-up is greater than certain value y^+, the expected number of samples needed to achieve this performance is $E[n] = 1/V(y^+)$. In order to evaluate the benefits of using a distribution for guiding search let us define the Search Improvement Factor as:

$$SIF = \frac{E_{\mathcal{U}}[n]}{E_{\mathcal{A}}[n]}, \tag{7.24}$$

where $E_{\mathcal{U}}[n]$ is the expected number of samples needed to achieve good performance when using the uniform distribution and $E_{\mathcal{A}}[n]$ is the expected number of samples needed to achieve good performance when using algorithm $\mathcal{A}$. In the PSD framework such algorithm is implemented by sampling the optimisation space using a specific search distribution (such as the IID-oracle). Thus, we will prefer SIF values greater than one as they indicate that our method speeds up (uniform) search.

Oracle Distributions

Table 7.3 shows the expected number of samples $E_{\mathcal{U}}[n]$ needed to achieve good performance for uniform search and the search improvement factors $SIFs$ for the oracle distributions, where a good solution has been defined to be a sequence that yields at least 95% of the maximum performance achieved for each benchmark.

The first column corresponds to the benchmarks used and the columns I and M correspond to iid and Markov distributions respectively. Note that those benchmarks for which no speed-up was obtained during the exhaustive experiments are marked with '-' indicating that their spaces

	TI			AMD		
		SIF			SIF	
PROGRAM	$E_{\mathcal{U}}[n]$	I	M	$E_{\mathcal{U}}[n]$	I	M
FFT	11.3	2.7	5.4	1.4	1.1	1.2
FIR	20.6	12.2	14.2	8.8	3.6	5.1
IIR	1.8	1.1	1.2	17.7	7.2	10.6
LATNRM	-	-	-	21.8	7.1	11.7
LMSFIR	-	-	-	15.2	6.9	9.5
MULT	-	-	-	82.6	17.5	39.8
ADPCM	66.3	3.6	13.8	-	-	-
COMPRESS	13.8	7.2	9.2	55.9	12.0	25.9
EDGE	96.4	18.1	45.4	29760	457	26720
HISTOGRAM	-	-	-	9.5	6.3	7.0
LPC	83.6	15.7	36.7	13.6	3.6	6.2
SPECTRAL	2.6	1.5	1.8	14.3	3.3	5.2
AVERAGE	37.0	5.0	8.8	2727	8.1	17.4

Table 7.3: Expected number of samples for uniform distribution and search improvement factors for **oracle** distributions to obtain 95% of maximum performance on the small space of the SUIF data set.

are not worth searching. It can be seen that both oracle distributions consistently improved search on both architectures. Furthermore, although in some easy-to-search spaces such as *iir* and *spectral* for the TI or *fft* for the AMD only marginal benefits are obtained, in difficult spaces such as *adpcm*, *edge* and *lpc* for the TI or *mult*, *compress* and *edge* for the AMD, the oracles speeded up search by an order of magnitude or more[3].

It is also possible to conclude that modelling interactions by using a Markov chain distribution does lead to greater improvements compared to an iid distribution. On average[4], the iid distribution and the Markov model improved search by factors of 5.0 and 8.8 for the TI and 8.1 and 17.4 for the AMD respectively.

[3] As pointed out in section 6.3.1, The benchmark *edge* is a needle-in-a-haystack problem (on the AMD) and the oracle distributions dramatically improved search.

[4] We have used the arithmetic mean for computing the average of $E_{\mathcal{U}}[n]$ and the geometric mean for averaging *SIF*s.

	TI			AMD		
		SIF			SIF	
PROGRAM	$E_{\mathcal{U}}[n]$	I	M	$E_{\mathcal{U}}[n]$	I	M
FFT	11.3	0.8	0.8	1.4	1.1	1.1
FIR	20.6	6.5	7.8	8.8	1.4	1.5
IIR	1.8	1.0	1.0	17.7	6.3	6.9
LATNRM	-	-	-	21.8	6.3	6.8
LMSFIR	-	-	-	15.2	3.3	4.4
MULT	-	-	-	82.6	12.7	24.2
ADPCM	66.3	1.4	1.4	-	-	-
COMPRESS	13.8	6.8	7.9	55.9	13.4	27.4
EDGE	96.4	2.2	2.3	29760	6.8	0.2
HISTOGRAM	-	-	-	9.5	6.3	7.0
LPC	83.6	2.2	2.3	13.6	3.3	4.7
SPECTRAL	2.6	0.9	0.9	14.3	3.3	4.8
AVERAGE	37.0	2.0	2.1	2727	4.5	4.1

Table 7.4: Expected number of samples for uniform distribution and search improvement factors for **predictive** distributions to obtain 95% of maximum performance on the small space of the SUIF data set.

Predictive Distributions

Having shown that using oracle distributions leads to an improvement in the search, the next step is to evaluate the predictive distributions when using 1-nearest neighbour. These results are shown in Table 7.4, where (as before) the benchmarks marked with '-' indicate that their spaces are uninteresting to search.

Not surprisingly, the search improvement factors of the predictive distributions are upper-bounded by the search improvement factors of the oracle distributions (with the exception of *compress* on the AMD). However, for most benchmarks the predictive distributions did improve search by focusing on regions of the space where good performance was obtained. Indeed, only for two benchmarks on the TI board (*fft* and *spectral*) and one benchmark on the AMD (*edge* when using the Markov distribution) did the predictive distributions harm performance. This latter case suggests overfitting, as the iid model speeded up search and has fewer parameters than the Markov model. It also confirms that transfer of knowledge to a needle-in-a-haystack problem can be very difficult, especially when having a small number of

training points. However, one can alleviate this effect by mixing the fitted distributions with the uniform distribution using weight factors of α and $(1-\alpha)$ respectively[5].

Best Search Improvement Factor: A Performance Upper Bound

It is interesting to analyse the performance of the predictive distributions compared to the best search improvement factor that can be achieved with the set of benchmarks available. Given a benchmark, this can be computed by finding the distribution on good solutions (on the rest of the benchmarks) that yields the maximum search improvement factor. Clearly, this is an upper bound on the SIF that can be achieved with a predictive distribution when using 1-nearest neighbour.

Table 7.5 shows the performance of the predictive distributions as a fraction of the best SIF. On the TI, We see that both predictive distributions achieve the best possible SIF on four benchmarks, namely *fir*, *adpcm*, *compress* and *edge*. On average, on the TI, the predictive distributions obtain 86% and 85% of the best possible SIF. On the AMD, we see that the iid predictive distribution achieves the best SIF or almost the best SIF on six benchmarks (*iir*, *lmsfir*, *mult*, *compress*, *edge* and *spectral*). The Markov distribution on the AMD, achieves the best SIF or almost the best SIF on *lmsfir*, *mult*, *compress* and *spectral*. However, the Markov distribution (on the AMD) obtains very suboptimal solutions for benchmarks *fir* and *edge*. On average, on the AMD, the predictive distributions obtain 86% and 62% of the best possible SIF.

7.5.7 Evaluation on the Large Space of the SUIF Data Set

In this section we aim to evaluate the performance of the PSD method on the large space of 90^{20} transformation sequences (see section 5.3.1 for details). In contrast with the small space, we do not have exhaustive data for the large space and thus we cannot compute the expected number of samples to achieve good performance $E_{\mathcal{U}}[n]$ in order to evaluate the benefits of our approach. Therefore, we have considered a different measure of comparison inspired by the concept of the area under the ROC (receiver operating characteristic) curve, which is commonly used in machine learning for the evaluation of classification methods.

The Area Under the Performance Curve (AUC)

A performance curve describes the best speed-up achieved so far by a search algorithm as a function of the number of iterations. Figure 7.3 shows an example of performance curves for the benchmark *mult* on the AMD architecture. Note that we have used the speed-up minus one

[5]Using $\alpha = 0.8$ the SIF for *edge* on the AMD with the Markov model increased to 0.7 while the average SIF for all the other benchmarks decreased from 5.7 to 4.9.

	TI		AMD	
PROGRAM	**I**	**M**	**I**	**M**
FFT	0.56	0.55	0.83	0.77
FIR	1.00	1.00	0.40	0.32
IIR	0.76	0.75	0.97	0.77
LATNRM	-	-	0.74	0.74
LMSFIR	-	-	0.96	0.98
MULT	-	-	1.00	1.00
ADPCM	1.00	1.00	-	-
COMPRESS	1.00	1.00	1.00	1.00
EDGE	1.00	1.00	1.00	0.05
HISTOGRAM	-	-	0.93	0.93
LPC	0.91	0.95	0.88	0.81
SPECTRAL	0.80	0.70	0.97	1.00
AVERAGE	**0.86**	**0.85**	**0.86**	**0.62**

Table 7.5: The performance of the predictive distributions as a fraction of the best search improvement factor (SIF/SIF^{best}) on the small space of the SUIF data set.

on the y-axis (so that the baseline performance corresponds to zero)[6] and a log-scale for the x-axis. It is clear that the area under the performance curve (AUC) will reward those methods that reach better performance and those that achieve good speed-ups in fewer iterations.

The AUCs have been computed for each benchmark when searching the optimisation space using uniform search, the iid predictive distribution and the Markov predictive distribution. Such AUC values have been averaged throughout 10 replications.

The AUCs for the TI and AMD are shown in Figure 7.4, where the computations have been done using performance curves for each benchmark like the one illustrated in Figure 7.3 after fifty iterations. Note that the results are shown only for those benchmarks for which an improvement was obtained with these experiments.

On the TI board, the iid distribution provides the best performance for most benchmarks, with a dramatic improvement achieved for *fft*. The Markov distribution improves performance over uniform in some cases but decreases it in others. On the AMD, both predictive distributions improve performance on most benchmarks; of the 10, the best AUC performance is given by iid on 5 (*fft*, *compress*, *mult*, *edge* and *fir*), and by Markov on the other 5 (*latnrm*, *lmsfir*, *iir*,

[6]By running the baseline program first, the minimum value of the best speed-up (so far) minus one is at least zero.

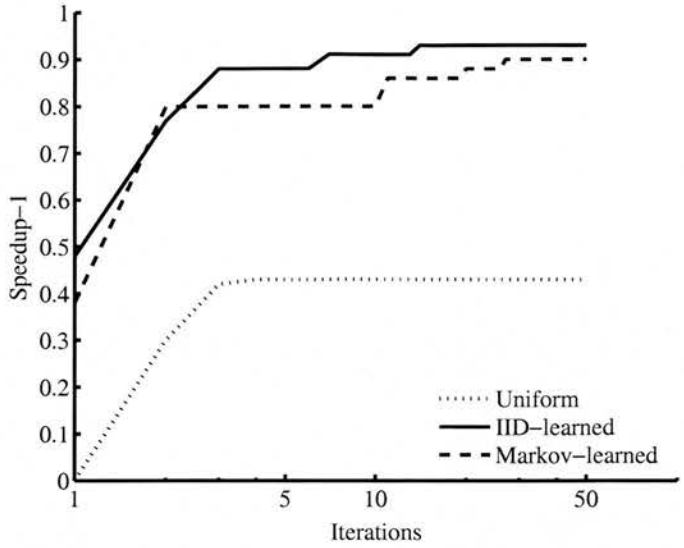

Figure 7.3: Performance curves for the benchmark *mult* on the AMD architecture on the large space of the SUIF data set of 90^{20} transformation sequences when using uniform search, search guided by the iid distribution and search guided by the Markov distribution.

spectral and *histogram*).

With these empirical results we conclude that having a predictive distribution generally improved search on the large space. A detailed analysis of the results shows that the improvements achieved by the predictive distribution after five iterations are at least as good as the ones obtained by uniform search after fifty iterations, which translates into an speed-up of iterative optimisation of an order of magnitude. For the small space the Markov distribution provided the best performance for most benchmarks. On the large space the choice between iid and Markov was less clear cut, but both predictive distributions give improvements over uniform search.

7.5.8 Comparison to Other Baselines

It has been shown so far that Predictive Search Distributions provide significant improvements in the compiler optimisation problems studied over iterative compilation (i.e. over search when using the uniform distribution). Nonetheless, comparisons to other baselines are necessary in order to demonstrate that this approach represents a significant advantage with respect to several simple heuristics that can be used for optimisation. In this section we provide the results of some possible heuristics on the compiler optimisation problems studied and show that such heuristics are outperformed by the PSD technique. Here we will focus on the analysis of the *small space of the SUIF data set* given the existence of exhaustive data for this space.

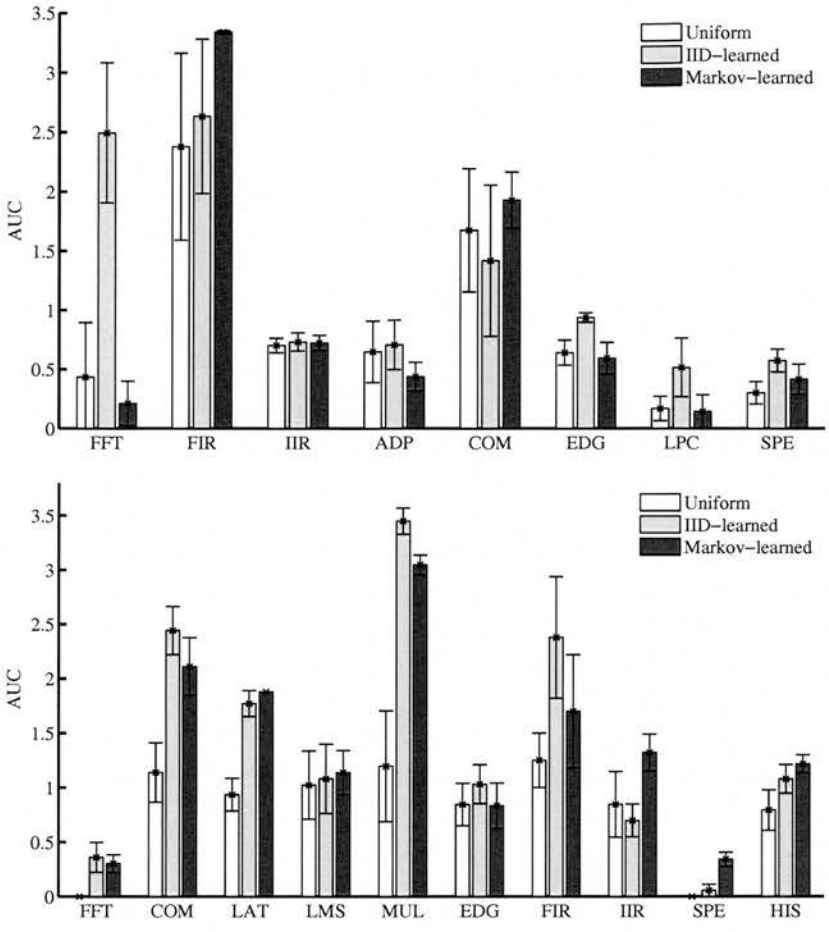

Figure 7.4: Areas under the performance curve (AUC) for TI (top) and AMD (bottom) on the large space of the SUIF data set of 90^{20} sequences. The error bars denote one standard deviation of the mean. The $\times$ symbol for *fft* and *spectral* on AMD means that uniform search did not provide any improvement in performance for the number of iterations considered. Results shown only for those benchmarks on which some improvement can be achieved.

Average Distribution

Predictive search distributions allows inductive generalisation to new optimisation problems by learning distributions on good solutions on previously (related) solved problems. Such generalisation relies on the assumption that there is a set of common features across these problems and that these features are informative about regions of the space where good solutions can be found. One possible way of empirically assessing the informativeness of the features on the compiler optimisation problem is to predict (on a new benchmark) the average distribution of the training data, which ignores the knowledge given by the code features. This distribution is obtained by uniformly mixing the distributions on good solutions of the training benchmarks. If the average distribution yields improvements that are similar to or greater than the ones ob-

	TI			AMD		
		SIF			SIF	
PROGRAM	$E_{\mathcal{U}}[n]$	I	M	$E_{\mathcal{U}}[n]$	I	M
FFT	11.3	0.8	0.9	1.4	1.1	1.2
FIR	20.6	1.2	1.3	8.8	0.9	1.1
IIR	1.8	1.0	1.0	17.7	3.1	3.9
LATNRM	-	-	-	21.8	3.5	4.1
LMSFIR	-	-	-	15.2	1.5	1.5
MULT	-	-	-	82.6	3.8	3.8
ADPCM	66.3	0.7	0.7	-	-	-
COMPRESS	13.8	1.2	1.3	55.9	3.7	3.4
EDGE	96.4	1.1	1.0	29760.0	2.4	2.4
HISTOGRAM	-	-	-	9.5	3.3	3.7
LPC	83.6	1.1	1.0	13.6	2.8	3.2
SPECTRAL	2.6	0.8	0.9	14.3	2.7	3.2
AVERAGE	**37.0**	**1.0**	**1.0**	**2727.3**	**2.4**	**2.6**

Table 7.6: Search improvement factors achieved on the small space of the SUIF data set when using the average training distributions.

tained with PSD, one can conclude that perhaps there is a set of common transformations that are beneficial for all or most benchmarks, which would overshadow the predictive power of the code features used.

Table 7.6 shows the search improvement factors when making predictions with the average training distribution. We see that using this distribution does not bring any improvement on average over uniform search on the TI architecture. Additionally, although the average distribution outperforms uniform search on the AMD architecture, we see that such speed-ups are significantly lower than the ones achieved with PSD (see Table 7.4). Indeed, the PSD technique outperforms the average distribution on most benchmarks on both architectures.

Best Training Distribution

In addition to comparing against a simple average distribution it is also possible to ascertain the best distribution on the training data. This can be found by selecting the distribution on good solutions of the training benchmark that achieves the maximum geometric mean of the search improvement factors (SIFs) on the rest of training benchmarks. In other words, if there are M training benchmarks, the best training distribution is the one that achieves the best average SIF

	TI			AMD		
		SIF			SIF	
PROGRAM	$E_{\mathcal{U}}[n]$	I	M	$E_{\mathcal{U}}[n]$	I	M
FFT	11.3	0.8	0.7	1.4	1.2	1.2
FIR	20.6	1.8	1.9	8.8	1.3	2.3
IIR	1.8	0.9	1.1	17.7	1.5	4.5
LATNRM	-	-	-	21.8	2.8	4.3
LMSFIR	-	-	-	15.2	2.8	4.4
MULT	-	-	-	82.6	6.7	2.1
ADPCM	66.3	0.7	1.0	-	-	-
COMPRESS	13.8	1.6	2.0	55.9	12.9	1.8
EDGE	96.4	0.4	0.0	29760.0	6.8	3.2
HISTOGRAM	-	-	-	9.5	3.7	3.5
LPC	83.6	1.3	0.0	13.6	2.4	4.7
SPECTRAL	2.6	0.9	0.0	14.3	3.4	4.8
AVERAGE	**37.0**	**1.0**	**0.0**	**2727.3**	**3.2**	**3.1**

Table 7.7: Search improvement factors achieved on the small space of the SUIF data set when using the best training distributions.

on the $M-1$ training programs. As before, if similar (or better) improvements are obtained with this approach, the informativeness of the features used for predictions would be questioned.

Table 7.7 shows the SIFs obtained when using the best training distribution. We see that the best training distribution not only does not outperform uniform search on the TI architecture when using the iid distribution but also considerably harms performance when using the Markov distribution. Note that SIFs with value zero are only a numerical approximation and they indicate that such factors are very small. On the AMD, as in the case of the average distribution, the best training distribution improves over uniform search but such improvements are lower than the ones achieved with PSD.

Best Sequences of Nearest Neighbour

In this method, given a new program one predicts the best performing sequences of its nearest neighbour. Unlike the two previous baseline heuristics where the features were not used, here the features are used to compute a program's nearest neighbour. However, rather than using the predictive distribution one simply predicts the best transformation sequences as given by the full table of speed-ups of the program's nearest neighbour.

	$\mathbf{SIF}_{Table}$	
PROGRAM	**TI**	**AMD**
FFT	0.03	1.39
FIR	0.01	0.00
IIR	1.82	0.05
LATNRM	-	10.89
LMSFIR	-	15.24
MULT	-	0.20
ADPCM	0.24	-
COMPRESS	13.8	3.73
EDGE	0.00	1.64
HISTOGRAM	-	9.54
LPC	0.00	13.56
SPECTRAL	2.59	14.26
AVERAGE	**0.13**	**1.58**

Table 7.8: Search improvement factors achieved on the small space of the SUIF data set when using the full table of speed-ups of each program's nearest neighbour.

Clearly, this is a particular instance of the PSD technique using a sum of delta functions as predictive distributions. However, it is interesting to analyse if having proper distributions over good solutions instead of recording the best transformation sequences does yield better improvements.

In order to compute the search improvement factors obtained with this heuristic, it is necessary to calculate the expected number of samples to achieve good performance ($E[n]$) when using the **full table** of speed-ups of the program's nearest neighbour. This can be computed as follows. The complete set of speed-ups of the program's nearest neighbour are sorted in descending order. Each of the corresponding sequences is then evaluated on the new program according to such order until 95% of the maximum speed-up is achieved.

Table 7.8 shows the search improvement factors obtained with this method. By comparing these results with the ones on Table 7.4 we see that the PSD method outperforms this baseline heuristic on average on both architectures. Indeed, although this simple method outperforms PSD on some benchmarks such as *compress* on the TI or *spectral* on the AMD, in general, greater benefits are obtained by building a smooth distribution over good solutions instead of simply using the stored solutions to previously solved problems. Furthermore, it can be seen on Table 7.8 that the "full-table" method significantly harms performance over uniform search

on the TI architecture. Additionally, this simple technique requires the storage of a large (if not all) number of solutions on previous problems, which is not the case for the PSD framework (see section 7.5.10 for more details).

7.5.9 Analysis of Learned Distributions

Given the effectiveness of the PSD technique, it is interesting to understand the distributions that have been fitted to each benchmark (i.e. the oracle distributions) and also the distributions that have been learnt and used for searching the optimisation space of a "new" benchmark (i.e. the predictive distribution).

Figure 7.5 shows the iid distributions on the TI (top) and on the AMD (bottom) architectures for the small space of the SUIF data set. By looking at the LHS of Figure 7.5 we notice that (as described in chapter 6), several transformations have a significant impact on the performance of each benchmark and they vary across the different programs. In particular, we can see that *unrolling*, *flattening*, *normalisation*, *common subexpression elimination*, *if hoisting* and *move loop-invariant conditionals* are very important transformations when modelling good transformations sequences for this set of benchmarks on the TI architecture. Similarly, we can highlight *unrolling*, *flattening*, *cse*, *hoisting of loop invariants* and *move loop-invariant conditionals* on the AMD architecture.

When comparing the LHS with the RHS of Figure 7.5, one particular case worth mentioning is the one obtained by analysing the rows corresponding to the benchmarks *fir* and *compress* on the TI, which seem to be very close in the parameter space of the iid distribution. When predicting on each of these benchmarks, they have been found to be each other's nearest neighbour. Moreover, this correspondence in similarity in code-feature space to similarity in parameter-space of the iid search distribution led to the maximum improvements obtained on the TI architecture when using the iid search distribution as seen in Table 7.4. However, this is less clear cut on the AMD architecture as there is less variability on the parameters of the iid distribution across benchmarks.

7.5.10 Effect of the Number of Training Samples

The results for the small space of the SUIF data set presented so far have been obtained when the training (model) distributions $g(\mathbf{x})$ have been learnt on each benchmark using the complete data. However, even for the small space, it would be impractical for the PSD technique to rely upon exhaustive data. Therefore, it is necessary to investigate the effect of the number of training samples on the effectiveness of the PSD technique.

The search improvement factors achieved with the predictive distributions on each benchmark as a function of different values of training samples N are shown on Figures 7.6 and 7.7 for the TI and on Figures 7.8 and 7.9 for the AMD. Note the differences in scales of the SIFs

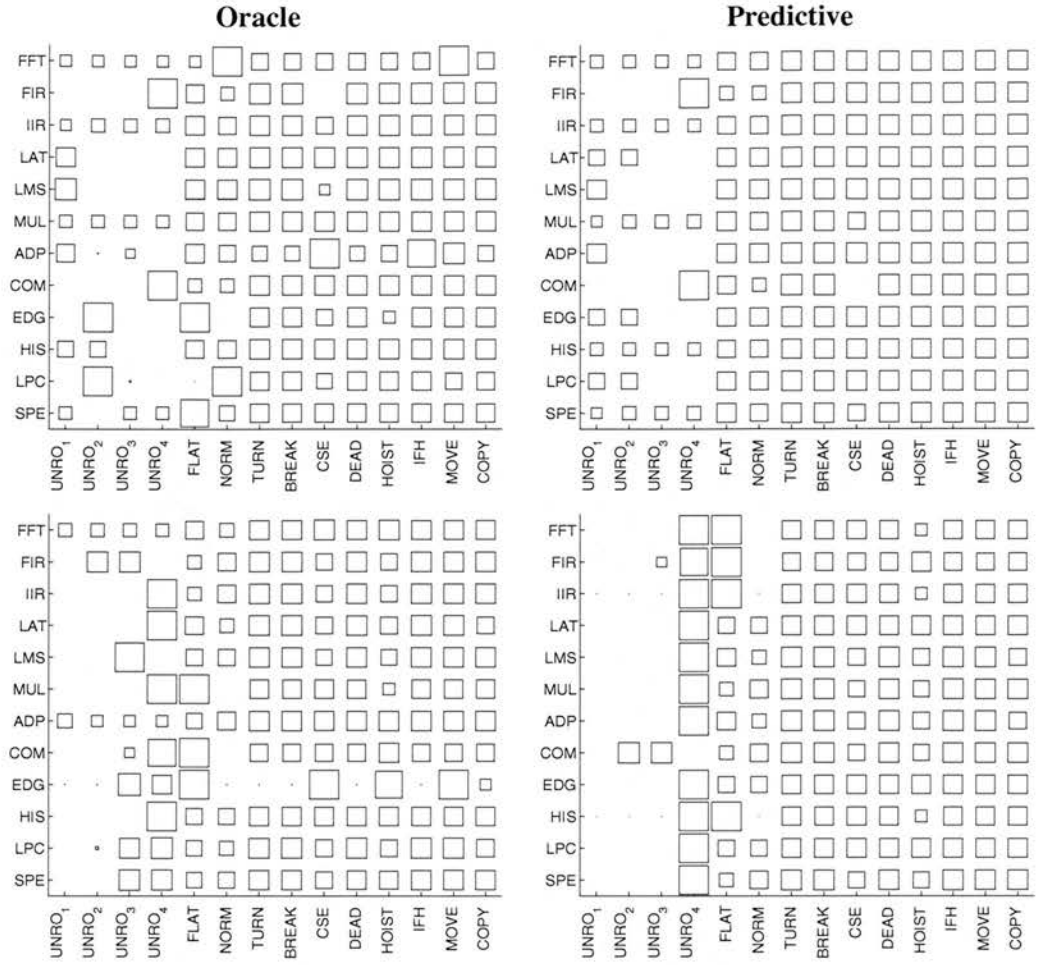

Figure 7.5: The oracle iid distributions and the predictive iid distributions on the TI (top) and on the AMD (bottom) on the small space of the SUIF data set. For each benchmark on the vertical axis and each transformation on the horizontal axis, the probability of applying a transformation $p(x_i) = a_j$ is proportional to the area of the corresponding region in a Hinton diagram.

across benchmarks. We have preferred to report the absolute values of the SIFs instead of a normalised measure to highlight the fact that the technique performs significantly better on some benchmarks than on others.

Although one may expect a non-decreasing behaviour of these curves as a function of N, as in the ones presented for *spectral* or *lpc* on the AMD (Figure 7.9), in general this is not the case. This may be simply due to sampling artefacts, especially when the number of samples is very small, i.e. $N < 500$.

The general trend is that, on average, there is not much sensitivity of the technique to the number of training samples used. However, when the number of training examples is very low, e.g. when $N = 100$ the SIFs are quite variable. For benchmarks such as *fir* and *compress* on the TI or *iir*, *mult* and *histogram* on the AMD, using $N = 500$ samples leads to similar SIFs to the

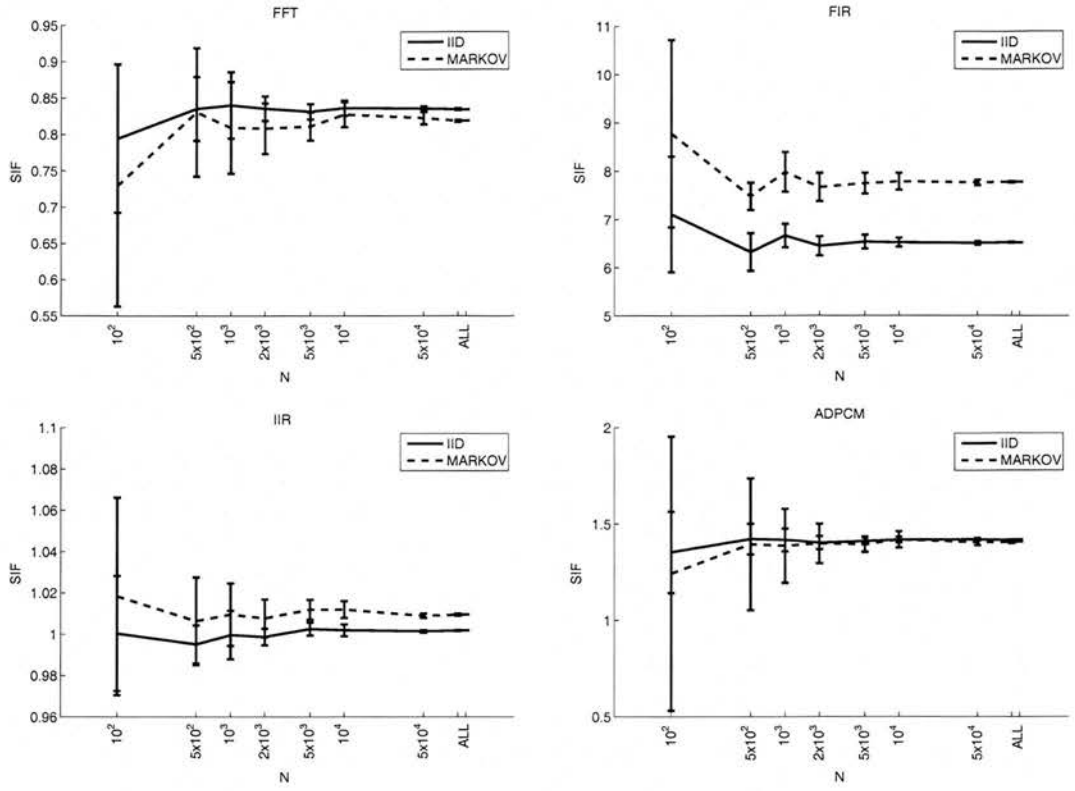

Figure 7.6: The search improvement factors of the predictive distributions on the small space of the SUIF data set for the TI board as a function of the number of samples per training benchmark. Results shown as averages over 10 replications with one standard deviation error bars on four benchmarks for which an improvement in performance can be achieved.

ones obtained when using the exhaustive data. For more difficult-to-search programs such as *adpcm* and *lpc* on the TI, one may require a few thousand examples in order to achieve good performance.

We can also highlight the fact that the results obtained with the Markov distribution are not significantly different from the iid distribution's on the TI board. Although on *compress* and *fir* the Markov distribution leads to greater SIFs, on the rest of benchmarks they provide roughly the same performance. This is not the case on the AMD, where the Markov distribution consistently outperforms the iid distribution on all but the benchmark *edge*, where the results are quite poor. These results are similar to the ones obtained when using the complete data for training.

7.6 Summary and Discussion

In this chapter we have presented predictive search distributions (PSD) as a general method for speeding up search on combinatorial optimisation problems. The general idea is to learn

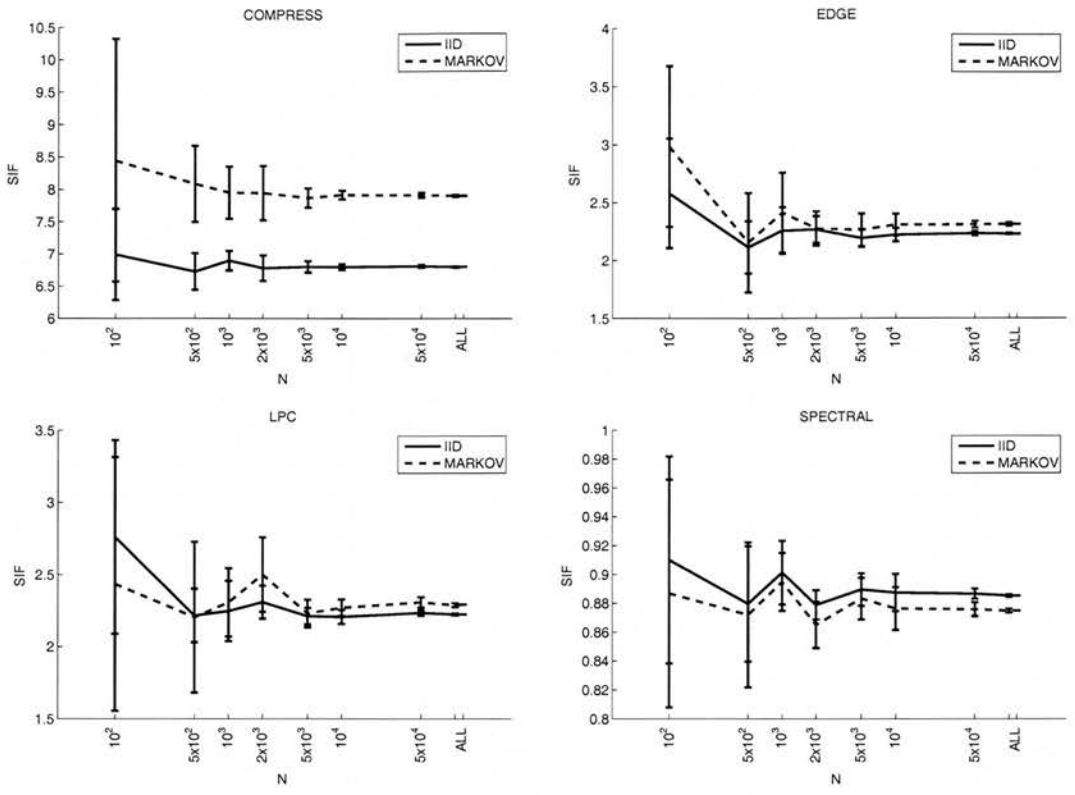

Figure 7.7: The search improvement factors of the predictive distributions on the small space of the SUIF data set for the TI board as a function of the number of samples per training benchmark. Results shown as averages over 10 replications with one standard deviation error bars on the remaining four benchmarks for which an improvement in performance can be achieved.

a distribution $q(\mathbf{x}|\mathbf{t})$ over good solutions $\mathbf{x}$ on a collection of optimisation problems, each of which can be characterised by a set of features $\mathbf{t}$. When a new problem is presented, the predictive distribution $q(\mathbf{x}|\mathbf{t})$ is used to focus search.

The PSD technique is inspired by recent work on Estimation of Distribution Algorithms (EDAs), where a probability distribution on good solutions is iteratively learned (or evolved) in order to solve a **single** optimisation problem. PSD aims to achieve **transference** across different optimisation problems by learning a distribution over EDA-search distributions on these problems, given a common representation (or feature vector) $\mathbf{t}$.

PSD can be learnt by maximisation of the conditional likelihood or by using memory-based methods. The latter are specially suitable when the number of problems (or tasks) is limited.

It is straightforward to make predictions with PSD. Given a new problem described by a set of features $\mathbf{t}$, one simply uses the distribution $q(\mathbf{x}|\mathbf{t})$ to guide search on the optimisation space of this new problem.

Given a set of optimisation problems standard model selection criteria such as the AIC, the BIC or Bayes factors can be used in order to select a suitable family of distributions Q, to

which the distribution q will be constrained.

The PSD technique has been applied to the compiler optimisation problem on the SUIF data set and has been shown to yield significant improvements on both the small space and the large space on the TI board and the AMD architecture.

One of the crucial requirements of the PSD technique is the existence of a common representation given by a set of features across different optimisation problems. Although in the compiler optimisation task these features appear naturally (e.g. code features of a program), in other problems such representation may become a hurdle. Furthermore, even if it is possible to extract a meaningful representation for a set of optimisation problems, learning a distribution over good solutions $g(\mathbf{x}|T)$ may require a large amount of training data when such distribution varies very rapidly with changes in the input T.

Other examples of domains where there are families of optimisation problems include finding the ground state of a spin glass (Pelikan and Goldberg, 2001), or the minimum balanced cut graph partitioning problem (Andreev and Räcke, 2004). Here families are induced by varying the edge weights in the input graph.

We have described in this chapter the direct approach to the problem of finding good transformation sequences for programs, where these sequences are modelled directly using search distributions. As explained in section 3.3, we can also adopt an indirect approach by building performance models of a program when being applied a sequence of transformations. This is addressed in the next chapter with multi-task Gaussian process regression models, which are used to exploit transference across programs by modelling the correlations between their performances directly. These models are then utilised as fast predictors of the performance of a new program (or a program for which very little data is available) in order to search for good compiler transformation sequences.

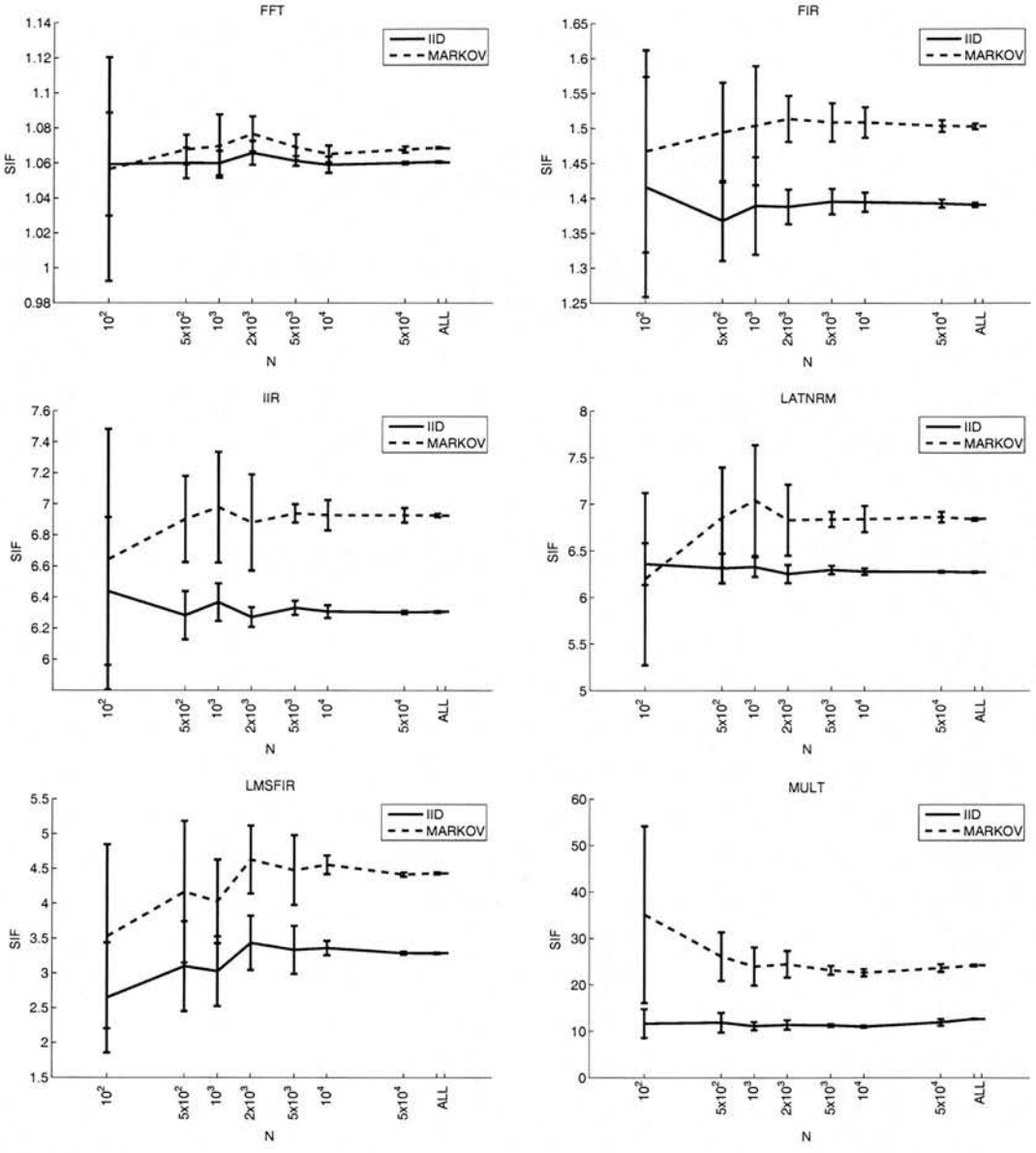

Figure 7.8: The search improvement factors of the predictive distributions on the small space of the SUIF data set for the AMD architecture as a function of the number of samples per training benchmark. Results shown as averages over 10 replications with one standard deviation error bars on six benchmarks for which an improvement in performance can be achieved.

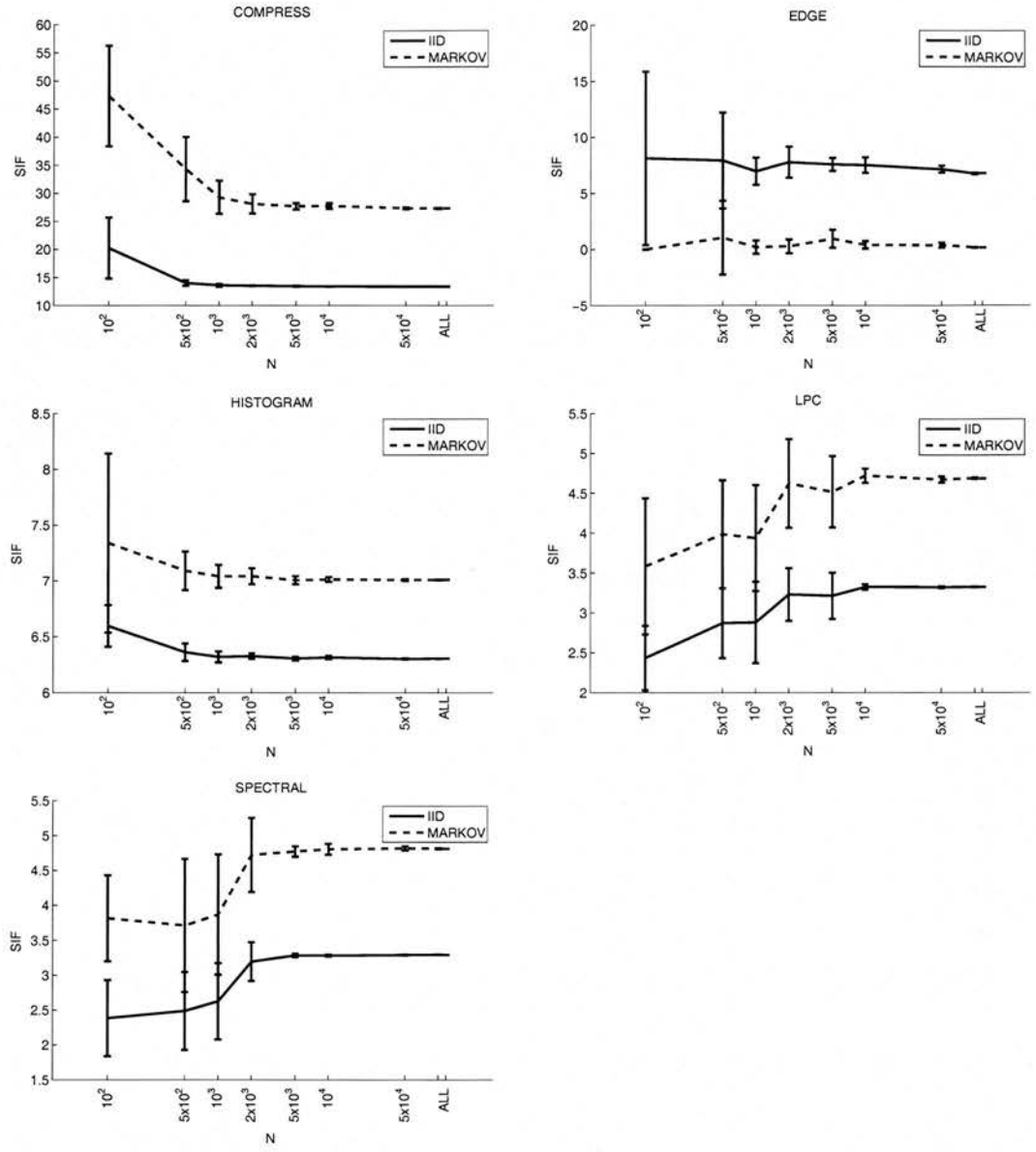

Figure 7.9: The search improvement factors of the predictive distributions on the small space of the SUIF data set for the AMD architecture as a function of the number of samples per training benchmark. Results shown as averages over 10 replications with one standard deviation error bars on the remaining five benchmarks for which an improvement in performance can be achieved.

Chapter 8

Multi-task Gaussian Process Prediction

Chapter 7 has described Predictive Search Distributions as a technique to tackle combinatorial optimisation problems. In this method a probability distribution over good solutions is learnt on a set of optimisation problems and used to guide search on a problem that has not been seen before.

In this chapter, we follow a rather different approach. Here, we build predictors of the evaluation function on the set of feasible solutions over different optimisation problems. These performance predictors (or proxies) are then used to approximate the evaluation function on a new problem. Clearly, if the search space is very large evaluating all feasible solutions may be prohibitive. In such cases, the learned performance predictor can be used within search (or sampling) algorithms avoiding the execution of the actual evaluation function. As in the case of the compiler optimisation problem, this evaluation function can be very costly and clear benefits can be obtained by following this approach.

As in Chapter 7, the focus here is on learning predictors over different tasks (problems) and exploiting the shared information across these tasks. Hence, this chapter analyses the construction of regression models in a multi-task learning scenario. In particular, it investigates multi-task regression in the context of Gaussian processes. As stated above, these regression models can be used in order to tackle optimisation problems where the evaluation function is expensive to execute. However, the models presented here can also be used in (general) regression problems where transference can be achieved.

The organisation of this chapter is as follows. Gaussian process (GP) regression is briefly described in section 8.1. Sections 8.2 and 8.3 explain single-task learning and multi-task learning. Sections 8.4 and 8.5 describe multi-task Gaussian process prediction with task-descriptor features and without task-descriptor features. Section 8.6 proposes a measure for quantifying inter-task transfer in multi-task GP regression. Section 8.8 explains how to apply multi-task

GP to the compiler optimisation problem and section 8.9 presents a summary and a discussion.

8.1 Regression with Gaussian Processes (GPs)

The standard supervised learning problem is the following. Given a data set of N (input-output) observations $\mathcal{D} = \{(\mathbf{x}_i, y_i) | i = 1, \ldots, N; \mathbf{x} \in \mathbb{R}^d\}$, and denoting X by the set of all input observations and $\mathbf{y} = (y_1, \ldots, y_N)^{\mathrm{T}}$, we want to make predictions at a new point $\mathbf{x}_*$ that is not included in the training set $\mathcal{D}$. In other words, our aim is to learn the mapping $\mathbf{x} \rightarrow y$ (based on the set $\mathcal{D}$) so that it provides us with inductive power, i.e. it allows us to generalise to new points that have not been seen before. In the case of regression, the target values y are (in general) real values.

One possible solution to this problem is to consider a parametric approach, for example $y = f(\mathbf{w}, \mathbf{x}) + \eta$, where $f : \mathbf{x} \rightarrow \mathbb{R}$ is a regression function that approximates the target values y corresponding to inputs $\mathbf{x}$, and η is additive noise. A possible choice for $f(\mathbf{w}, \mathbf{x})$ is a linear model $f(\mathbf{w}, \mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$. This is a simple and well known model commonly used in statistics due to its easy interpretability (see e.g. Neter et al., 1996, Chapter 6).

However, such model lacks enough flexibility to fit complex data sets and predictions at new points $\mathbf{x}_*$ may be rather poor. One can augment the flexibility of the model by considering more complex parameterisations of the regression function $f(\mathbf{w}, \mathbf{x})$, for example by including nonlinearities (as in e.g. neural networks). However, it is difficult to ascertain beforehand the level of flexibility of such functions as one can easily *overfit* the data. In order to avoid this, it is customary to make use of *regularisation* procedures that penalise highly complex models.

A rather more elegant approach is to consider a *nonparametric Bayesian* view of the regression problem on the function space $\mathcal{F}$. Here, we place a prior (probability distribution) over functions reflecting our initial beliefs of what functions are more likely than others. We can then update our beliefs on these functions after seeing some data $\mathcal{D}$.

There are several advantages of following this approach. Firstly, we do not have to worry about how well we can fit our data since our function is not constrained to have a parametric form. In principle, by being *nonparametric* we have enough flexibility to model complex data.

Secondly, by being *Bayesian* we consider our prior beliefs on the functions f and use this information along with the observed data in order to obtain a distribution $p(\mathbf{f}_* | \mathbf{x}_*, \mathcal{D})$ that we use for making predictions on a new data point $\mathbf{x}_*$. Clearly, this is much more informative than a single point prediction. In contrast to the frequentist approach where a hypothesis is assumed to be fixed and the probability distribution of the (observed or unobserved) data is considered, in the Bayesian approach a probability distribution over the hypotheses is considered and inference is done by conditioning on the observed data (evidence). In other words, in Bayesian inference one has a probability distribution over unknown parameters and uses the rules of

probability in order to make predictions. This contrasts with the frequentist view where the unknown parameters are assumed to be fixed. The reader is referred to O'Hagan and Forster (1994); Jeffreys (1966) for a comprehensive description of Bayesian statistics.

A final advantage of having a nonparametric Bayesian approach to the regression problem and more specifically Gaussian processes is that most commonly used regression methods such as linear regression, splines or neural networks (when the number of hidden units tends to infinity) can be seen as instances of this approach.

One possible reason for concern is that naïvely one would have to consider the function values at infinitely many points, which obviously is intractable. Fortunately, as we shall see later, in the Gaussian process framework one only cares about those points at which observations are made.

In the following paragraphs a brief overview of Gaussian processes is provided. This follows the description given by Rasmussen and Williams (2006, Chapter 1).

8.1.1 Gaussian Process

A Gaussian process (GP) is a *stochastic process* where every subset of random variables have a joint Gaussian distribution. It is therefore a generalisation of the Gaussian (probability) distribution. The probability distribution of a function $f(\mathbf{x})$ is a Gaussian process if for any finite subset of points $\mathbf{x}_1, \ldots, \mathbf{x}_N$, the function values $f(\mathbf{x}_1), \ldots, f(\mathbf{x}_N)$ follow a Gaussian distribution. We can denote a Gaussian process with:

$$f(\mathbf{x}) \sim \mathcal{GP}(\mu(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')), \tag{8.1}$$

$$\mu(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})], \tag{8.2}$$

$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - \mu(\mathbf{x}))(f(\mathbf{x}') - \mu(\mathbf{x}'))], \tag{8.3}$$

where $\mu(\mathbf{x})$ and $k(\mathbf{x}, \mathbf{x}')$ are the mean function an the covariance function of the GP respectively. As in the case of a Gaussian distribution, a Gaussian process is completely determined by its mean function and its covariance function. Here we consider GPs with zero mean function.

8.1.2 The Covariance Function

It is clear from equation 8.1 that a critical component in GPs is the definition of a covariance (or kernel) function $k(\mathbf{x}, \mathbf{x}')$. Intuitively, we can think of the covariance function as a definition of similarity in input space. Ideally, under the supervised learning setting, points that are nearby in input space should have similar values in output space. Therefore, predictions at a new data point $\mathbf{x}_*$ should be strongly influenced by the target values of those data points in $\mathcal{D}$ that are closer to $\mathbf{x}_*$.

The matrix K resulting from evaluating the covariance function at all pairwise input points in $\mathcal{D}$ such that $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ is known as the *covariance matrix*. In the sequel we will sometimes also refer to this matrix as the *Gram matrix*.

The only constraint on the covariance function is that it must generate a positive semi-definite (PSD) matrix at any subset of points in X. In other words, a (real PSD) covariance matrix must satisfy $\mathbf{b}^T K \mathbf{b} \geq 0$ for all $\mathbf{b} \in \mathbb{R}^N$.

A covariance function is said to be *stationary* if it is translation-invariant in input space, i.e. it is a function of $\mathbf{x} - \mathbf{x}'$. If further, the covariance function is a function of the magnitude $\|\mathbf{x} - \mathbf{x}'\|$ then it is said to be *isotropic*.

Further literature on covariance (or kernel) functions can be found in Rasmussen and Williams (2006, chapter 4), MacKay (1998) and Schölkopf and Smola (2001, chapters 4 and 13).

The Squared Exponential Covariance Function

One possible choice of covariance function is the squared exponential (SE) (or Gaussian) kernel:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_s^2 \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^T C (\mathbf{x} - \mathbf{x}')\right), \tag{8.4}$$

where σ_s is the signal variance and C is a symmetric matrix that can have different parameterisations. For example, having $C = \ell^{-2} I$ we obtain the *isotropic* squared exponential kernel. Alternatively, having $C = \text{diag}(\boldsymbol{\ell})^{-2}$ with $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_d)$ we effectively implement automatic relevance determination (ARD) as in Neal (1996). Each ℓ_j is known as the characteristic length-scale of the corresponding dimension and it characterises the distance along that particular direction for which the function values are expected to vary significantly.

The parameters of the covariance function are known in the GP literature as the *hyperparameters* of the GP. In the case of the SE covariance function the hyperparameters are σ_s and the parameters of C.

8.1.3 Gaussian Process Prediction

Recall that our supervised learning problem described at the beginning of this section is to learn the mapping $\mathbf{x} \to y$ based on the observations $\mathcal{D}$. We can approach this problem by having the following model:

$$y = f(\mathbf{x}) + \eta, \tag{8.5}$$

$$f(\mathbf{x}) \sim \mathcal{GP}(\mathbf{0}, k(\mathbf{x}, \mathbf{x}')), \qquad \eta \sim \mathcal{N}(0, \sigma_n^2), \tag{8.6}$$

where $k(\mathbf{x}, \mathbf{x}')$ is given for example by equation 8.4, and we assume noisy observations with noise variance σ_n^2. Clearly, the function values at the training data points distribute according

to $\mathbf{f} \sim \mathcal{N}(\mathbf{0}, K(X,X))$ and the target values $\mathbf{y} \sim \mathcal{N}(\mathbf{0}, K(X,X)+\sigma_n^2 I)$, with X being the set of all input observations $\mathbf{x}_1, \ldots, \mathbf{x}_N$. For simplicity, let us denote the covariance matrix at the training data points by K and the vector of covariances between a test point $\mathbf{x}_*$ and the training points X by $\mathbf{k}_*$. Hence, by computing the conditional distribution of two jointly Gaussian random vectors (see e.g. Rasmussen and Williams, 2006, appendix A.2) we can obtain the posterior distribution:

$$f_*|\mathbf{x}_*, X, \mathbf{y} \sim \mathcal{N}(\mathbb{E}[f_*], \mathbb{V}[f_*]), \tag{8.7}$$

$$\mathbb{E}[f_*] = \mathbf{k}_*^T (K+\sigma_n^2 I)^{-1}\mathbf{y}, \tag{8.8}$$

$$\mathbb{V}[f_*] = k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{k}_*^T (K+\sigma_n^2 I)^{-1}\mathbf{k}_*. \tag{8.9}$$

Note that if the mean prediction is used at a new data point (which in this case minimises the mean square error and the mean absolute error), such prediction is a linear combination of the training targets $\mathbf{y}$. Therefore, Gaussian process prediction is a *linear prediction* model. However, one of the most interesting features of GPs is that most common parametric models can be seen as an instance of a GP model. In particular, one can obtain the GP prediction equations by placing a Gaussian prior over the weights of a linear regression model, see for example Rasmussen and Williams (2006, section 2.1) or MacKay (1998). Additionally, it has been shown by Neal (1996) and Williams (1997) that under certain assumptions a neural network with an infinite number of hidden units tends to a Gaussian process. Further relationships of GPs to other methods such as splines or Support Vector Regression have also been established in the literature, see e.g. Rasmussen and Williams (2006, chapter 6) or MacKay (2003, chapter 45).

8.1.4 Learning Hyperparameters

We have described above how to make predictions with Gaussian processes when the hyperparameters of the covariance function are fixed. However, in general, given a data set we do not know a good setting for these hyperparameters beforehand. Therefore, we can attempt to learn them from our set of observations $\mathcal{D}$. Let us include the noise variance from the model in equations (8.5) and (8.6) into the set of hyperparameters of our GP. We can do this by noting that $\mathrm{cov}(y_p, y_q) = k(\mathbf{x}_p, \mathbf{x}_q) + \sigma_n^2 \delta_{pq}$, with $\delta_{pq} = 1$ if $p = q$ and $\delta_{pq} = 0$ otherwise. Let us also denote this set of hyperparameters by $\boldsymbol{\theta}$. One possible approach to learning $\boldsymbol{\theta}$ from $\mathcal{D}$ is to find those $\boldsymbol{\theta}$ that maximise the marginal likelihood (or evidence):

$$p(\mathbf{y}|X) = \int p(\mathbf{y}|\mathbf{f}, X) p(\mathbf{f}|X)\, \mathbf{df}, \tag{8.10}$$

with $\mathbf{y}|\mathbf{f},X \sim \mathcal{N}_{\mathbf{y}}(\mathbf{f},\sigma_n^2 I)$ and $\mathbf{f}|X \sim \mathcal{N}_{\mathbf{f}}(\mathbf{0},K)$. We can solve this integral analytically as follows:

$$\begin{aligned} p(\mathbf{y}|X) &= \int \mathcal{N}_{\mathbf{y}}(\mathbf{f},\sigma_n^2 I)\mathcal{N}_{\mathbf{f}}(\mathbf{0},K)\,\mathbf{df} && (8.11)\\ &= \int \mathcal{N}_{\mathbf{f}}(\mathbf{y},\sigma_n^2 I)\mathcal{N}_{\mathbf{f}}(\mathbf{0},K)\,\mathbf{df} && (8.12)\\ &= Z^{-1}\int \mathcal{N}_{\mathbf{f}}(\mu_f,C_f)\,\mathbf{df} && (8.13)\\ &= \mathcal{N}_{\mathbf{y}}(\mathbf{0},K+\sigma_n^2 I), && (8.14) \end{aligned}$$

where we have applied the property of the product of two Gaussians: $\mathcal{N}_{\mathbf{x}}(\mathbf{a},A)\mathcal{N}_{\mathbf{x}}(\mathbf{b},B) = Z^{-1}\mathcal{N}_{\mathbf{x}}(\mu_x,C_x)$ with $Z^{-1} = \mathcal{N}_{\mathbf{a}}(\mathbf{b},A+B)$. In order to avoid stability problems and considering that the logarithm function is monotonically increasing, it is customary to maximise the (marginal) log-likelihood instead:

$$\mathcal{L} = \log p(\mathbf{y}|X) = -\frac{1}{2}\mathbf{y}^T(K+\sigma_n^2 I)^{-1}\mathbf{y} - \frac{1}{2}\log|K+\sigma_n^2 I| - \frac{n}{2}\log 2\pi. \quad (8.15)$$

The partial derivatives of the marginal likelihood with respect to a parameter θ_j of the covariance function can be obtained as follows:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta_j} &= -\frac{1}{2}\operatorname{tr}\left(K_\sigma^{-1}\frac{\partial K_\sigma}{\partial \theta_j}\right) + \frac{1}{2}\mathbf{y}^T K_\sigma^{-1}\frac{\partial K_\sigma}{\partial \theta_j}K_\sigma^{-1}\mathbf{y} && (8.16)\\ &= \frac{1}{2}\operatorname{tr}\left((\boldsymbol{\alpha}\boldsymbol{\alpha}^T - K_\sigma^{-1})\frac{\partial K_\sigma}{\partial \theta_j}\right), && (8.17) \end{aligned}$$

with $K_\sigma = K+\sigma_n^2 I$ and $\boldsymbol{\alpha} = K_\sigma^{-1}\mathbf{y}$.

We see that the gradients need the computation of $\partial K_\sigma/\partial\theta_j$, which depends on the form of the covariance function used. In the case of the SE covariance function with ARD let us denote $\boldsymbol{\theta} = (\log\sigma_s, \log\ell_1, \ldots, \log\ell_d, \log\sigma_n)^{\mathrm{T}}$, where we have taken the log of the parameters in order to avoid numerical problems during optimisation. Hence we have that:

$$\frac{\partial K_\sigma}{\partial \log\sigma_s} = 2K, \qquad \frac{\partial K_\sigma}{\partial \log\ell_i} = \frac{1}{\ell_i^2}(K \bullet D^{(i)}), \qquad \frac{\partial K_\sigma}{\partial \log\sigma_n} = 2\sigma_n^2 I, \quad (8.18)$$

with $\bullet$ being the Hadamard product and $D^{(i)}(j,k) = (x_i^j - x_i^k)^2$, i.e. a matrix with pairwise distances on dimension i.

Thus, we can make use of equations (8.15), (8.16), and (8.18) (for the case of the SE covariance function) with any gradient-based optimisation method in order to learn the hyperparameters of the GP given some data $\mathcal{D}$.

We have seen that for making predictions (equations (8.8) and (8.9)) as well as for learning hyperparameters (equations (8.15) and (8.16)) one needs to compute the inverse of the Gram matrix K_σ with time complexity $O(N^3)$. As the number of training points increases (e.g. $N > 10000$) this becomes prohibitive and it can be a limitation for applying GPs to real problems. Fortunately, there has been a lot of work in recent years in order to make GPs scalable to large data sets.

8.1.5 Approximation Methods for Large Data Sets

The problem of dealing with large data sets in Gaussian processes has been very much studied in the GP literature, see e.g. Rasmussen and Williams (2006, chapter 8) and Quiñonero-Candela et al. (2007) for an overview. Most of these approximation methods rely on the use of sparse approximations of the Gram matrix where only Q out of N points are selected as *inducing inputs*. For example, in the the Nyström approximation of K in the marginal likelihood we make $K \approx \widetilde{K} \stackrel{\text{def}}{=} K_{\cdot I}(K_{II})^{-1}K_{I\cdot}$, where I indexes Q rows/columns of K. As explained in Rasmussen and Williams (2006, chapter 8), this result is also obtained from both the subset of regressors (SoR) and projected process (PP) approximations for the posterior at the training points. We can then make use of the *Woodbury*'s identity to obtain $(K+\sigma_n^2 I)^{-1} \approx \sigma_n^{-2} - \sigma_n^{-2}K_{\cdot I}\left(\sigma_n^2 K_{II} + K_{I\cdot}K_{\cdot I}\right)^{-1}K_{I\cdot}$, where we note that we need the inversion of a $Q \times Q$ matrix instead of an $N \times N$ matrix.

In practice, selecting Q depends upon time constraints, computing resources and the quality of the solution found for each problem. For example, Rasmussen and Williams (2006, section 8.3.7) obtain good results when using $Q = 4096$ out of $N = 44484$ data-points on a robot arm inverse dynamics problem. Moreover, as pointed out in Rasmussen and Williams (2006, page 183), it is useful to start learning the hyperparameters of a GP with a low number of selected samples Q and to systematically increase this number to see if there is any performance to be gained at the cost of additional computation.

8.2 Single-task Learning

Single-task learning is the standard machine learning set up where given a set of input-output training points $\mathcal{D} = \{(\mathbf{x}_i, y_i)|i = 1,\ldots,N; \mathbf{x} \in \mathbb{R}^d\}$, we wish to learn a regression function $f(\mathbf{x})$ in order to make predictions on unseen data-points $\mathbf{x}_*$. Let us refer to this task as the *test task*, since this is the problem for which we wish to make predictions. Although there may be data from other problems available, we simply do not make use of this data as this may reduce the flexibility of our model.

8.3 Multi-task Learning

Multi-task learning is an area of active research in machine learning and has received a lot of attention over the past few years, see e.g. Thrun and O'Sullivan (1996); Caruana (1997) as some of the earlier references. It also goes under the names of inductive transfer, transfer learning and lifelong learning (Thrun and Pratt, 1998). In addition, it is very prevalent in the statistics literature where it is known under the names of multi-level modelling or hierarchical modelling (see e.g. Goldstein, 2003). A common set up is that there are multiple related tasks

for which we want to avoid tabula rasa learning by sharing information across the different tasks. The hope is that by learning these tasks simultaneously one can improve performance over the single-task (or "no transfer") case, i.e. when each task is learnt in isolation.

However, as pointed out in Baxter (2000) and supported empirically by Caruana (1997), assuming relatedness in a set of tasks and simply learning them together can be detrimental. It is therefore important to have models that will generally benefit related tasks and will not hurt performance when these tasks are unrelated. This is investigated below in the context of Gaussian process (GP) prediction.

The setting is a generalisation of the single-task scenario. Here we have M tasks and a (possibly) different set of input-output pairs $\mathcal{D}^j | j = 1,\ldots,M$. The goal is to learn regression functions $f_j : \mathbf{x} \rightarrow y$ across all the tasks so that we can make predictions at unseen tasks or at unseen points of the training problems.

8.4 Multi-task GP with Task-descriptor Features

Let us consider M tasks for each we wish to learn the mapping $f_j(\mathbf{x})$ for $j = 1,\ldots,M$, where $\mathbf{x}$ is a vector of input features. Additionally, for each task j we have task-descriptor features (or task-specific feature vector) $\mathbf{t}_j$. Thus we can also write $f_j(\mathbf{x}) = f(\mathbf{t}_j,\mathbf{x})$ for some function f. This problem was considered by Bakker and Heskes (2003) using neural network predictors. Here we discuss how to address this problem using kernel machines, more specifically, Gaussian process predictors.

As explained above (section 8.1), there are several advantages for considering Gaussian processes instead of other (parametric) regression methods. In the specific case of neural networks, it is necessary to emphasise that they are quite tricky to train, due to local optima in weight space, the selection of the number of hidden units, hidden layers, etc. In contrast, Gaussian process regression (and many other kernel prediction methods) are underpinned by convex optimisation problems (given the kernel). This is the main motivation to produce a kernel-based solution to this problem.

The main idea here is that given a set of tasks for which input features $\mathbf{x}$ and task-descriptor features $\mathbf{t}$ are available, it is possible to exploit the shared information across these tasks by *directly modelling correlations* between them. For example, as illustrated in Figure 8.1, where $\mathbf{x}$ and $\mathbf{t}$ are shown schematically as one-dimensional variables, for fixed values of $\mathbf{t}$ we obtain sample functions (bold lines) as functions of $\mathbf{x}$. These sample functions for different values of $\mathbf{t}$ are correlated, which contrast to *independent* draws over sample functions if the $\mathbf{t}$-dimension is omitted.

This contrasts with some previous work on multi-task learning with Gaussian processes, where the set of related tasks share a set of *hyperparameters* (Minka and Picard, 1999; Yu

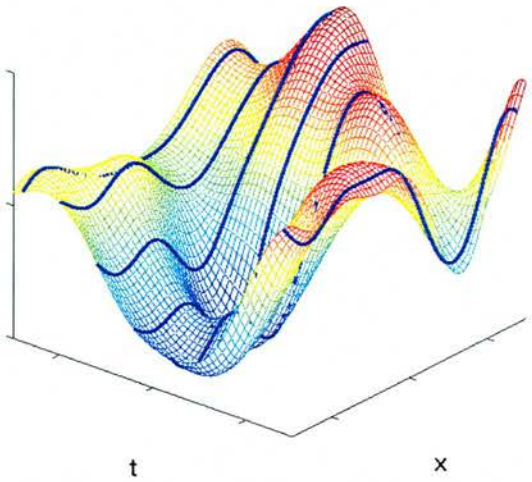

Figure 8.1: A schematic illustration of drawing sample functions in joint $\mathbf{x}$ and $\mathbf{t}$ space. For fixed values of $\mathbf{t}$ we obtain sample functions (bold lines) as functions of $\mathbf{x}$. Note that these sample functions for different values of $\mathbf{t}$ are correlated; this is in contrast to *independent* draws over sample functions if the $\mathbf{t}$-dimension is omitted.

et al., 2005), and the tasks are conditionally independent given the kernel.

8.4.1 The Combined Method

In this method the input features $\mathbf{x}$ and the tasks-descriptor features are simply concatenated into a single vector $\mathbf{z}$ so that $\mathbf{z}^{\mathrm{T}} = (\mathbf{x}^{\mathrm{T}}, \mathbf{t}^{\mathrm{T}})$. All the training data from the individual tasks is then combined into one large training set, and hyperparameter learning as well as predictions with Gaussian processes are performed on this large data set as explained in sections 8.1.3 and 8.1.4.

An interesting situation occurs when the kernel function $k(\mathbf{z}, \mathbf{z}')$ decomposes as $k(\mathbf{z}, \mathbf{z}') = k^f(\mathbf{t}, \mathbf{t}')k^x(\mathbf{x}, \mathbf{x}')$. Therefore, there is a decomposition of the problem into learning task similarity (as measured by k^f) and input similarity (as measured by k^x). An example where such a decomposition occurs is the commonly used squared exponential (or Gaussian) covariance function (see section 8.1.2). This *combined* approach is very similar to *co-kriging*, a well known regression technique in the geostatistics literature (see e.g. Cressie, 1993, section. 3.2.3). Here we use task-specific features in order to define a kernel which incorporates correlations between tasks.

8.4.2 The Gating Network Method

An alternative approach to the combined method is to consider different kernel predictors for each task. Such kernel predictors can be trained independently and then combined using a gating network, similar in spirit to the work by Jacobs et al. (1991).

Let us denote the prediction of expert j (i.e. on task j) for input $\mathbf{x}$ by $p_j(y|\mathbf{x})$, with $j = 1,\ldots,M$. When given a new task with task-descriptor features $\mathbf{t}$ such predictors can be combined by using:

$$p(y|\mathbf{t},\mathbf{x}) = \sum_{j=1}^{M} p(j|\mathbf{t},\mathbf{x}) p_j(y|\mathbf{x}), \tag{8.19}$$

where $p(j|\mathbf{t},\mathbf{x})$ is a gating network outputting mixing proportions that sum to 1. One possible parameterisation of the gating network is given by:

$$p(j|\mathbf{t},\mathbf{x}) \propto \exp\left\{-(\beta_x \|\mathbf{u}_j - \mathbf{x}\|^2 + \beta_t \|\mathbf{t} - \mathbf{t}_j\|^2)\right\}, \tag{8.20}$$

where $\mathbf{u}_j \in \mathbb{R}^{|\mathbf{x}|}$, $\beta_x > 0$ and $\beta_t > 0$ are parameters, and $\mathbf{t}_j$ is the task-specific feature vector for task j. Normalisation is obtained by summing over all reference tasks. Alternatively, one could use a softmax network with input vector $\mathbf{z}$. The gating network can be trained so as to maximise the conditional likelihood of the training data (equation (8.19)) using gradient-based methods.

In contrast to the conditional likelihood training of mixture-of-experts models (Jacobs et al., 1991; Rasmussen and Ghahramani, 2002), our assumption is that the experts $p_j(y|\mathbf{x})$ have been trained separately from the gating network $p(j|\mathbf{t},\mathbf{x})$. The intuition is that by learning the parameters of the gating network we can utilise the information about previously solved related problems for solving new tasks, even when the training data for the new task might be quite limited. This approach is also attractive computationally, as it allows us to quickly mix heterogeneous experts without a need of expensive approximations. Note however that this method is a rather constrained mixture of experts, as the parameters of the experts and the network are not jointly optimised in order to maximise the conditional likelihood.

8.5 Multi-task GP without Task-descriptor Features

Section 8.4 has described multi-task Gaussian process prediction when task-descriptor features $\mathbf{t}$ are used in a parametric covariance function over different tasks. Such a function may be too constrained by both its parametric form and the task descriptors to model task similarities effectively. In addition, for many real-life scenarios task-descriptor features are either unavailable or difficult to define correctly.

This section describes a model that attempts to learn inter-task dependencies based solely on the *task identities* and the observed data for each task. Such model learns a "free-form" task-similarity matrix, which is used in conjunction with a parameterized covariance function over the input features $\mathbf{x}$. This contrasts with the model described in section 8.4 where task relationships are learnt based on a set of task-descriptor features $\mathbf{t}$.

8.5.1 The Model

Let us redefine X as the set of N distinct inputs $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and $\mathbf{y}$ as the complete set of responses for M tasks such that $\mathbf{y} = (y_{11}, \ldots, y_{N1}, \ldots, y_{12}, \ldots, y_{N2}, \ldots, y_{1M}, \ldots, y_{NM})^{\mathrm{T}}$, where $y_{i\ell}$ is the response for the ℓ^{th} task on the i^{th} input $\mathbf{x}_i$. Let us also denote the $N \times M$ matrix Y such that $\mathbf{y} = \mathrm{vec}(Y)$. Given a set of observations $\mathbf{y}_o$, which is a subset of $\mathbf{y}$, we want to predict some of the unobserved response-values $\mathbf{y}_u$ at some input locations for certain tasks.

Let us place a GP prior over the latent functions $\{f_\ell\}$ so that we directly model the correlations between the tasks. Assuming that the GPs have zero mean we set

$$\langle f_\ell(\mathbf{x}) f_m(\mathbf{x}') \rangle = K^f_{\ell m} k^x(\mathbf{x}, \mathbf{x}') \qquad y_{i\ell} \sim \mathcal{N}(f_\ell(\mathbf{x}_i), \sigma^2_\ell), \tag{8.21}$$

where K^f is a positive semi-definite (PSD) matrix that specifies the inter-task similarities, k^x is a covariance function over inputs, and σ^2_ℓ is the noise variance for the ℓ^{th} task. Below we focus on *stationary* covariance functions k^x; hence, to avoid redundancy in the parameterisation, we further let k^x be only a *correlation* function (i.e. it is constrained to have unit variance), since the variance can be explained fully by K^f.

The important property of this model is that the joint Gaussian distribution over $\mathbf{y}$ is not block-diagonal with respect to the tasks, so that observations of one task can affect the predictions on another task. In Yu et al. (2007); Bonilla et al. (2007) this property also holds, but instead of specifying a general PSD matrix K^f, these authors set $K^f_{\ell m} = k^f(\mathbf{t}_\ell, \mathbf{t}_m)$, where $k^f(\cdot, \cdot)$ is a covariance function over the task-descriptor features $\mathbf{t}$.

One popular setup for multi-task learning is to assume that tasks can be clustered, and that there are inter-task correlations between tasks in the same cluster. This can be easily modelled with a general task-similarity K^f matrix: if we assume that the tasks are ordered with respect to the clusters, then K^f will have a block diagonal structure. Of course, as we are learning a "free form" K^f the ordering of the tasks is irrelevant in practice (and is only useful for explanatory purposes).

For scenarios where the number of input observations is small, multi-task learning augments the data set with a number of different tasks, so that model parameters can be estimated more confidently; this helps to minimise over-fitting. In the model presented here, this is achieved by having a common covariance function over the features $\mathbf{x}$ of the input observations. This contrasts with the semiparametric latent factor model (SLFM) of Teh et al. (2005), where with the same set of input observations, one has to estimate the parameters of several covariance functions belonging to different latent processes (see section 8.7 for details).

8.5.2 Predictions

Predictions in this model can be done by using the standard GP formulae for the mean and variance of the predictive distribution as given in equations (8.8) and (8.9) with the covariance

function given in equation (8.21). Thus, the mean and variance of the predictive distribution on a new data-point $\mathbf{x}_*$ for task ℓ are given by:

$$\mathbb{E}[f_\ell(\mathbf{x}_*)] = (\mathbf{k}_\ell^f \otimes \mathbf{k}_*^x)^T \Sigma^{-1} \mathbf{y}, \tag{8.22}$$

$$\mathbb{V}[f_\ell(\mathbf{x}_*)] = K_{\ell\ell}^f k^x(\mathbf{x}_*, \mathbf{x}_*) - (\mathbf{k}_\ell^f \otimes \mathbf{k}_*^x)^T \Sigma^{-1} (\mathbf{k}_\ell^f \otimes \mathbf{k}_*^x), \tag{8.23}$$

$$\text{with } \Sigma = K^f \otimes K^x + D \otimes I, \tag{8.24}$$

where $\otimes$ denotes the Kronecker product, $\mathbf{k}_\ell^f$ selects the ℓ^{th} column of K^f, $\mathbf{k}_*^x$ is the vector of covariances between the test point $\mathbf{x}_*$ and the training points, K^x is the matrix of covariances between all pairs of training points, D is an $M \times M$ diagonal matrix in which the $(\ell,\ell)^{\text{th}}$ element is σ_ℓ^2, and Σ is an $MN \times MN$ matrix.

8.5.3 Learning Hyperparameters

Given the set of observations $\mathbf{y}_o$, we wish to learn the parameters $\boldsymbol{\theta}_x$ of k^x and the matrix K^f to maximise the marginal likelihood $p(\mathbf{y}_o|X, \boldsymbol{\theta}_x, K^f)$ (equation (8.15)). One way to achieve this is to use the fact that $\mathbf{y}|X \sim \mathcal{N}(\mathbf{0}, \Sigma)$. Therefore, gradient-based methods can be readily applied to maximise the marginal likelihood. In order to guarantee positive-semidefiniteness of K^f, one possible parameterisation is to use the Cholesky decomposition $K^f = LL^T$ where L is lower triangular. Computing the derivatives of the marginal likelihood with respect to L and $\boldsymbol{\theta}_x$ is straightforward. A drawback of this approach is its computational cost as it requires the inversion of a matrix of potential size $MN \times MN$ (or solving an $MN \times MN$ linear system) at each optimisation step. Note, however, that one only needs to actually compute the Gram matrix and its inverse at the visible locations corresponding to $\mathbf{y}_o$.

Alternatively, it is possible to exploit the Kronecker product structure of the full covariance matrix as in Zhang (2007), where an Expectation-Maximisation (EM, Dempster et al., 1977) algorithm[1] is proposed such that learning of $\boldsymbol{\theta}_x$ and K^f in the M-step is decoupled. This has the advantage that closed-form updates for K^f and D can be obtained and that K^f is guaranteed to be positive-semidefinite.

We have seen that Σ needs to be inverted (in time $O(M^3N^3)$) for both making predictions and learning the hyperparameters (when considering noisy observations). This can lead to computational problems if MN is large. In section 8.5.5 we give some approximations that can help speed up these computations.

[1] An Expectation-Maximisation algorithm attempts to find the maximum likelihood estimates of the parameters of a probabilistic model that contain *latent variables* by iteratively performing two steps: (E-step) computing the expected complete data log-likelihood and (M-step) estimating the parameters that maximise this expected data log-likelihood.

8.5.4 Noiseless Observations and the Cancellation of Inter-task Transfer

One particularly interesting case to consider is noise-free observations at the same locations for all tasks (i.e. a block-design) so that $\mathbf{y}|X \sim \mathcal{N}(\mathbf{0}, K^f \otimes K^x)$. In this case maximising the marginal likelihood $p(\mathbf{y}|X)$ with respect to the parameters $\boldsymbol{\theta}_x$ of k^x reduces to maximising $-M\log|K^x| - N\log|Y^T(K^x)^{-1}Y|$, an expression that does not depend on K^f. After convergence we can obtain K^f as $\hat{K}^f = \frac{1}{N}Y^T(K^x)^{-1}Y$. The intuition behind is this: the responses Y are correlated via K^f and K^x. We can learn K^f by decorrelating Y with $(K^x)^{-1}$ first so that only correlation with respect to K^f is left. Then K^f is simply the sample covariance of the decorrelated Y.

Unfortunately, in this case there is effectively no transfer between the tasks (given the kernels). To see this, consider making predictions at a new location $\mathbf{x}_*$ for all tasks. We have (using the mixed-product property of Kronecker products) that

$$\bar{\mathbf{f}}(\mathbf{x}_*) = \left(K^f \otimes \mathbf{k}_*^x\right)^{\mathrm{T}} \left(K^f \otimes K^x\right)^{-1} \mathbf{y} \tag{8.25}$$

$$= \left((K^f)^{\mathrm{T}} \otimes (\mathbf{k}_*^x)^{\mathrm{T}}\right) \left((K^f)^{-1} \otimes (K^x)^{-1}\right) \mathbf{y} \tag{8.26}$$

$$= \left[\left(K^f (K^f)^{-1}\right) \otimes \left((\mathbf{k}_*^x)^{\mathrm{T}} (K^x)^{-1}\right)\right] \mathbf{y} \tag{8.27}$$

$$= \begin{pmatrix} (\mathbf{k}_*^x)^{\mathrm{T}} (K^x)^{-1} \mathbf{y}_{\cdot 1} \\ \vdots \\ (\mathbf{k}_*^x)^{\mathrm{T}} (K^x)^{-1} \mathbf{y}_{\cdot M} \end{pmatrix}, \tag{8.28}$$

and similarly for the covariances. Thus, in the noiseless case with a block design, the predictions for task ℓ depend only on the targets $\mathbf{y}_{\cdot\ell}$. In other words, there is a cancellation of transfer. One can in fact generalise this result to show that the cancellation of transfer for task ℓ does still hold even if the observations are only sparsely observed at locations $X = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ on the other tasks. This result is known as *autokrigeability* in the geostatistics literature (Wackernagel, 1998), and is also related to the *symmetric Markov property* of covariance functions that is discussed in O'Hagan (1998). It is necessary to emphasise that if the observations are noisy, or if there is not a block design, then this result on cancellation of transfer will not hold. This result can also be generalised to multidimensional tensor product covariance functions and grids (Williams et al., 2007).

8.5.5 Constraining the Number of Parameters and Approximations

Specifying a full rank K^f requires $M(M+1)/2$ parameters, and for large M this would be a lot of parameters to estimate. One parameterisation of K^f that reduces this problem is to use a `PPCA` model (Tipping and Bishop, 1999) $K^f \approx \widetilde{K}^f \stackrel{\text{def}}{=} U\Lambda U^{\mathrm{T}} + s^2 I_M$, where U is an $M \times P$ matrix of the P principal eigenvectors of K^f, Λ is a $P \times P$ diagonal matrix of the corresponding eigenvalues, and s^2 can be determined analytically from the eigenvalues of K^f (see Tipping

and Bishop, 1999, and references therein). For numerical stability, we may further use the incomplete-Cholesky decomposition setting $U\Lambda U^{\mathrm{T}} = \tilde{L}\tilde{L}^{\mathrm{T}}$, where $\tilde{L}$ is a $M \times P$ matrix. In section 8.8.6 we will consider the case $s = 0$, i.e. rank-P approximation to K^f.

Additionally, if the number of input points N is large, we can make use of GP approximations to K^x in order to speed up computations (as explained in section 8.1.5). Here we propose the use of the Nyström approximation of K^x in the marginal likelihood, so that $K^x \approx \widetilde{K}^x \stackrel{\text{def}}{=} K^x_{\cdot I}(K^x_{II})^{-1}K^x_{I\cdot}$, where I indexes Q rows/columns of K^x. In fact for the posterior at the training points this result is obtained from both the subset of regressors (SoR) and projected process (PP) approximations described in Rasmussen and Williams (2006, chapter 8).

Applying both approximations to get $\Sigma \approx \widetilde{\Sigma} \stackrel{\text{def}}{=} \tilde{K}^f \otimes \tilde{K}^x + D \otimes I_N$, we have, after using the *Woodbury* identity, $\widetilde{\Sigma}^{-1} = \Delta^{-1} - \Delta^{-1}B\left[I \otimes K^x_{II} + B^{\mathrm{T}}\Delta^{-1}B\right]^{-1}B^{\mathrm{T}}\Delta^{-1}$ where $B \stackrel{\text{def}}{=} (\tilde{L} \otimes K^x_{\cdot I})$, and $\Delta \stackrel{\text{def}}{=} D \otimes I_N$ is a diagonal matrix. As $\tilde{K}^f \otimes \tilde{K}^x$ has rank PQ, we have that the computation of $\tilde{\Sigma}^{-1}\mathbf{y}$ takes $O(MNP^2Q^2)$.

8.6 Quantifying Inter-task Transfer

It is possible to quantify the amount of inter-task transfer that is taking place when making predictions with multi-task GP. Consider the mean prediction given in equation (8.8), which can be rewritten as:

$$\mathbb{E}[f_*(i,\mathbf{x}_*)] = \mathbf{h}^T(i,\mathbf{x}_*)\mathbf{y}, \tag{8.29}$$

where $\mathbf{h}(i,\mathbf{x}_*)$ is the weight function (see e.g. Rasmussen and Williams, 2006, section 2.6). Note that in the case of multi-task GP with task-specific features $h(\mathbf{z}_*) = h(i,\mathbf{x}_*)$. If we order the training points according to the task they belong to, then $\mathbf{h}$ can be partitioned as $\mathbf{h}^{\mathrm{T}}(i,\mathbf{x}_*) = (h^1_1,\ldots,h^1_N,\ldots,h^M_1,\ldots,h^M_N)$ and similarly for $\mathbf{y}$, where the superscript identifies the task. We can now measure the contribution of task j when making prediction on test point $\mathbf{x}_*$ belonging to task i by computing:

$$r^j(i,\mathbf{x}_*) = \frac{\|\mathbf{h}^j(i,\mathbf{x}_*)\|}{\|\mathbf{h}(i,\mathbf{x}_*)\|}. \tag{8.30}$$

Averaging $r^j(i,\mathbf{x}_*)$ over test points gives a summary measure $\bar{r}^j$ for the contribution of task j to the test problem. We prefer this measure as compared to looking directly at K^f as the interpretation of K^f is complicated by the inversion of the kernel matrix in GP prediction.

8.7 Related Work

There has been a lot of work in recent years on multi-task learning (or inductive transfer) using methods such as Neural Networks, Gaussian Processes, Dirichlet Processes and Support Vector Machines, see e.g. Caruana (1997); Thrun (1996) for early references. The key issue concerns what properties or aspects should be shared across tasks.

As mentioned in section 8.4, multi-task learning with task-specific features is discussed in Bakker and Heskes (2003), but using neural network predictors. They propose two ways to use the task-specific features: One is to define a task-specific prior in weight space (section 4.1 in their paper). The second is to use a gating network (although in their case this only depended on $\mathbf{t}$ and not on $\mathbf{x}$). Note that neither of these methods introduces inter-task correlations in the prior. Yu et al. (2007) have recently investigated the combined method discussed above under the assumption of factorisation of the kernel with respect to $\mathbf{x}$ and $\mathbf{t}$ features. In their case the setup was as a relational model, e.g. for predicting movie ratings based on user and movie features.

There is also a lot of work on multi-task learning when there are no task-specific features. In this case one can make various assumptions about how to induce transfer between tasks. For example Minka and Picard (1999) assumed that a number of related tasks shared the same kernel parameters, and these were optimised on the set of tasks available. In a similar vein, Yu et al. (2005) induced transfer between tasks by assuming a common covariance for the tasks, with a Normal-Inverse-Wishart prior. However, note that in these cases the different tasks are conditionally independent given the kernel; in contrast our methods discussed above are stronger in that they directly model correlations between the tasks.

Within the GP literature, Minka and Picard (1999); Lawrence and Platt (2004); Yu et al. (2005); Schwaighofer et al. (2005); Yu et al. (2006) give models where the covariance matrix of the full (noiseless) system is block diagonal, and each of the M blocks is induced from the same kernel function. Under these models each $\mathbf{y}_{\cdot i}$ is conditionally independent, but inter-task tying takes place by sharing the kernel function across tasks. In contrast, in our models and in Teh et al. (2005); Yu et al. (2007) the covariance is not block diagonal.

The semiparametric latent factor model (SLFM, Teh et al., 2005) involves having P latent processes (where $P \leq M$) and each of these latent processes has its own covariance function. The noiseless outputs are obtained by linear mixing of these processes with a $M \times P$ matrix Φ. The covariance matrix of the system under this model has rank at most PN, so that when $P < M$ the system corresponds to a degenerate GP. Our model is similar to Teh et al. (2005) but simpler, in that all of the P latent processes share the same covariance function; this reduces the number of free parameters to be fitted and should help to minimise overfitting. With a common covariance function k^x, it turns out that K^f is equal to $\Phi\Phi^{\mathrm{T}}$, so a K^f that is strictly positive definite corresponds to using $P = M$ latent processes. Note that if $P > M$ one can always find an $M \times M$ matrix Φ' such that $\Phi'\Phi'^{\mathrm{T}} = \Phi\Phi^{\mathrm{T}}$. We note also that the approximation methods used in Teh et al. (2005) are different to ours, and were based on the subset of data (SoD) method using the informative vector machine (IVM) selection heuristic.

In the geostatistics literature, the prior model for $f_{\cdot}$ given in equation (8.21) is known as the *intrinsic correlation model* (Wackernagel, 1998), a specific case of *co-kriging*. A sum of

such processes is known as the *linear coregionalization model* (LCM, Wackernagel, 1998) for which Zhang (2007) gives an EM-based algorithm for parameter estimation. Our model for the observations corresponds to an LCM model with two processes: the process for f_i and the noise process. Note that SLFM can also be seen as an instance of the LCM model. To see this, let E_{pp} be a $P \times P$ diagonal matrix with 1 at (p,p) and zero elsewhere. Then we can write the covariance in SLFM as $(\Phi \otimes I)(\sum_{p=1}^{P} E_{pp} \otimes K_p^x)(\Phi \otimes I)^{\mathrm{T}} = \sum_{p=1}^{P}(\Phi E_{pp}\Phi^{\mathrm{T}}) \otimes K_p^x$, where $\Phi E_{pp}\Phi^{\mathrm{T}}$ is of rank 1.

There are also other assumptions one can make about ways to share information between tasks; for example, one can consider mixture models for task clustering (Bakker and Heskes, 2003). Evgeniou et al. (2005) consider methods for inducing correlations between tasks based on a correlated prior over linear regression parameters; In fact this corresponds to a GP prior using the kernel $k(\mathbf{x},\mathbf{x}') = \mathbf{x}^T A \mathbf{x}'$ for some positive definite matrix A. In their experiments they use a restricted form of K^f with $K^f_{\ell m} = (1-\lambda) + \lambda M \delta_{\ell m}$ (their equation 25), i.e. a convex combination of a rank-1 matrix of ones and a multiple of the identity. Notice the similarity to the PPCA form of K^f given in section 8.5.5.

8.8 Compiler Optimisation as a Performance Prediction Problem

As mentioned at the beginning of this chapter, one possible approach to combinatorial optimisation is to use a regression function $f_j(\mathbf{x})$ in order to approximate the evaluation function of optimisation problem j. Thus, such a regression function can be used as a proxy for the actual evaluation function. Clearly, this is particularly useful when the evaluation function is very expensive to execute. For example, in the case of the compiler optimisation problem each function evaluation requires the compilation and execution of a program. However, if the space of solutions $\mathcal{X}$ is very large then even if this regression function can be learnt accurately it would be very time consuming to find the $\mathbf{x}$ that optimises $f_j(\mathbf{x})$. Thus, in cases where the search space is very large it will be necessary to use the learned performance predictor as a proxy for the evaluation function within search or sampling methods.

8.8.1 Formulation: The Performance Prediction Problem

As stated above, we are interested in learning the mapping $\mathbf{x} \rightarrow y$ for problem j with a regression function $f_j(\mathbf{x})$ that can be used to make predictions on unseen inputs $\mathbf{x}_*$. For the compiler optimisation problem $\mathbf{x} \in \mathcal{X}$ is a representation of a transformation sequence and y is the performance *speed-up* of a program j under the application of the given sequence of compiler transformations. Therefore, our regression task is to predict the speed-up of a given program under transformation $\mathbf{x}$. Let us refer to this task as the *performance prediction problem.*

Since we are interested in the Multi-task learning setting we will consider a collection of

optimisation problems (or tasks) in order to build a regression function on all these problems as described in sections 8.4 and 8.5. We identify each task as a specific program (i.e. the performance prediction problem on a specific program) and the input for each task is given by sequences of transformations **x**. Below, we provide details of the set up for both models (multi-task GP with task-descriptor features and multi-task GP without task-descriptor features) and an empirical evaluation on the small space of the SUIF data set on the TI architecture. See chapter 5 for details on the experimental set up under which this data set has been generated.

8.8.2 Input Features

Two different codings of the sequence of transformations into the **x**-vector of features have been used: a code-features representation (C) and a transformation-based representation (T).

The **code-features representation** (C) is obtained by computing features believed by compiler experts to be informative about program performance. These features are extracted from the *transformed* version of the program after applying a transformation sequence so that they encode the result of each sequence.

As mentioned at the beginning of Chapter 5, the input features used for distinct machine learning techniques may well be very different. We have found experimentally that the features used for the Predictive Search Distribution approach (see Table 7.2) were not sufficiently informative about the effect of the transformations used on the performance of the programs. Therefore, another set of features has been used for the regression approach to compiler optimisation. These code features are based on three distinct metrics: *code size*, the total number of *instructions executed* and the *parallelism* existing among those instructions. For each of these metrics, a vector of high-level machine-independent instructions is derived by using the SUIF compiler infrastructure (Hall et al., 1996). Our high-level instructions correspond to the ones used by the SUIF Intermediate Representation (IR). A total of 83 features were extracted per program. These were reduced to 18 features using PCA, retaining 95% of the variance. One advantage of the code-features representation is that these features can be extracted even if new code transformations that have not been seen before are applied.

The **transformation-based representation** (T) used considers a bag-of-characters representation where the presence or absence of the transformations within a sequence is recorded in a binary vector. Clearly, this throws away the ordering information within a sequence and one might lose predictive performance due to this loss of information. However, using more complex representations that, for example include the order information or the positional information, would require larger amounts of data for training.

8.8.3 Task-descriptor Features

For multi-task GP with task-descriptor features it is necessary to define the task descriptors (or task-specific feature vector) $\mathbf{t}$ used to describe a program. A response-based approach in order to extract useful descriptors for a program is presented below.

The Canonical Responses

Task-descriptor features are defined by selecting a small number of sequences and recording the corresponding speed-ups on the given task. We will refer to this set of sequences as *canonical sequences* and to their corresponding speed-ups as *canonical responses*. We select the set of canonical transformation sequences using the technique of *principal variables* (McCabe, 1984).

Here (as in PCA) the dimensionality reduction problem is formulated as the linear mapping from a high-dimensional vector (the speedups for all sequences) to a lower dimensional one. However, in this case the linear mapping simply copies some of the variables and discards the rest, hence the term "principal variables". McCabe considers a number of different criteria for selecting a subset of variables. Here we choose the set of included variables $S_{(1)}$ so as to maximise the determinant of the covariance matrix of these variables, i.e. $|\Sigma_{S_{(1)}}|$. This may be interpreted as maximisation of a Gaussian approximation to the mutual information $I(S_{(1)};T)$ which the retained variables $S_{(1)}$ contain about the identity of the task T. (An alternative criterion is to minimise the trace of the conditional covariance matrix of the discarded variables given the selected variables, i.e. $\mathrm{tr}(\Sigma_{S_{(2)}|S_{(1)}})$ where $S_{(2)}$ denotes the set of variables discarded.) As searching for the optimal partition is computationally expensive, the suggestion in McCabe (1984) is followed and a greedy forward selection strategy to select the subset is used.

As specified above, the canonical responses would be extracted using all the sequences on each of the training problems. However, it has been found that using canonical responses extracted from only 2048 randomly selected sequences yields almost as good performance as extraction from the larger set. In the experiments, 8 canonical variables are used.

The response-based approach is not the only one that we can consider for defining task-specific features. For instance we might describe each untransformed program with code features (see section 8.8.2). Notice that in contrast to section 8.8.2, this would be characterising a *program*, not a *sequence of transformations*. However, experimentally, it has been found that the response-based method is superior.

8.8.4 Evaluation Set Up

In the sequel, let us refer to multi-task GP with task-descriptor features as the **task-descriptor** method and to multi-task GP without task-descriptor features as the **free-form** method.

Task-descriptor method

Leave-one-out *cross-validation* (LOO-CV) has been used for evaluating the performance of the models, so for each LOO-CV experiment there are $M = 11 - 1 = 10$ reference tasks[2]. Obviously, it would be impractical to have a system that relies on exhaustive training data for making predictions. Therefore, we have sampled the space of each benchmark and investigated different sample sizes; below the results for $n_r = 256$ training points per benchmark are reported. Note also that in this set up the canonical sequences were obtained in the LOO-CV framework, so that for a given test task the canonicals were extracted from only the 10 reference tasks.

In addition to the data from the reference problems, access to varying amounts of data from the test problem has also been considered. These points are chosen according to the ordered list of the canonical sequences. A minimum of 8 canonicals are required so as to define the task-specific features, but more can also be used as described in section 8.8.5 below.

For this particular application, in a real-life scenario it is critical to achieve good performance with a low number of data-points from the test task, given that the inclusion of this requires the compilation and execution of a (potentially) different version of a program. Therefore, sizes of $n_{te} = 8$, 16, 32, 64 and 128 points from the test problem have been investigated.

As one of our goals is to evaluate inter-task transfer, we must also consider what performance can be obtained using only these small amounts of test data (i.e. the "no transfer" case). To do this we use GP regression, and also consider a simple baseline predictor based on the median[3] of the speed-ups on the n_{te} canonical sequences. However, as the choice of canonical sequences depends on the reference problems, we also compare with the median speed-up of all the sequences on each test problem. The measure of performance used is the mean absolute error (MAE) computed over all the (n_*) sequences excluding the examples from the test problem used for training. Figure 8.2 illustrates the multi-task setting for this method.

Free-form method

For this scenario only $N = 16, 32, 64$ and 128 transformation sequences per program have been used for training as $N = 8$ would clearly be insufficient to estimate the parameters in the model. All the $M = 11$ programs (tasks) have been used for training, and predictions have been done at the (unobserved) remaining inputs, i.e. at all the sequences excluding the N examples from the test problem used for training. Due to the variability of the results depending on training set selection, 10 different replications have been done.

[2]The benchmark *mult* has not been included given that there is no variability in its speed–up values.

[3]The median is the optimal value to minimise mean absolute error; the mean is optimal for mean squared error.

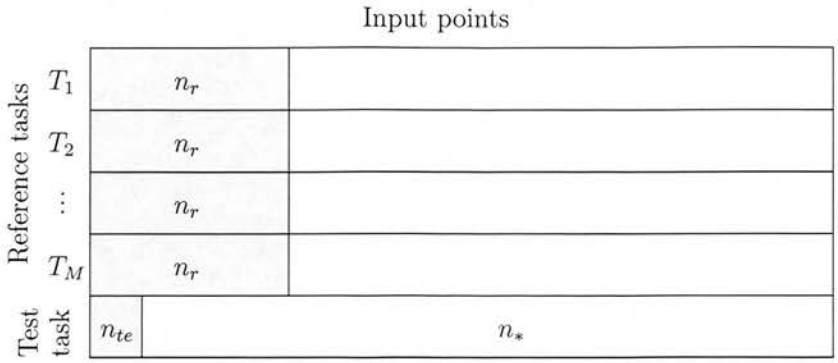

Figure 8.2: The multi-task scenario for the task-descriptor method. Training is done by including n_r data-points from each reference task and n_{te} data-points from the test task. The goal is to make predictions on the unseen n_* data-points on the test task.

8.8.5 Methods Used

For the **task-descriptor** method, multi-task Gaussian process regression has been used as explained in section 8.4. A squared exponential covariance function with automatic relevance determination (ARD) has been used as explained in section 8.1.2. Prediction with linear regression models was also tried, but this gave inferior performance to the GPs. For example, the average mean absolute errors (MAE) across programs for the combined method when using GPs and linear regression were 0.0576 and 0.1207 respectively. Hyperparameter learning has been performed by maximising the marginal log-likelihood (equation 8.15) of the data from the reference problems and the n_{te} test points.

For the "no transfer" case using only n_{te} points from the test problem there is a danger that the full ARD model would involve too many parameters and thus be in danger of overfitting. In this case for $n_{te} \leq 64$, an isotropic covariance function (with all ℓ_i's tied) was used instead of the ARD model. Experiments showed that the performance of the isotropic-no-transfer GP predictor was better than or equal to the ARD-no-transfer model.

For the gating network the setting given by equation (8.20) has been used. The gating network has been trained to maximise the conditional likelihood of the training data (equation (8.19)) using gradient-based methods. In addition to the data from the M reference data sets, the speed-ups obtained on the canonical sequences of the new task have also been included to train this network; here the new task has been given equal weight to the other tasks, even though there is less training data, so as to emphasise the importance of the task-specific data.

For the **free-form** method a squared-exponential (or Gaussian) covariance function k^x with ARD has been used and a free-form for K^f has been adopted. In order to constrain the number of parameters in K^f a Cholesky decomposition $K^f = LL^T$ has been used. As in the feature-based case, hyperparameters of the models $\boldsymbol{\theta}_x, L, D$ have been learnt so as to maximise the marginal likelihood $p(\mathbf{y}_o|X, K^f, \boldsymbol{\theta}_x)$ using gradient-based search in MATLAB with Carl Ras-

mussen's `minimize.m`[4]. In the experiments this method usually outperformed EM in the quality of the solutions found and in the speed of convergence.

Where relevant the hyperparameters were initialised as follows: all the length scales were initialised to 1, the signal variance was initialised to 1, and the noise variance to 0.01 (and for the free-form method all σ_l^2 were constrained to be equal). Additionally, for the free-form method K^f was initialised (given $k^x(\cdot,\cdot)$) by using the noise-free expression $\hat{K}^f = \frac{1}{N} Y^T (K^x)^{-1} Y$ given in section 8.5.4 .

8.8.6 Results

Below we first present results for multi-task learning with task-descriptor features with both the combined and gating methods, using either code features (C) or transformations (T) as the representation for $\mathbf{x}$. Afterwards, the results of multi-task GP without using task-descriptor features are compared with the feature-based method.

Predictions with the task-descriptor method

Figure 8.3 shows results for the case when $n_{te} = 8$, i.e. we use a very small amount of data from the test problem. The four methods T-combined, C-combined, T-gating and C-gating are shown along with the two baselines: median (of the 8 canonical responses) and median (of all test data). Note that there are error bars on the four transfer methods due to the random selection of the $n_r = 256$ training points from each of the reference problems; 10 repetitions were used to assess this variability. The median of the test data gives the best possible MAE for a given test problem without looking at the $\mathbf{x}$ data; it defines a reference point but requires all speed-ups from the test problem to compute it, rather than the 8 which are available to the transfer methods (and median canonicals).

In Figure 8.3 we see the trend that those problems with higher variability (as seen in Figures 6.1 and 6.2) generally have a larger error. T-gating generally performs worse than T-combined. For the code features representation, we observe that C-gating generally performs better than C-combined. The best performing transfer method on average is T-combined. This gives some significant improvements over the median predictors, particularly on problems *compress*, *fir*, *histogram*, *latnrm*, and *lmsfir*, and gives similar performance to the medians for the other problems. Its average MAE performance is 0.0576 compared to 0.1162 for the average of the median canonicals. Note that Bakker and Heskes (2003) only present results which are aggregated over all tasks, rather than a more detailed decomposition like the one given here.

One reason why the gating network approach may be limited is that as the component predictors are trained on the individual reference tasks, it may not be expected to generalise well if the pattern of speed-ups on the test problem is very different from those of the reference

[4]This can be downloaded from `http://www.gaussianprocess.org/gpml/code/gpml-matlab.zip`.

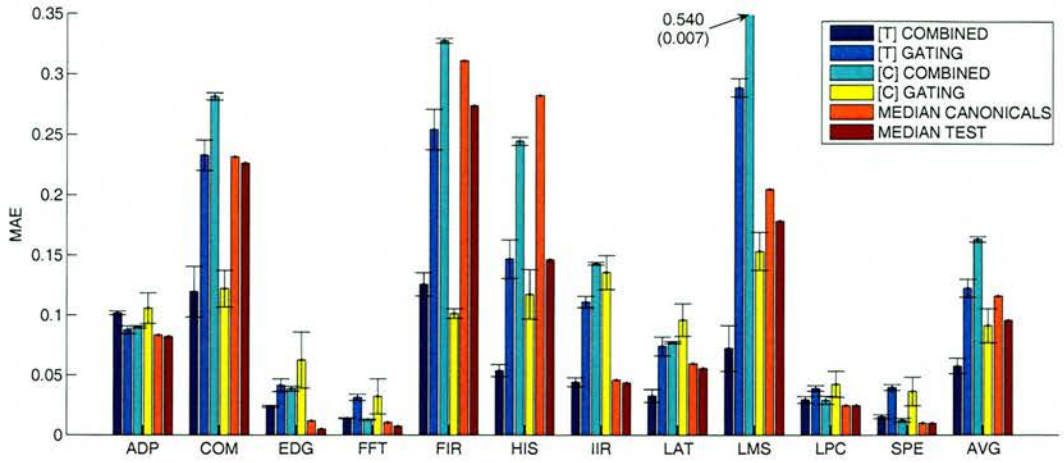

Figure 8.3: Mean absolute error (MAE) on each of the problems (programs) and the average on the small space of the SUIF data set on the TI board, for the 4 methods T-combined, C-combined, T-gating and C-gating and the two baselines median (canonical) and median (of all test data). The error bars denote one standard deviation. The error bars for the averages are computed as the average of the standard deviations over all problems. [C] denotes the code-features representation and [T] the transformations representation. For C-combined on LMS the MAE is 0.54, with standard deviation of 0.007. The results on the program *mult* are not reported as there is no variability on the speed-ups of this benchmark.

problems. This might be overcome by joint training of the component predictors and the gating network, as in the mixture of experts architecture (Jacobs et al., 1991).

Figures 8.4 and 8.5 show in more detail the performance of various methods as a function of n_{te}. They show the performance of the T-combined and two T-no-transfer methods, along with the median (canonicals) baseline for each of the 11 problems; the bottom right-hand panel in Figure 8.5 shows the averages. Note that, as in Figure 8.3, the results on the benchmark *mult* have not been reported as there is no variability in the speed-ups of this program.

The T-no-transfer-canonicals method uses a GP predictor trained on only the n_{te} canonical sequences; consequently there are no error bars for these curves. In contrast, T-no-transfer-random is trained on a set randomly selected data points of size n_{te}. Generally the performance of T-no-transfer-canonicals is superior to T-no-transfer-random.

The plots in Figures 8.4 and 8.5 reinforce the message of Figure 8.3. They also show that for those problems where transfer is significant, this advantage tends to disappear when n_{te} reaches higher values of around 128. However, note that for the compilers application it is desirable to make as few runs as possible on the test problem, so it is definitely the small n_{te} values that are most relevant in this case. Thus we can conclude that for the T-combined method transfer learning generally either improves performance or leaves it about the same in comparison to the T-no-transfer-canonicals method.

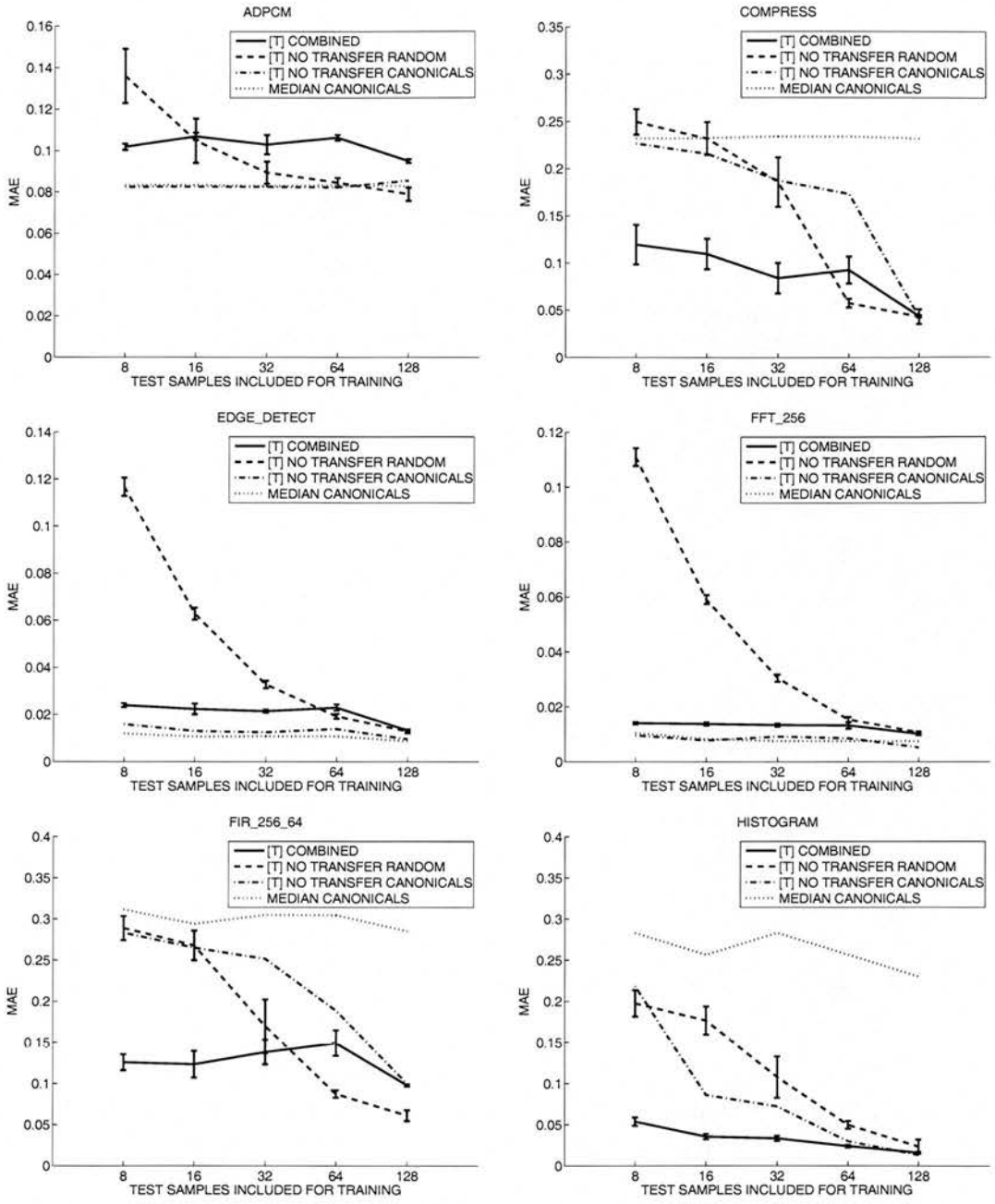

Figure 8.4: The performance of the T-combined (multi-task GP with task-descriptor features) and T-no-transfer methods and median (canonicals) as a function of n_{te} on the small space of 6 problems (programs) of the SUIF data set on the TI board. The error bars denote one standard deviation.

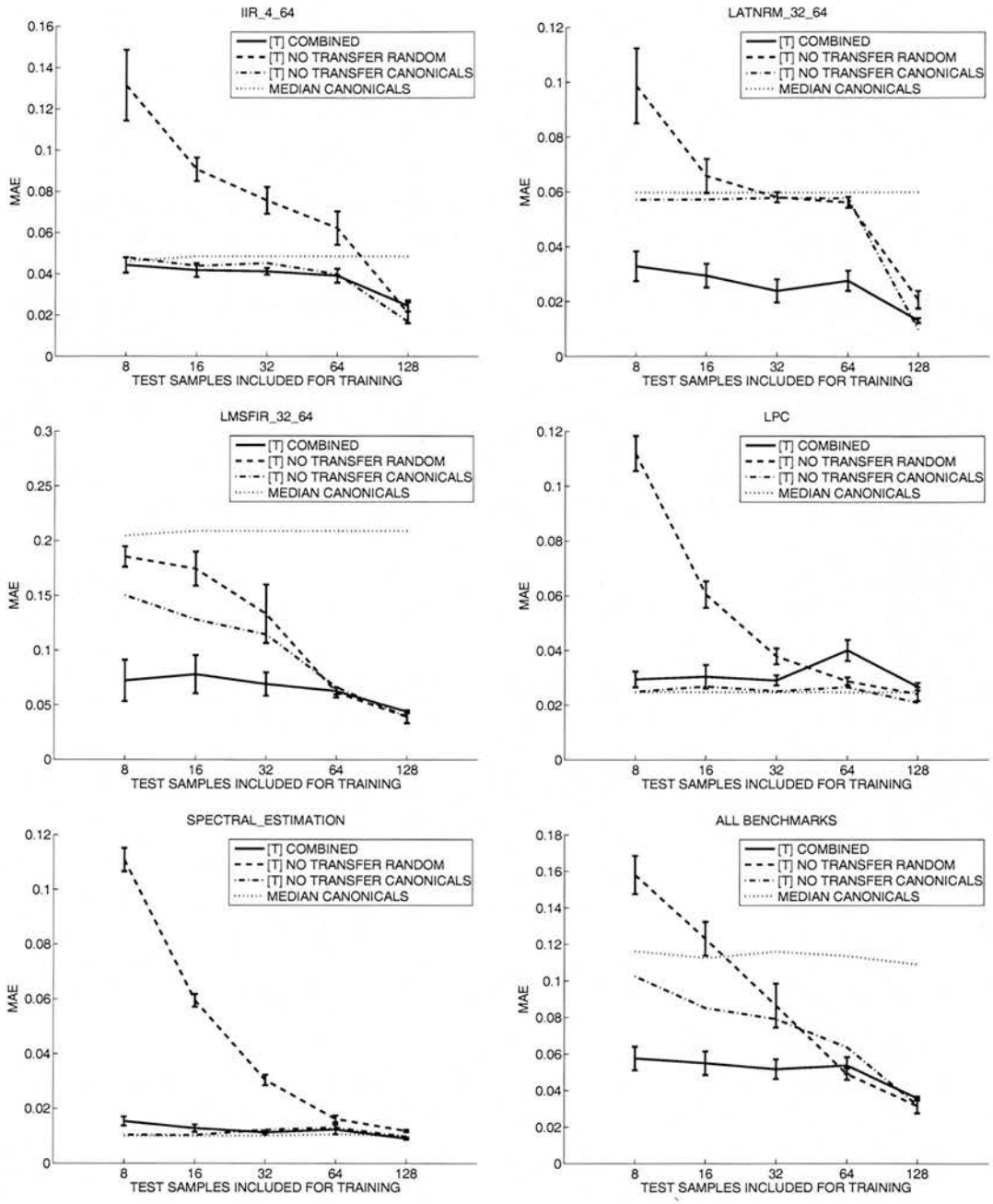

Figure 8.5: The performance of the T-combined (multi-task GP with task-descriptor features) and T-no-transfer methods and median (canonicals) as a function of n_{te} on the small space of the remaining 5 problems (programs) of the SUIF data set on the TI board. The bottom right panel shows the average performances. The error bars denote one standard deviation.

Predictions with the free-form method

Figures 8.6 and 8.7 show the mean absolute errors obtained on the compiler data for every task and on average all the tasks (Figure 8.7 bottom right). As before, the results on the benchmark *mult* are not reported due to the lack of variability in the speed-ups of this program.

Let us take for example *histogram* (bottom right of Figure 8.6) where learning the tasks simultaneously brings major benefits over the no transfer case. Here, the free-form multi-task GP method provides a reduction on the mean absolute error of up to 6 times. Additionally, it is consistently (although only marginally) superior to the task-descriptor approach. For *fir* (bottom left of figure 8.6), the free-form method not only significantly outperforms the no transfer case but also provides greater benefits over the task-descriptor method (which for $N = 64$ and 128 is worse than no transfer). The benchmark *adpcm* (top left of figure 8.6) is the only case out of all 11 tasks where the free-form method degrades performance, although it should be noted that all the methods perform similarly. Further analysis of the data indicates that learning on this task is hard as there is a lot of variability that cannot be explained by the bag-of-characters representation used for the input features. Finally, for *all tasks* (Figure 8.7 bottom right) on average the free-form method brings significant improvements over single task learning and consistently outperforms the task-descriptor method. For all tasks except one the model provides better or roughly equal performance than the non-transfer case and the task-descriptor model.

Using the Predictions for Optimisation

It has been shown so far that multi-task GP brings further benefits over the no transfer scenario on the compiler performance prediction problem. Here we aim to evaluate the performance of these models on the compiler optimisation problem, i.e. when using such performance predictors in order to find a good set of transformation sequences.

As shown in equation (8.8), predictions with Gaussian processes require the inversion of the Gram matrix once, which can be stored and used in conjunction with the test covariances in order to compute a linear combination of the training targets. Therefore, this prediction procedure can be used in order to estimate the expected performance of a potentially large set of transformation sequences. Those transformation sequences that are expected to provide high speed-ups are then evaluated on the actual architecture[5]. Hence, the regression models proposed in this chapter can be used as *proxies* of the evaluation function, which in the compiler optimisation problem corresponds to the compilation and execution of a program on the actual architecture.

For comparative purposes a similar evaluation to the one presented in chapter 7 is given

[5]Obviously, since we have exhaustive data, this evaluation is performed by looking at the table of complete speed-ups corresponding to each program.

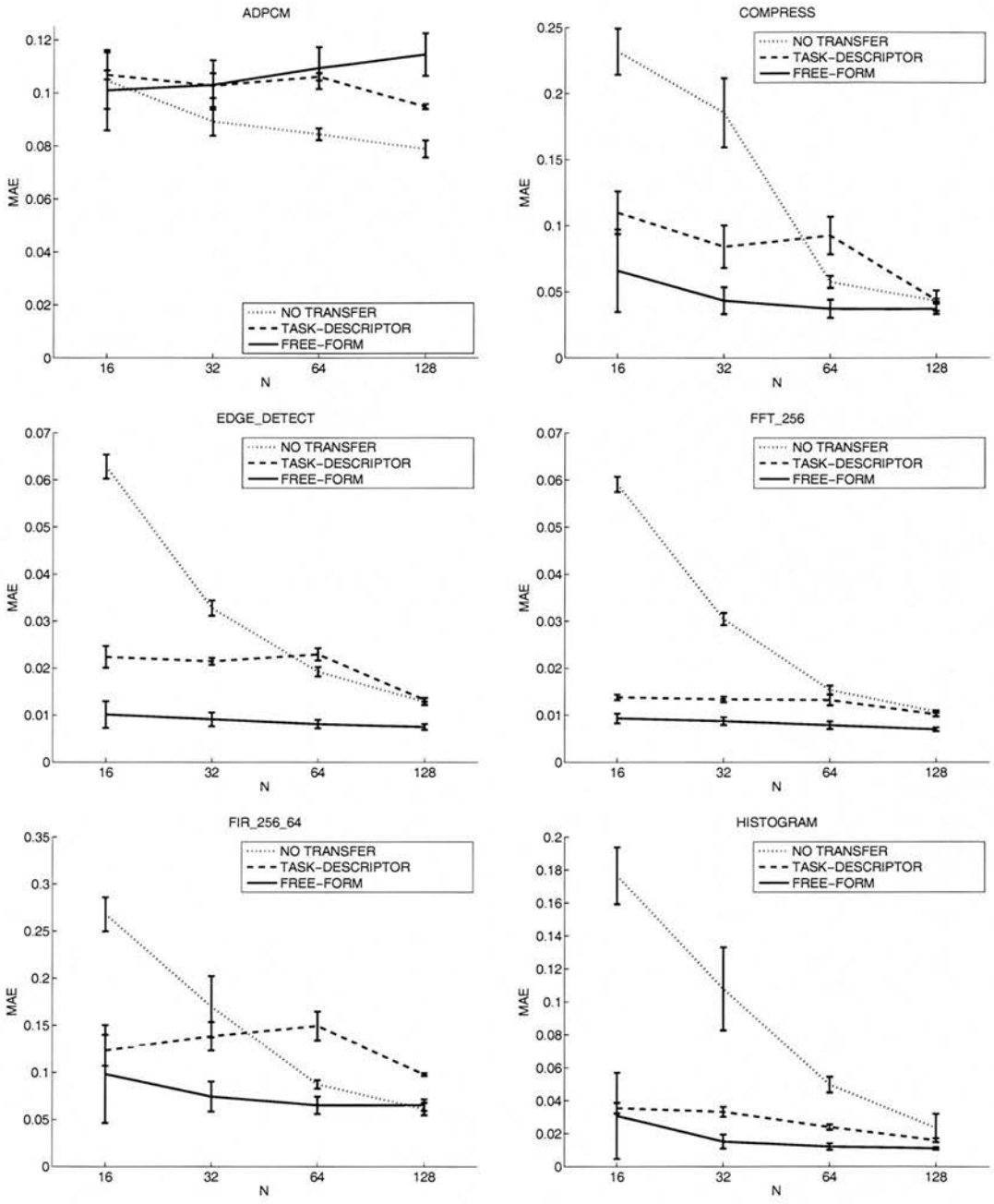

Figure 8.6: The performance of the transfer methods using multi-task GP with task-descriptor features (TASK-DESCRIPTOR); multi-task GP without task-descriptor features (FREE-FORM) and the no transfer method as a function of N on the small space of 6 problems (programs) of the SUIF data set on the TI board. The error bars denote one standard deviation.

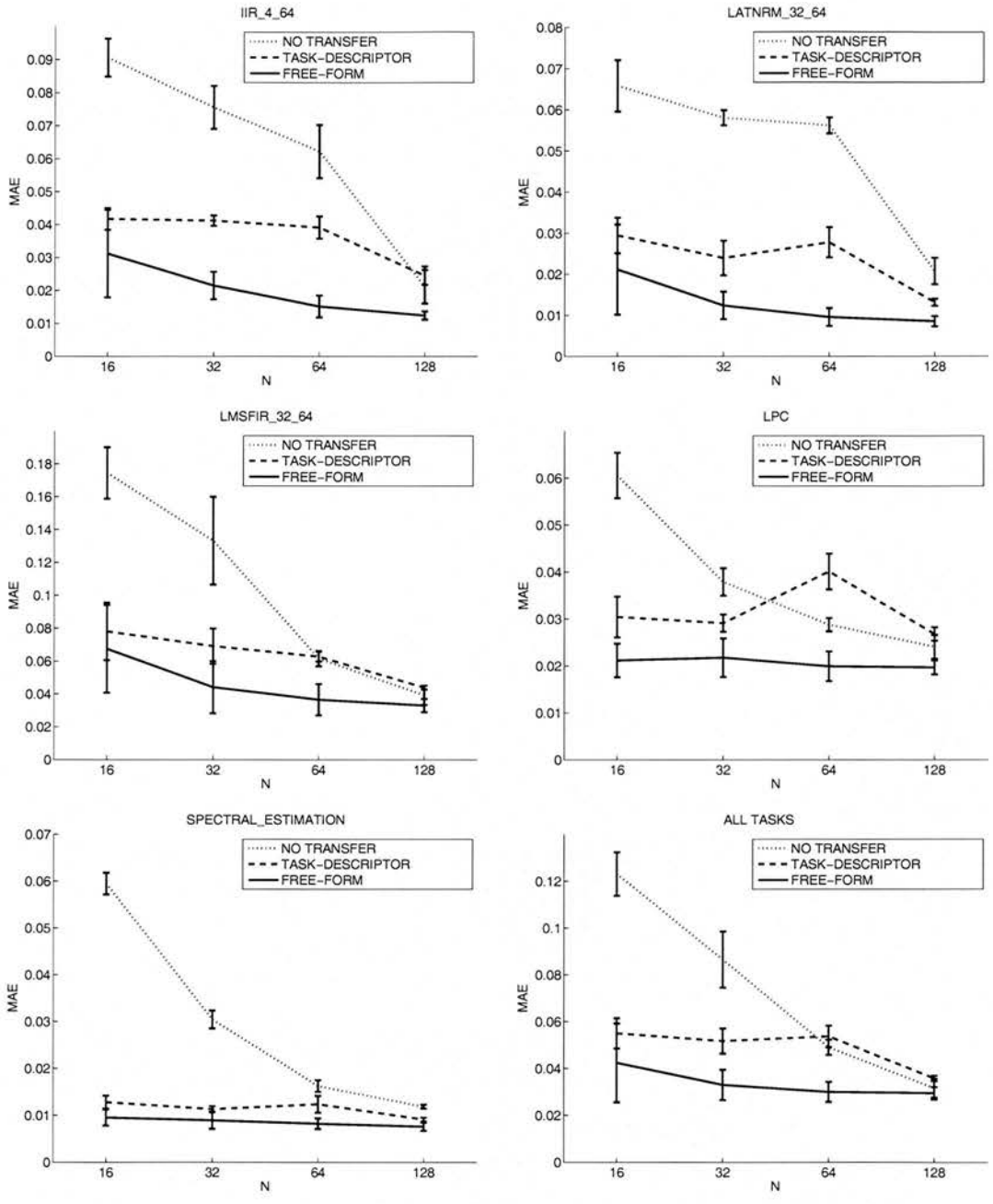

Figure 8.7: The performance of the transfer methods using multi-task GP with task-descriptor features (TASK-DESCRIPTOR); multi-task GP without task-descriptor features (FREE-FORM) and the no transfer method as a function of N on the small space of the remaining 5 problems (programs) of the SUIF data set on the TI board. The bottom right panel shows the average performances. The error bars denote one standard deviation.

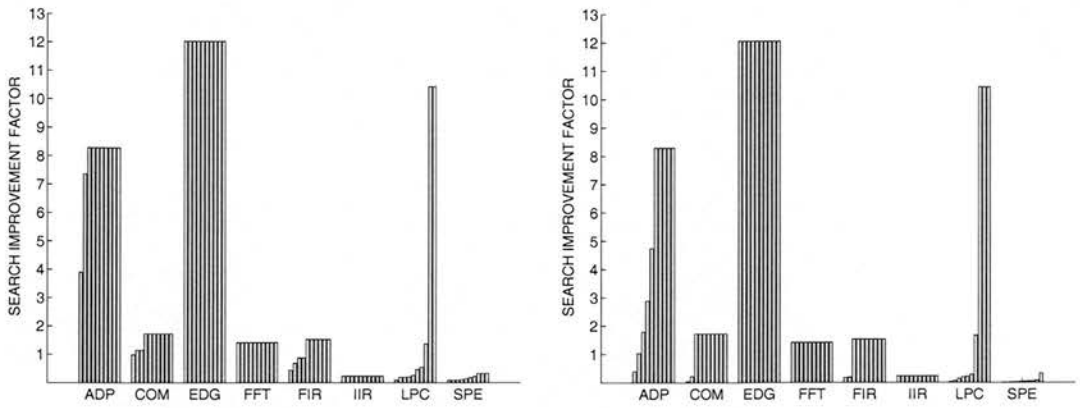

Figure 8.8: The performance of the multi-task GP methods when used for optimisation for 10 different replications on the small space of the SUIF data set on the TI board for those benchmarks for which some improvement can be achieved. On the LHS the task-descriptor method and on the RHS the free-form method. The number of test samples used for training in both methods is 8, which have been included for the computation of the $SIFs$.

here. Recall from section 7.5.6 that we defined $E_{\mathcal{U}}$ as the expected number of samples under uniform search necessary to achieve a performance value that is at most within $\varepsilon\%$ of the global maximum. Similary, $E_{\mathcal{A}}$ is the the expected number of samples under search algorithm $\mathcal{A}$ necessary to achieve a performance value that is at most within $\varepsilon\%$ of the global maximum. The search improvement factor is then defined as $SIF = E_{\mathcal{U}}/E_{\mathcal{A}}$, so that SIF values greater than 1 correspond to an improvement over uniform search.

Here we will compute $E_{\mathcal{A}}$ as the number of samples needed to achieve good performance (as defined above) empirically determined by evaluating those sequences that are expected to have the greatest speed-ups as predicted by the regression models. As in section 7.5.6 we use $\varepsilon = 5\%$.

Figure 8.8 shows the SIFs when using the task-descriptor method (left) and the free-form method (right) for those benchmarks where some improvement over the baseline version of the program is available. We see that unlike the performance prediction problem where the free-form method provided the best results, for the optimisation task both methods have roughly the same performance. We also see that for six out of eight benchmarks the methods give some improvement over uniform search with the best $SIFs$ obtained for *edge*, *lpc* and *adpcm*. Note however, that for those benchmarks where the methods degrade performance over uniform search (*iir* and *spectral*), the $E_{\mathcal{U}}$ is very low (1.8 and 2.6 respectively as shown in Table 7.3), so the methods still require a low number of samples to achieve good performance.

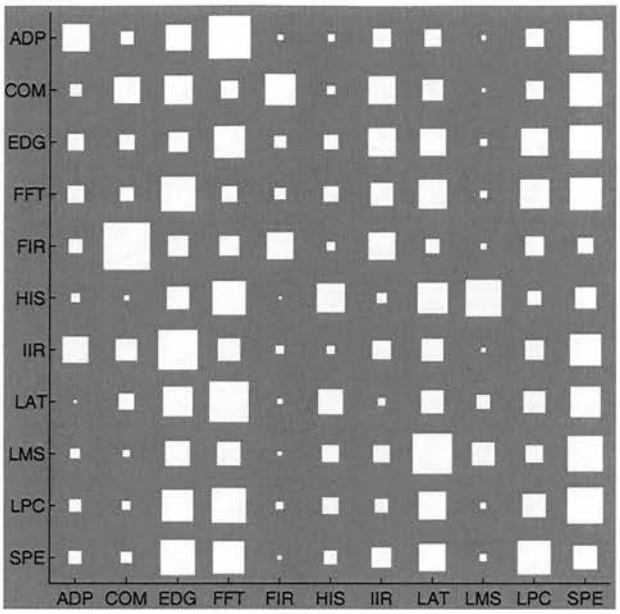

Figure 8.9: Hinton diagram indicating the $\bar{r}$ values for each task (program) on the small space of the SUIF data set on the TI board in order to illustrate inter-task transfer. The r values measure the relative contribution of the coefficients of a specific task with respect the contribution of all the tasks when making predictions. Each row corresponds to the $\bar{r}$ values for the test tasks labelled on the left. The order of the tasks is *adpcm*, *compress*, *edge*, *fft*, *fir*, *histogram*, *iir*, *latnrm*, *lmsfir*, *lpc* and *spectral*.

Analysing Inter-Program Similarity

It has been shown in section 8.6 that one way of quantifying inter-task transfer is to look at statistics of the r values, which are the coefficients of the training targets when making predictions. The r values measure the relative contribution of the coefficients of a specific task with respect the contribution of all the tasks when making predictions. Figure 8.9 shows the $\bar{r}$ values for each of the test tasks for the task-descriptor method with 256 training points per task and including $n_{te} = 8$ points from the test task. We see that the contribution from the test benchmarks is, in general, significant for making predictions, considering that only $n_{te} = 8$ test points have been used. However, predictions are also helped by the contribution of other tasks. For example, predictions on *fir* rely heavily on training data from benchmark *compress*. Similarly, predictions on all benchmarks except *histogram* are only weakly related to training data from benchmark *lmsfir*.

8.9 Summary and Discussion

This chapter has described an alternative machine learning approach to the problem of combinatorial optimisation. It contrasts with the standard sequential prediction formulation of the

problem in that it attempts to build an estimator of the evaluation function instead of a predictor of good points in the optimisation space. Such an approach is particularly convenient for those optimisation problems where the evaluation function is very costly. For example, in the compiler optimisation problem each function evaluation requires the compilation and execution of a program.

The main idea is that having an estimator for the evaluation function of an optimisation problem allows us to rapidly search the optimisation space without actually executing the evaluation function. Therefore, a performance predictor acts as a proxy for the evaluation function. In order to use such performance predictors in practice, one can evaluate a potentially large set of points in the optimisation space and then execute the actual evaluation function on those points that are expected to provide high performance. If the search space is very large so that it cannot be exhaustively enumerated, the proxy predictors can be used in conjunction with search (or sampling) methods.

One of the fundamental features of the regression methods proposed in this chapter is that they aim to achieve *transference* across tasks by exploiting the shared information between them. Indeed, multi-task Gaussian process prediction achieves this by directly modelling correlations between tasks. For the case when tasks-descriptor features are available, the combined method uses a joint covariance function over tasks-descriptor features and input features. When such a covariance function decomposes (as in the case of the square exponential covariance function), one effectively learns task similarity and input similarity as given by k^f and k^x. When task-descriptor features are not available or when they are difficult to define correctly, a "free-form" task similarity matrix K^f can be learnt. This can provide greater flexibility for modelling task relationships. Additionally, it has been shown that for the case of noise-free observations and block-design experiments a *cancellation of inter-transfer* occurs.

Multi-task Gaussian process prediction has been applied to the compiler performance prediction problem and has been shown to achieve better performance or roughly the same performance than the no transfer case. When using such predictors for optimisation, it has been shown that significant speed-ups can be achieved on the small space of the SUIF data set on the TI board.

It has also been shown that when using multi-task GP it is possible to quantify the amount of inter-task transfer that is taking place when making predictions. This has been applied to the compiler problem and some program similarities have been established.

As it has been emphasised in this chapter, following the indirect approach of learning performance predictors in order to tackle optimisation problems is best suited to those problems where the evaluation function is costly. Standard sequential prediction methods such as the technique of Predictive Search Distributions (PSD) should be preferred when this is not the case. However, if there are strong constraints on the number of samples that can be taken

from a new optimisation problem, performance predictors can be used as complementary techniques for sampling or search methods or even for PSD, when instead of actually executing the evaluation function the estimations given by the predictors are used.

Chapter 9

Conclusions and Future Work

This thesis has presented a machine learning approach to the problem of *compiler optimisation*. Compiler optimisation is the task of making a compiler produce *better* code, i.e. code that runs faster, occupies less memory or consumes less energy. The focus has been on optimising a compiler for speed, i.e. making a compiler generate code that runs faster. Although numerous program transformations for compiler optimisation have been proposed in the literature, it is difficult to ascertain when and how a particular code transformation should be applied to a program, as many of these transformations can be beneficial or detrimental depending (e.g.) on the code being compiled. Furthermore, some transformations can enable/disable or increase/decrease the applicability or effectiveness of other transformations, which is known as the problem of *interactions* between program transformations. Traditional approaches adopted by compiler writers use hand-crafted *heuristics* that dictate when and how a particular code transformation should be applied to a program. However, these heuristics require a lot of time and effort to construct and may sacrifice performance on programs they have not been tuned for.

The machine learning approach proposed in this thesis provides a solution to the problem of compiler optimisation that automatically generates optimisation strategies and that allows the generalisation of these strategies to programs that have not been seen before. In other words, it allows a compiler to tune itself in order to optimise programs while requiring very little human intervention. This approach is based upon the construction of models that capture the information that features of the programs provide about "good" sequences of code transformations and their performances.

Two different but related areas of research in compiler optimisation have been identified: *global optimisation* and *predictive modelling*. Unlike most previous work (see Chapter 4) that have addressed these problems independently, this thesis has presented a unified framework for compiler optimisation that uses predictive modelling in order to search the optimisation space of sequences of code transformations. This is achieved by exploiting *transference* across

programs based upon two different formulations of the problem: as a sequential prediction task and as a performance prediction problem. While the former (i.e. the sequential prediction task) is tackled with the Predictive Search Distributions technique (proposed in Chapter 7), the latter (i.e. the performance prediction problem) is addressed with multi-task Gaussian process prediction (see Chapter 8).

Both approaches, Predictive Search Distributions (PSD) and multi-task Gaussian process prediction, are formulated as general machine learning techniques. In particular, the PSD method is proposed in order to speed up search in combinatorial optimisation problems by learning a distribution over good solutions on a set of problem instances and using that distribution to search the optimisation space of a problem that has not been seen before. Likewise, multi-task Gaussian process prediction is proposed as a general method for multi-task learning that directly models the correlation between several machine learning tasks and thus it exploits the shared information across the tasks.

Unlike most previous approaches to compiler optimisation with machine learning that learn when/how to apply a single transformation in isolation or a fixed-order set of binary optimisation flags available within a compiler, the techniques proposed in this thesis are capable of dealing with the general problem of predicting good *sequences* of compiler transformations.

9.1 Contributions

The following are the specific contributions of this thesis:

1. A general framework for compiler optimisation based upon machine learning techniques has been proposed. This framework tackles the problems of global optimisation and predictive modelling in a unified manner by using a *transfer learning* approach. Thus, transference is exploited across different programs by learning predictive models on these programs in order to search the optimisation space of programs that have not been seen before, or programs for which very little data is available. Within this framework, a direct or an indirect approach can be adopted. In the direct approach, the problem is formulated as a sequential prediction task, i.e. predicting "good" transformation sequences. In the indirect approach, the optimisation task is formulated as a regression problem where proxy models of the performance of the programs under the application of compiler transformation sequences are constructed, which are then used in order to search the optimisation space of new programs or programs for which very little data is available (Chapter 3).

2. The direct approach to compiler optimisation (i.e. the sequential prediction task) has been addressed with the technique of Predictive Search Distributions (PSD), which has been proposed as a general method for speeding up search on combinatorial optimisation

problems. The main idea is to learn a distribution over good solutions on a collection of optimisation problems that can be characterised by a set of features and use this distribution to focus search on a problem that has not been seen before.

Thus, the method of Predictive Search Distributions (PSD) has been used to learn a distribution over "good" compiler transformation sequences across different programs, and this distribution has been utilised to focus the search of transformation sequences when a new program is presented. Significant improvements in performance have been achieved by this method on the SUIF data set (Chapter 7).

3. The indirect approach to compiler optimisation (i.e. the use of performance models for optimisation) has been formulated with the Multi-task Gaussian process prediction technique, which has been proposed as a general method for achieving *transference* across different machine learning tasks. The general idea is that of exploiting the shared information across the different tasks by directly modelling the correlations between them. This method can be used when task-features are available (multi-task GP with task-specific features) or when these features are unavailable or are difficult to define correctly (multi-task GP without task-specific features). An important characteristic of the technique is that observations on one task affect the predictions on the others.

   Thus, Multi-task GP has been used to exploit the shared information across different programs and their performances in order to predict the performance speed-up of a program when being applied a sequence of code transformations. This method has been shown, in general, to outperform the "no transfer" scenario (i.e. learning each performance prediction task on a single benchmark basis without using data from the other programs). Additionally, the predictions obtained with Multi-task GP have been used to search the optimisation spaces of the (small) SUIF data set, and significant speed-ups have been obtained (Chapter 8).

4. The problem of identifying and quantifying the main effects of program transformations and their interactions has been approached by extending the well-known statistical technique of analysis of variance (ANOVA) to deal with sequence data. Results have been reported on the small space of the SUIF data set (Chapter 6, sections 6.4 and 6.5).

5. An extensive review and characterisation of the related work on compiler optimisation with machine learning or/and artificial intelligence has been presented. This review has been focused on the problems of *global optimisation*, *predictive modelling*, *performance prediction* and optimisation space *characterisation* (Chapter 4).

9.2 Future Work

This thesis has presented a *probabilistic* machine learning approach to the problem of compiler optimisation. Although significant contributions have been made to the compiler optimisation area and to the machine learning field, there are several challenges yet to be addressed in the future. This section describes specific areas that can be tackled by future research in order to improve and extend the work presented in this thesis. In particular, section 9.2.1 proposes the analysis of the techniques described in this thesis on other benchmarks and transformation spaces. Section 9.2.2 discusses the advantages and disadvantages of optimising programs at finer levels of granularity such as functions or loops when using machine learning. Additionally, while sections 9.2.3 and 9.2.4 describe opportunities and challenges for learning across different input data, architectures and several objective functions, sections 9.2.5 and 9.2.6 explain future extensions to the techniques of predictive search distributions and multi-task Gaussian process prediction. Finally, section 9.2.7 proposes the use of search-tree methods for compiler optimisation and section 9.2.8 explains how the general problem of learning a mapping from an arbitrary program representation to a transformation sequence can be investigated with machine learning techniques.

9.2.1 Analysis of Other Benchmarks and Transformation Spaces

The methods proposed in section 6.4, Chapter 7 and Chapter 8 have been applied to the SUIF data set (described in section 5.1), which is based on the application of source-to-source transformations to 12 programs from the UTDSP benchmark suite (Lee, 1997). These methods can be readily used in order to investigate other benchmarks, transformation spaces and compiler infrastructures.

The MediaBench suite (Lee et al., 1997) and the MiBench suite (Guthaus et al., 2001) are examples of programs of interest to the embedded systems community. Other important transformations that were not included in the experimental set up of this thesis are, for example, *loop tiling* and *loop fusion*. Thus, future work can investigate the effect of a richer set of program transformations on a larger set of benchmarks by using the methods proposed in this thesis.

Going beyond program transformations applied at the source level, other environments (e.g. compilers and architectures) can be considered. For example, new experiments can be executed on the PathScale compiler (PathScale, 2005) for C, C++ and Fortran programs, and/or on the Jikes framework (Burke et al., 1999) for Java programs. This will help to demonstrate the applicability and portability of the different approaches proposed in this thesis for learning in compiler optimisation. Additionally, it will provide a direct understanding of the benefits of the transformations, which is made difficult by the application of source-level transformations

on top of the optimisations applied by the base compiler.

9.2.2 Optimisation at Finer Levels of Granularity

The approach to compiler optimisation adopted in this thesis has relied upon the application of transformation sequences at a program level. It has been shown that, when using the techniques proposed in this thesis, this approach has achieved significant improvements in the performance of the programs used. However, as pointed out in section 5.3, transformations applied at a program level may lead to inferior performance compared to the case when these transformations are applied at finer levels of granularity such as functions or loops. Indeed, different transformation sequences can be required for distinct sections of a program and optimisation opportunities may be missed when assuming a single best transformation sequence for a complete program. The advantages of a *local* optimisation approach are the following:

- The use of transformations applied at finer levels of granularity represents a more flexible approach to compiler optimisation since different parts of a program may benefit from different optimisation sequences.

- Applying transformations at finer levels of granularity, such as function-level or loop-level, can provide a more efficient way of obtaining training data to be used by machine learning techniques. Indeed, by focusing on learning at finer levels of granularity and ignoring the effect of interactions between different code segments, it is possible to generate several training data points per program.

However, there are at least two obstacles for this approach to be successful. Firstly, the *instrumentation* of the code (i.e. the introduction of external code in order to measure execution times) at finer levels of granularity is more invasive than the instrumentation at a program level and inevitably the former yields noisier measurements. Therefore, a very cautious pre-processing stage should be followed when including the data that will be used for learning.

Secondly, for this approach to be feasible, it will be necessary to assume that *the effect of interactions* between optimisations applied to different parts of the code is negligible. However, independent optimisation of code segments such as functions or loops may not lead to an improvement in performance of the complete program.

Thus, immediate future work on investigating the effect of local-level optimisation compared to program-level optimisation can be focused on addressing the following questions:

- Do local transformations provide better speed-ups?: experiments that allow a systematic comparison of local optimisation strategies versus global optimisation strategies must be carried out in order to analyse if indeed locally-applied transformations provide further benefits over globally-applied transformations.

- Do code segments within a program interact?: as explained above, optimising (local) parts of the code independently may not lead to (global) optimisation of the complete program. Here we are interested in evaluating if the effect of optimisations on different code segments within a program is additive. If this is not the case, quantifying the effect of these interactions will be of great relevance to the compiler community. Readily available techniques such as ANOVA can prove useful for this purpose.

Having investigated the issues above, machine learning can be used in order to predict good compiler transformations for code segments such as functions or loops.

9.2.3 Learning across Different Input Data and Architectures

This thesis has addressed the problem of compiler optimisation by learning "good" transformation sequences or the effect of transformation sequences across different benchmarks when their corresponding input data has been fixed. However, in practice, the behaviour of a program may change when different input data is presented, for example when having different matrix sizes in a matrix multiplication algorithm.

Although there is some empirical evidence that a fixed set of compiler optimisations may be a good compromise across different data sets (see e.g. Fursin et al., 2007), in general, it will be necessary to learn a different optimisation strategy for distinct input data. Thus, future work must involve the analysis of the effect of input data on program behaviour and the evaluation of techniques that account for different input data sets when learning an optimising compiler.

A simple approach to learning across different input data is to characterise this input by a set of features. For example, Vuduc et al. (2004) formulate the problem of selecting the best implementation of a matrix multiplication algorithm for a given input as a classification task where the input is characterised by a vector containing the matrices' sizes.

Alternatively, the problem of learning across different input data can be tackled by considering programs with different inputs as different instances. In this case, it would be necessary to extract dynamic features that characterise the run-time behaviour of a program (see e.g. Cavazos et al., 2007, as a recent reference). A potential disadvantage of this approach is that collecting such dynamic information would require at least one execution of the test program.

In addition to learning across input data, it will be interesting to investigate if it is possible to *transfer* the knowledge on good compiler transformation sequences across different architectures. As in the case of *transference* across programs, learning optimisation strategies across architectures requires the extraction of informative (architecture) features. Nonetheless, even if these features are difficult to define correctly, it is possible to use techniques such as the ones described in Chapter 8 as long as some samples can be drawn from the test architecture and test program.

9.2.4 Multi-Objective Optimisation

This thesis has focused on the problem of optimising a compiler in order to make programs run *faster*. However, in practical scenarios such as embedded architectures it is important to consider other objectives besides execution time. For example, minimising the size of a program and reducing the power consumed during its execution are important goals to be achieved by an optimising compiler that generates code for an embedded system.

Nevertheless, the machine learning techniques proposed in this thesis are not restricted to optimising a program for execution time. Indeed, if measurements of the execution time, code size and power consumption are available, the application of the techniques proposed in this thesis is straightforward as the only modification needed is the formulation of an objective function that considers a trade-off between these three different objectives. However, the general problem of optimising such a potentially complex objective function can be very hard.

9.2.5 Predictive Search Distributions

Given the limited amount of training data available for the application of predictive search distributions (PSD), we have used in this thesis very simple distributions in order to model "good" transformation sequences (i.e. an iid distribution and a Markov chain distribution) and a very simple method to learn PSD (i.e. nearest neighbours). As described in section 7.3.2, more complex distributions can be used and learning methods based on the maximisation of the conditional likelihood can be adopted. Thus, future work can investigate more complex parametrised distributions if enough data is available. Suitable choices of distributions can be, for example, hidden Markov models (HMMs) and conditional random fields (CRFs, Lafferty et al., 2001).

From a machine learning perspective, the PSD technique can be seen as a general method to tackle combinatorial optimisation problems. It will be interesting to investigate other domains where this technique can be applied. As mentioned in section 7.6, families of optimisation tasks can be induced, for example, by varying the edge weights in the input graph of problems such as the ground state of a spin glass (Pelikan and Goldberg, 2001), or the minimum balanced cut graph partitioning problem (Andreev and Räcke, 2004).

9.2.6 Multi-task Gaussian Process Performance Prediction

Building regression models to predict the performance of programs under the effect of compiler optimisations is a difficult task. Indeed, although the methods presented in Chapter 8 achieved good results compared to other approaches such as linear regression, median predictors and single-task learning, there is still room for improving the accuracy of the predictors[1].

[1]Note, however, that the ultimate goal of finding transformation sequences that provide significant speed-ups can be achieved without requiring very accurate models.

Several directions can be considered in the future in order to enhance the effectiveness of these regression models. For example, the investigation of more suitable program features for predicting speed-up performances, the use of active data selection and the choice or design of more elaborate covariance functions. These directions for future research are briefly described below.

Program Features

Unquestionably, one of the most important directions for future work on building regression models for predicting performance speed-ups is the investigation of suitable features that characterise the behaviour of programs. Note that this is also relevant to Predictive Search Distributions but its effect is seen more directly in performance models if they are assessed based on the accuracy of their predictions.

Reasonably accurate regression models based on code features can be built on a single benchmark scenario, i.e. without considering transference across benchmarks. However, this scenario is not very interesting as it requires a large number of evaluations for a new program. In a multiple-benchmark scenario (or multi-task learning scenario) the use of code features for regression has been shown to underperform other approaches that are based on dynamic information such as the *canonical responses*.

This motivates further research on the extraction of code features and on how they can be used in order to facilitate transference across programs. It is important to emphasise that the representation of a program is not restricted to a flat vector of features but structured representations such as trees can also be considered. As we shall see in section 9.2.8, learning with structured data is also a very interesting area for future research.

As shown in Chapter 8 and in Cavazos et al. (2007), dynamic information such as the canonical responses or performance counter information proves useful when predicting performance speed-ups or good compiler transformations for programs. Thus, future work can include the study of the informativeness of such descriptors and should consider models that are based on both *static and dynamic information*.

Active Learning

An alternative way of improving the accuracy of the performance predictors is the use of smarter techniques for the selection of the samples on which the regression models are constructed. Indeed, a careful selection of the transformation sequences to be included in the training data can provide further benefits over the case when the data-points are selected at random.

The problem of sequentially selecting these data-points in order to improve the performance of the models can be seen as an active data selection (or experimental design) problem.

Useful references for approaching this problem are for example MacKay (1992) and Cohn et al. (1995).

Covariance Functions

A rather different way to improve the performance of the regression models is to consider a careful selection/design of a covariance function for Gaussian processes. Several examples of commonly used covariance functions for Gaussian processes can be found in Rasmussen and Williams (2006, Chapter 4). However, there is not much work in the literature regarding suitable covariance functions for heterogeneous input spaces as the one investigated in Chapter 8 that considers a program-dependent representation and a transformation-dependent representation.

Finally, as we shall see in the next section, it is possible to formulate the problem of finding good optimisation sequences with a search-tree approach, where each program version corresponds to a node (in a search tree), which is obtained by applying a compiler transformation. An interesting application of the regression models proposed in Chapter 8 is the use of these models in order to estimate the *value* function and the *reward* function in this approach.

9.2.7 A Search-tree Formulation of Compiler Optimisation

As described in the MSc thesis of Gupta (2007, pages 7–8 and 35–36)[2], the compiler optimisation problem can be formulated using a search-tree approach. Here each node of the tree is represented with a set of features $\mathbf{t}$ that describe a version of the program, with the root node being the baseline program. The children of a node are obtained by applying a compiler transformation and each node has associated with it a *reward* $r(\mathbf{t})$ value that can be, for example, the speed-up of the corresponding transformed program. The goal is to find the node with maximum reward in the tree and the transitions (i.e. the sequence of transformations) that lead to that node. The key issue is that the reward value is known only at a limited number of nodes (given the very large number of possible states) and, therefore, a performance model (or proxy) is needed to approximate the reward value at the unknown locations. In order to find the node with maximum reward in the tree the *value function* $V(\mathbf{t})$ is defined as the optimal reward in the sub tree rooted at node $\mathbf{t}$ and is computed using the Bellman's equation:

$$V(\mathbf{t}) = \max(r(\mathbf{t}), \max_{\mathbf{t}'} V(\mathbf{t}')), \tag{9.1}$$

where $\mathbf{t}'$ is a successor of $\mathbf{t}$, $r(\mathbf{t})$ is the reward function and $V(\mathbf{t})$ is the value function. Note the similarity of this approach with reinforcement learning (RL), see for example Kaelbling et al. (1996) for a concise survey of RL methods, and Sutton and Barto (1998) for a more comprehensive description. The main difference here is that we are interested in finding the

[2]Supervised by Chris Williams with input from Peter Dayan.

maximum reward in the search tree rather than in maximising the expected reward in the long run. In order to achieve this, Gupta (2007) proposes an algorithm that approximates the reward function and the value function with linear-in-the-parameters models. Although it is shown that such a formulation places several difficulties in search problems (see Gupta, 2007, pages 60–65), it will be interesting to investigate the results in real compiler data as the one used in this thesis. Furthermore, as emphasised throughout this thesis, a very interesting goal is that of achieving *transference* across programs. Thus, search-tree methods based on multi-task Gaussian process predictors (Chapter 8) can also be investigated.

9.2.8 Learning in Structured Spaces

Predicting good compiler transformations for programs can be seen as the problem of learning in structured spaces where the goal is to model the mapping $\mathbf{t} \rightarrow \mathbf{x}$, where $\mathbf{t}$ is a representation of a program and $\mathbf{x}$ is a transformation sequence that is expected to achieve best performance. Note that in general, for any given program $\mathbf{t}$, $\mathbf{x}$ can have an arbitrary length ℓ. Additionally, $\mathbf{t}$ is not necessarily a vector of features but it can be a more general representation of a program as given by, for example, an Abstract Syntax Tree (AST) or a Control Flow Graph (CFG).

The structured prediction problem can be addressed with Predictive Search Distributions (Chapter 7) by considering complex parametrised distributions, e.g. second order Markov models, Hidden Markov Models (HMMs) or Conditional Random Fields (CRFs, Lafferty et al., 2001). If practical constraints are placed on the number of samples that can be drawn from such distributions, it is possible to make single-point predictions by considering the argmax of the predicted distributions. In other words, given a new program $\mathbf{t}^*$ we predict $\mathbf{x}^*$ such $\mathbf{x}^* = \text{argmax}_{\mathbf{x} \in \mathcal{X}} p(\mathbf{x}|\mathbf{t}^*)$. Additionally, it is possible to deal with variable-length sequences by learning a length-conditional model on good solutions $p(\mathbf{x}|\mathbf{t}, \ell)$ and a posterior over lengths on these solutions $p(\ell|\mathbf{t})$.

Multi-task Gaussian process prediction (Chapter 8) can exploit the structured nature of the problem by using (structured) covariance functions on both the transformation sequence space and the program representation space. There has been a lot of work in recent years on kernels for non-vectorial data as described in Rasmussen and Williams (2006, section 4.4), Schölkopf and Smola (2001, Chapter 13) and Shawe-Taylor and Cristianini (2004, Chapter 11). However, the regression approach to compiler optimisation has the disadvantage that it can be very slow if the transformation-sequence space is very large as the learned regression function would need to scan as many sequences as possible on a test program.

Alternatively, the recent advances in kernels for structured data can be used by directly considering the problem of learning the general mapping from a structured representation of a program to a transformation sequence. One possible approach to learning with structured data is **Kernel Dependency Estimation** (Weston et al., 2003). Here the authors consider the

general problem of finding a mapping between a general class of objects to another (possibly different) class of objects. To achieve this, inputs and outputs are mapped to kernel spaces by using (generally) different appropriate kernel functions $\Phi_k(\mathbf{t})$ and $\Phi_m(\mathbf{x})$ respectively. Their solution is based on the following steps:

1. Decompose $\Phi_m(\mathbf{x})$ into p orthogonal directions by using kernel PCA (see e.g. Schölkopf and Smola, 2001, Chapter 14).

2. Learn the mapping from $\Phi_k(\mathbf{t})$ to each of the above directions independently using e.g. standard kernel regression methods.

3. Solve the pre-image problem, i.e. map the solution found in the $\mathcal{M}$ space to the original $\mathcal{X}$ space.

The final step above is, in general, very hard and it may be necessary to search the space of candidate solutions instead. Additionally, the solution above is strongly dependent upon having "good" kernel functions. The art of designing appropriate kernel functions for structured objects is similar the the task of designing suitable covariance functions in Gaussian processes. In the case of the compiler optimisation problem we are mainly interested in finding good string representations for sequences of transformations and tree-based representations for programs. Therefore, string kernels and tree kernels (see e.g. Lodhi et al., 2002; Vishwanathan and Smola, 2003) can be readily applied.

In summary, this thesis has extensively reviewed the previous literature on compiler optimisation with machine learning and has proposed novel methods on the areas of performance prediction, predictive modelling and characterisation of the optimisation space that are significant contributions not only to the compiler optimisation field but also to machine learning research. Additionally, based upon the research carried out during the development of this thesis, directions for future work have been proposed not only to improve what has been achieved in this thesis but also to pursue the ultimate goal of constructing a machine-learning-based adaptive optimising compiler.

Appendix A

Transformations Used on the Large Space of the SUIF Data Set

Id	Transformation
1	Aggressively scalarise constant array references
2	Annotate array forms
3	Array Delinearisation
4–6	Array Padding with parameters 0, 2, 4
7	Array Scalarisation
8	Bit Packing
9	Bounds Comparison Substitution
10	Break load constant instructions
11	Chain multiple array references
12	Common Subexpression Elimination
13	Common Subexpression Elimination (no pointers)
14	Constant Folding
15	Constant Propagation
16	Control Simplification
17	Copy Propagation
18	Dead Code Elimination
19	Dismantle abs instructions
20	Dismantle array instructions
21	Dismantle composite float and integer instructions
22	Dismantle composite float instructions
23	Dismantle divceil instructions
24	Dismantle divfloor instructions

Id	Transformation
25	Dismantle divmod instructions
26	Dismantle empty TREE_FORs
27	Dismantle integer abs instructions
28	Dismantle integer max instructions
29	Dismantle integer min instructions
30	Dismantle max instructions
31	Dismantle memcpy instructions
32	Dismantle min instructions
33	Dismantle multi-way branches
34	Dismantle non-constant FORs
35	Dismantle TREE_BLOCKs
36	Dismantle TREE_BLOCKs with empty symbol table
37	Dismantle TREE_FORs
38	Dismantle TREE_FORs with modified index variable
39	Dismantle TREE_FORs with spilled index variable
40	Dismantle TREE_LOOPs
41	Eliminate enumeration types
42	Eliminate struct copies
43	Eliminate sub-variables
44	Elimination of unused symbols
45	Elimination of unused types
46–50	Expression Tree Breakup with parameters 0, 1, 2, 3, 4
51	Extract array upper bounds
52	Find FOR loops
53	Fix address taken
54	Fix bad nodes
55	FOR Loop Normalisation
56	Form Arrays
57	Forward Propagation
58	Global variable privatisation
59	Globalise local static variables
60	Guard FORs
61	Hoisting of loop invariants
62	IF Hoisting
63	Improve array bound information
64	Induction Variable Detection

Id	Transformation
65	Kill redundant line marks
66	Lift call expressions
67	Loop flattening
68–71	Loop Unrolling with parameters 1, 2, 3, 4
72	Mark constant variables
73	MOD/REF Annotations
74	Move loop-invariant conditionals
75	Privatisation
76	Put in explicit load/stores for non-local variables
77	Reassociation
78	Reconstruct explicit array references
79	Reduction Detection
80	Replace call-by-reference
81	Replace constant variables
82	Scalarisation
83	Scalarise constant array references
84–87	Splitting of deep FOR loops with parameters 1, 2, 3, 4
88	Strictly fix bad nodes
89	Turn imperfectly nested loops into perfectly nested loops
90	Unstructured control flow optimisation

Table A.1: Transformations used on the large space of the SUIF data set.

Bibliography

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

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition.

Akaike, H. (1974). A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723.

Aldous, D. J. (1985). Exchangeability and related topics. In *École d'été de probabilités de Saint-Flour, XIII—1983*, volume 1117 of *Lecture Notes in Mathematics*, pages 1–198. Springer, Berlin.

Almagor, L., Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S. W., Subramanian, D., Torczon, L., and Waterman, T. (2003). Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms. Technical report, Los Alamos Computer Science Institute.

Almagor, L., Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S. W., Subramanian, D., Torczon, L., and Waterman, T. (2004). Finding effective compilation sequences. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 231–239, Washington, DC, USA. ACM Press.

Andreev, K. and Räcke, H. (2004). Balanced graph partitioning. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 120–124, New York, NY, USA. ACM Press.

Bacon, D. F., Graham, S. L., and Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420.

Bakker, B. and Heskes, T. (2003). Task clustering and gating for Bayesian multitask learning. *Journal of Machine Learning Research*, 4:83–99.

Baluja, S. (1994). Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, USA.

Baluja, S. and Caruana, R. (1995). Removing the genetics from the standard genetic algorithm. In *Proceedings of the 12th International Conference on Machine Learning*, pages 38–46.

Baluja, S. and Davies, S. (1997). Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. In *Proceedings of the 14th International Conference on Machine Learning*, pages 30–38. Morgan Kaufmann.

Baluja, S. and Davies, S. (1998). Fast probabilistic modeling for combinatorial optimization. In *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 469–476, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Baxter, J. (2000). A model of inductive bias learning. *Journal of Artificial Intelligence Research*, 12:149–198.

Benitez, M. E. and Davidson, J. W. (1988). A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, New York, NY, USA. ACM Press.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Bodin, F., Mével, Y., and Quiniou, R. (1998). A user level program transformation tool. In *Proceedings of the 12th International Conference on Supercomputing*, pages 180–187, New York, NY, USA. ACM Press.

Bonilla, E. V. (Oct. 2004). Predicting good compiler transformations using machine learning. Master's thesis, School of Informatics, University of Edinburgh, UK.

Bonilla, E. V., Agakov, F. V., and Williams, C. K. I. (2007). Kernel multi-task learning using task-specific features. In *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics*. Omnipress.

Bonilla, E. V., Chai, K. M. A., and Williams, C. K. I. (2008). Multi-task Gaussian process prediction. In Platt, J., Koller, D., Singer, Y., and Roweis, S., editors, *To appear in Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA.

Bonilla, E. V., Williams, C. K. I., Agakov, F. V., Cavazos, J., Thomson, J., and O'Boyle, M. F. P. (2006). Predictive search distributions. In Cohen, W. W. and Moore, A., editors, *Proceedings of the 23rd International Conference on Machine learning*, pages 121–128, New York, NY, USA. ACM.

Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., Srinivasan, H., and Whaley, J. (1999). The jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM Conference on Java Grande*, pages 129–141, New York, NY, USA. ACM Press.

Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., and Zorn, B. (1997). Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222.

Caruana, R. (1997). Multitask learning. *Machine Learning*, 28(1):41–75.

Cavazos, J. (2005). *Automatically constructing compiler optimization heuristics using supervised learning*. PhD thesis, Department of Computer Science, University of Massachusetts Amherst, USA.

Cavazos, J., Dubach, C., Agakov, F., Bonilla, E., O'Boyle, M. F. P., Fursin, G., and Temam, O. (2006). Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 24–34, New York, NY, USA. ACM.

Cavazos, J., Eliot, J., and Moss, B. (2004). Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 183–194, New York, NY, USA. ACM Press.

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., and Temam, O. (2007). Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA. IEEE Computer Society.

Cavazos, J. and O'Boyle, M. F. P. (2006). Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object oriented Programming Systems, Languages, and Applications*, pages 229–240, New York, NY, USA. ACM Press.

Chow, C. and Liu, C. (1968). Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467.

Chow, K. and Wu, Y. (1999). Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*.

Cohn, D. A., Ghahramani, Z., and Jordan, M. I. (1995). Active learning with statistical models. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems 7*, pages 705–712. The MIT Press.

Cooper, K. D., Schielke, P. J., and Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms. *SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 34(7):1–9.

Cooper, K. D., Subramanian, D., and Torczon, L. (2002). Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22.

Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Morgan Kaufmann.

Cressie, N. A. C. (1993). *Statistics for Spatial Data*. Wiley, New York.

de Bonet, J., Isbell, C., and Viola, P. (1997). MIMIC: Finding optima by estimating probability densities. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*. The MIT Press.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38.

Dongarra, J. and Hinds, A. (1979). Unrolling loops in FORTRAN. *Software: Practice and Experience*, 9:219–226.

Eeckhout, L., Sundareswara, R., Yi, J. J., Lilja, D. J., and Schrater, P. (2005a). Accurate statistical approaches for generating representative workload compositions. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 56–66, Austin, Texas, USA. IEEE.

Eeckhout, L., Sundareswara, R., Yi, J. J., Lilja, D. J., and Schrater, P. (2005b). Correction to "Accurate statistical approaches for generating representative workload compositions". `http://www.elis.ugent.be/~leeckhou/papers/correction.pdf`.

Eeckhout, L., Vandierendonck, H., and Bosschere, K. D. (2002). Workload design: Selecting representative program-input pairs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 83–94, Washington, DC, USA. IEEE Computer Society.

Evgeniou, T., Micchelli, C. A., and Pontil, M. (2005). Learning multiple tasks with kernel methods. *Journal of Machine Learning Research*, 6:615–537.

Franke, B. and O'Boyle, M. (2001). An empirical evaluation of high level transformations for embedded processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 59–66, New York, NY, USA. ACM Press.

Franke, B., O'Boyle, M., Thomson, J., and Fursin, G. (2005). Probabilistic source-level optimisation of embedded programs. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 78–86, New York, NY, USA. ACM Press.

Fursin, G., Cavazos, J., O'Boyle, M. F., and Temam, O. (2007). MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers*, Ghent, Belgium.

Fursin, G., Cohen, A., O'Boyle, M., and Temam, O. (2005). A practical method for quickly evaluating program optimizations. In *High Performance Embedded Architectures and Com pilers*, volume 3793 of *Lecture Notes in Computer Science*, pages 29–46. Springer, Berlin.

Fursin, G. G. (2004). *Iterative Compilation and Performance Prediction for Numerical Applications*. PhD thesis, School of Informatics, University of Edinburgh, UK.

Fursin, G. G., O'Boyle, M., and Knijnenburg, P. (2002). Evaluating iterative compilation. In *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computers*, pages 305–315.

Goldstein, H. (2003). *Multilevel Statistical Models*. Hodder Arnold.

Gupta, A. (2007). Approximating search using reinforcement learning methods. Master's thesis, School of Informatics, University of Edinburgh, UK.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual Workshop on Workload Characterization*, pages 3–14, Washington, DC, USA. IEEE Computer Society.

Hall, M. W., Anderson, J.-A. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., and Lam, M. S. (1996). Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89.

Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. (2005a). Automatic selection of compiler options using non-parametric inferential statistics. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA. IEEE Computer Society.

Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. (2005b). Optimizing general purpose compiler optimization. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 180–188, New York, NY, USA. ACM Press.

Heckerman, D., Geiger, D., and Chickering, D. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243.

Hedayat, A., Sloane, N. J. A., and Stufken, J. (1999). *Orthogonal Arrays: Theory and Applications*. Springer Series in Statistics. Springer, first edition.

Hoerl, A. E. and Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67.

Horvitz, E., Ruan, Y., Gomes, C., Kautz, H., Selman, B., and Chickering, D. M. (2001). A Bayesian approach to tackling hard computational problems. In *Proceedings of the 17th Annual Conference on Uncertainty in Artificial Intelligence*, pages 235–244, San Francisco, CA. Morgan Kaufmann.

İpek, E., McKee, S. A., de Supinski, B. R., Schulz, M., and Caruana, R. (2006). Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87.

Jeffreys, H. (1966). *Theory of Probability*. Oxford University Press.

Jiménez, D. A. (2005). Piecewise linear branch prediction. In *Proceedings of the 32nd Annual International Conference on Computer Architecture*, pages 382–393.

Johnson, A. and Shapiro, J. L. (2001). The importance of selection mechanisms in distribution estimation algorithms. In *Proceedings of the 5th International Conference on Artificial Evolution*, pages 91–103, London, UK. Springer-Verlag.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

Karkhanis, T. S. and Smith, J. E. (2004). A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 338–349, Washington, DC, USA. IEEE Computer Society.

Kass, R. E. and Raftery, A. E. (1995). Bayes factors. *Journal of the American Statistical Association*, 90(430):773 – 795.

Kautz, H. A., Horvitz, E., Ruan, Y., Gomes, C. P., and Selman, B. (2002). Dynamic restart policies. In *Proceedings of the 18th National Conference on Artificial Intelligence and 14th*

Conference on Innovative Applications of Artificial Intelligence, pages 674–681, Edmonton, Alberta, Canada. AAAI Press.

Kelly, W. and Pugh, W. (1993). A framework for unifying reordering transformations. Technical Report CS-TR-3193, University of Maryland.

Kisuki, T., Knijnenburg, P. M. W., and O'Boyle, M. F. P. (2000). Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA. IEEE Computer Society.

Knijnenburg, P. M. W., Kisuki, T., and O'Boyle, M. F. P. (2002). Iterative compilation. In Deprettere, F., Teich, J., and Vassiliadis, S., editors, *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation*, number 2268 in Lecture Notes in Computer Science, pages 171–187. Springer-Verlag New York, Inc.

Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., and Gallivan, K. (2003). Finding effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tools for Embedded Systems*, pages 12–23, New York, NY, USA. ACM Press.

Kulkarni, P. A., Whalley, D. B., Tyson, G. S., and Davidson, J. W. (2006). Exhaustive optimization phase order space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 306–318, Washington, DC, USA. IEEE Computer Society.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning*, pages 282–289. Morgan Kaufmann, San Francisco, CA.

Langley, P. (1996). *Elements of Machine Learning*. Morgan Kaufmann, San Francisco, USA.

Larrañaga, P. and Lozano, J. A. (2001). *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, Norwell, MA, USA.

Lawrence, N. D. and Platt, J. C. (2004). Learning to learn with the informative vector machine. In *Proceedings of the 21st International Conference on Machine Learning*, New York, NY, USA. ACM.

Lee, C. (1997). UTDSP benchmark suite. `http://www.eecg.toronto.edu/~corinna/`.

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. (1997). MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the International Symposium on Microarchitecture*, pages 330–335.

Lee, H., von Dincklage, D., Diwan, A., and Moss, J. E. B. (2004). Understanding the behavior of compiler optimizations. Technical Report CU-CS-972-04, University of Colorado at Boulder.

Lee, H., von Dincklage, D., Diwan, A., and Moss, J. E. B. (2006). Understanding the behavior of compiler optimizations. *Software: Practice and Experience*, 36(8):835–844.

Lindgren, B. W. (1993). *Statistical Theory*. Chapman & Hall, fourth edition.

Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., and Watkins, C. J. C. H. (2002). Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444.

Long, S. and O'Boyle, M. (2004). Adaptive Java optimisation using instance-based learning. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 237 246, New York, NY, USA. ACM Press.

MacKay, D. J. C. (1992). Information-based objective functions for active data selection. *Neural Computation*, 4(4):590–604.

MacKay, D. J. C. (1998). Introduction to Gaussian processes. In Bishop, C. M., editor, *Neural Networks and Machine Learning*, NATO ASI Series, pages 133–166. Kluwer.

MacKay, D. J. C. (2003). *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press.

McCabe, G. P. (1984). Principal variables. *Technometrics*, 26(2):137–144.

Minka, T. P. and Picard, R. W. (1999). Learning how to learn is learning with point sets. `http://research.microsoft.com/~minka/papers/point-sets.html`.

Monsifrot, A. and Bodin, F. (2001). Computer aided hand tuning (CAHT): "applying case-based reasoning to performance tuning". In *Proceedings of the 15th International Conference on Supercomputing*, pages 196–203, New York, NY, USA. ACM Press.

Monsifrot, A., Bodin, F., and Quiniou, R. (2002). A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50. Springer-Verlag.

Montgomery, D. C. (1997). *Design and Analysis of Experiments*. John Wiley and Sons, forth edition.

Moss, E., Utgoff, P., Cavazos, J., Brodley, C., Scheeff, D., Precup, D., and Stefanović, D. (1998). Learning to schedule straight-line code. In *Advances in Neural Information Processing Systems 10*, pages 929–935, Cambridge, MA, USA. MIT Press.

Mühlenbein, H. (1997). The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346.

Mühlenbein, H. and Mahnig, T. (1999). The factorized distribution algorithm for additively decompressed functions. In *Proceedings of the Congress on Evolutionary Computation*, pages 752–759, Piscataway, NJ. IEEE Service Center.

Mühlenbein, H. and Paass, G. (1996). From recombination of genes to the estimation of distributions I. Binary parameters. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, pages 178–187, London, UK. Springer-Verlag.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Number 118 in Lecture Notes in Statistics. Springer, New York.

Neter, J., Kutner, M. H., Nachtsheim, C. J., and Wasserman, W. (1996). *Applied Linear Statistical Models*. Irwin Publishing, fourth edition.

O'Hagan, A. (1998). A Markov property for covariance structures. Statistics Research Report 98-13, Nottingham University.

O'Hagan, A. and Forster, J. (1994). *Bayesian Inference*, volume 2B of *Kendall's Advanced Theory of Statistics*. Arnold, London.

Pan, Z. and Eigenmann, R. (2006). Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA. IEEE Computer Society.

PathScale (2005). Pathscale EKOPath compilers. `http://www.pathscale.com`.

Pelikan, M. and Goldberg, D. E. (2001). Escaping hierarchical traps with competent genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 511–518, San Francisco, California, USA. Morgan Kaufmann.

Pelikan, M. and Goldberg, D. E. (2003). Hierarchical BOA solves ising spin glasses and MAXSAT. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1271–1282. Springer.

Pelikan, M., Goldberg, D. E., and Cantú-paz, E. E. (2000). Linkage problem, distribution estimation, and Bayesian networks. *Evolutionary Computation*, 8(3):311–340.

Pelikan, M., Goldberg, D. E., and Lobo, F. (1999). A survey of optimization by building and using probabilistic models. Technical Report IlliGAL-99018, Illinois Genetic Algorithms Laboratory.

Pinkers, R., Knijnenburg, P., Haneda, M., and Wijshoff, H. (2004a). Analysis of compiler options using orthogonal arrays. In *Proceedings of the 11th International Workshop on Compilers for Parallel Computers*, pages 137–148.

Pinkers, R. P. J., Knijnenburg, P. M. W., Haneda, M., and Wijshoff, H. A. G. (2004b). Statistical selection of compiler options. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 494–501, Washington, DC, USA. IEEE Computer Society.

Quiñonero-Candela, J., Rasmussen, C. E., and Williams, C. K. I. (2007). Approximation methods for Gaussian process regression. In *Large Scale Kernel Machines*. MIT Press.

Rasmussen, C. E. and Ghahramani, Z. (2002). Infinite mixtures of Gaussian process experts. In Diettrich, T. G., Becker, S., and Ghahramani, Z., editors, *Advances in Neural Information Processing Systems 14*. MIT Press.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, Massachusetts.

Rubinstein, R. Y. and Kroese, D. P. (2004). *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*. Information Science and Statistics. Springer-Verlag New York, Inc., New York, USA.

Schölkopf, B. and Smola, A. J. (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA.

Schwaighofer, A., Tresp, V., and Yu, K. (2005). Learning Gaussian process kernels via hierarchical Bayes. In *Advances in Neural Information Processing Systems 17*, Cambridge, MA. MIT Press.

Schwarz, G. (1978). Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464.

Shapiro, J. L. (2003). Scaling of probability-based optimization algorithms. In S. Becker, S. T. and Obermayer, K., editors, *Advances in Neural Information Processing Systems 15*, pages 383–390. MIT Press, Cambridge, MA.

Shapiro, J. L. (2005). Drift and scaling in estimation of distribution algorithms. *Evolutionary Computation*, 13(1):99–125.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.

Singer, J., Brown, G., and Watson, I. (2007). Branch prediction with Bayesian networks. In *Proceedings of the First Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation*, pages 96–112.

Stephenson, M. and Amarasinghe, S. (2004). Predicting unroll factors using nearest neighbors. Technical Report MIT-TM-938, Massachusetts Institute of Technology.

Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. (2003). Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–90, New York, NY, USA. ACM Press.

Stephenson, M. and Amarasinghe, S. P. (2005). Predicting unroll factors using supervised classification. In *3rd IEEE/ACM International Symposium on Code Generation and Optimization, 20-23 March 2005, San Jose, CA, USA*, pages 123–134. IEEE Computer Society.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, USA.

Teh, Y. W., Seeger, M., and Jordan, M. I. (2005). Semiparametric latent factor models. In Cowell, R. G. and Ghahramani, Z., editors, *Proceedings of the 10th International Conference on Artificial Intelligence and Statistics*, pages 333–340. Society for Artificial Intelligence and Statistics.

Texas Instruments (2003). TMS320C6000 chip support library API user's guide.

Thrun, S. (1996). Is learning the n-th thing any easier than learning the first? In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*, pages 640–646. The MIT Press.

Thrun, S. and O'Sullivan, J. (1996). Discovering structure in multiple learning tasks: The TC algorithm. In *Proceedings of the 13th International Conference on Machine Learning*, pages 489–497. Morgan Kaufmann.

Thrun, S. and Pratt, L., editors (1998). *Learning to Learn*. Kluwer Academic Publishers.

Tipping, M. E. and Bishop, C. M. (1999). Probabilistic principal component analysis. *Journal of the Royal Statistical Society, Series B*, 61(3):611–622.

Triantafyllis, S., Vachharajani, M., and August, D. I. (January 2005). Compiler optimization-space exploration. *The Journal of Instruction-Level Parallelism*, 7:1–25.

Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. I. (2003). Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code*

Generation and Optimization, pages 204–215, Washington, DC, USA. IEEE Computer Society.

Vishwanathan, S. V. N. and Smola, A. J. (2003). Fast kernels for string and tree matching. In Becker, S., Thrun, S., and Obermayer, K., editors, *Advances in Neural Information Processing Systems 15*, pages 569–576, Cambridge, MA, USA. MIT Press.

Vuduc, R., Demmel, J., and Bilmes, J. (2004). Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94.

Wackernagel, H. (1998). *Multivariate Geostatistics: An Introduction with Applications*. Springer-Verlag, Berlin, 2nd edition.

Weston, J., Chapelle, O., Elisseeff, A., Schölkopf, B., and Vapnik, V. (2003). Kernel dependency estimation. In Becker, S., Thrun, S., and Obermayer, K., editors, *Advances in Neural Information Processing Systems 15*, pages 873–880, Cambridge, MA, USA. MIT Press.

Williams, C. K. I. (1997). Computing with infinite networks. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*, pages 295–301. The MIT Press.

Williams, C. K. I., Chai, K. M. A., and Bonilla, E. V. (2007). A note on noise-free Gaussian process prediction with separable covariance functions and grid designs. Technical Report EDI-INF-RR-1228, University of Edinburgh.

Wu, H., Chen, L., Manzano, J., and Gao, G. R. (2006a). A user-friendly methodology for automatic exploration of compiler options. In *Proceedings of the International Conference on Programming Languages and Compilers*. CSREA Press.

Wu, H., Park, E., Chen, L., del Cuvillo, J., and Gao, G. R. (2006b). User-friendly methodology for automatic exploration of compiler options: A case study on the Intel XScale microarchitecture. In *Proceedings of the International Conference on Programming Languages and Compilers*. CSREA Press.

Wu, H., Park, E., Kaplarevic, M., Zhang, Y., Bolat, M., Li, X., and Gao, G. R. (2007). Automatic program segment similarity detection in targeted program performance improvement. In *Parallel and Distributed Processing Symposium*, pages 1–8. IEEE International.

Yu, K., Chu, W., Yu, S., Tresp, V., and Xu, Z. (2007). Stochastic relational models for discriminative link prediction. In *Advances in Neural Information Processing Systems 19*, Cambridge, MA. MIT Press.

Yu, K., Tresp, V., and Schwaighofer, A. (2005). Learning Gaussian processes from multiple tasks. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 1012–1019, New York, NY, USA. ACM Press.

Yu, S., Yu, K., Tresp, V., and Kriegel, H.-P. (2006). Collaborative ordinal regression. In Cohen, W. W. and Moore, A., editors, *Proceedings of the 23rd International Conference on Machine Learning*, pages 121–128, New York, NY, USA. ACM.

Zhang, H. (2007). Maximum-likelihood estimation for multivariate spatial linear coregionalization models. *Environmetrics*, 18(2):125–139.

Zhang, W. and Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1114–1120. Morgan Kaufmann.

Zhao, M., Childers, B. R., and Soffa, M. L. (2003). Predicting the impact of optimizations for embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 1–11. ACM Press.