Using Machine-Learning to Efficiently Explore the Architecture/Compiler Co-Design Space

Christophe Dubach

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2009

Abstract

Designing new microprocessors is a time consuming task. Architects rely on slow simulators to evaluate performance and a significant proportion of the design space has to be explored before an implementation is chosen. This process becomes more time consuming when compiler optimisations are also considered. Once the architecture is selected, a new compiler must be developed and tuned. What is needed are techniques that can speedup this whole process and develop a new optimising compiler automatically.

This thesis proposes the use of machine-learning techniques to address architecture/compiler co-design. First, two performance models are developed and are used to efficiently search the design space of a microarchitecture. These models accurately predict performance metrics such as cycles or energy, or a tradeoff of the two. The first model uses just 32 simulations to model the entire design space of new applications, an order of magnitude fewer than state-of-the-art techniques. The second model addresses offline training costs and predicts the average behaviour of a complete benchmark suite. Compared to state-of-the-art, it needs five times fewer training simulations when applied to the SPEC CPU 2000 and MiBench benchmark suites.

Next, the impact of compiler optimisations on the design process is considered. This has the potential to change the shape of the design space and improve performance significantly. A new model is proposed that predicts the performance obtainable by an optimising compiler for any design point, without having to build the compiler. Compared to the state-of-the-art, this model achieves a significantly lower error rate.

Finally, a new machine-learning optimising compiler is presented that predicts the best compiler optimisation setting for any new program on any new microarchitecture. It achieves an average speedup of 1.14x over the default best *gcc* optimisation level. This represents 61% of the maximum speedup available, using just one profile run of the application.

Acknowledgements

First of all, I wish to thank my adviser, Prof. Michael O'Boyle, for his guidance and the wise advice he gave me throughout my three years of PhD study.

I would also like to thank my colleagues and friends from the CArD research group for all the interesting discussions and interactions we had. In particular, Tim and Edwin with whom I had successful collaborations, resulting in parts of this thesis. Thanks also to everyone who took the time to read through parts of my thesis (Mike, Tim, Björn, Hugh and John).

I would like to thank all my friends in Edinburgh with whom I have had a great time during the last three years. In particular Dominik, Jorge, Chris, Gaya and the other members of the "PhDUnsocials" clan.

Finally, I am grateful to my parents, who have always respected my choices and constantly encouraged me to pursue my goals.

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. Some of the material used in this thesis has been published in the following papers:

- Christophe Dubach, Timothy M. Jones, and Michael F. P. O'Boyle "Microarchitectural Design Space Exploration Using An Architecture-Centric Approach". In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, 2007

- Christophe Dubach, Timothy M. Jones, and Michael F.P. O'Boyle. "Exploring and Predicting the Architecture/Optimising Compiler Co-Design Space". In: *Proceedings of the 2008 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2008.

(*Christophe Dubach*)

Table of Contents

Chapter 1

Introduction

Computer systems have become increasingly complex. The interaction between processors, compilers and application software, means that these systems are hard to build. In addition, time-to-market has become one of the major driving forces behind the development of computer systems and, as such, has put pressure on system designers.

In this fast evolving environment, designers require powerful tools that *automate* part of the design process. By using such tools, the design of computer systems becomes more efficient. However, it remains a difficult and time consuming task, where human expertise plays a critical role.

On the hardware front, microprocessors have become increasingly diverse and complex. Designers have to find new ways of using the large numbers of transistors available, which steadily increase according to Moore's law. The use of specialised EDA (Electronic Design Automation) tools is therefore essential in facilitating the design process. Processors are typically designed and described with the help of high-level languages. From this description, the complete process is automated down to the transistor level. However, since more options are available to the designer through the use of higher-level languages, more time is spent testing alternative designs with different parameters.

On the compiler front, considerable effort has been spent in developing powerful compiler technology. Programs are no longer written in assembly code (except in some rare cases) and the increased use of high-level programming languages means that more pressure is put onto the compilers. Compilers have to perform sophisticated high-level optimisations in order to extract performance from the application. However, the design of such compilers, and especially the design of compilers that can optimise code efficiently for different architectures, has become too complex to be tackled by human expertise alone. New techniques are therefore required that can assist the designer in this task.

1.1 Machine-learning for Computer Systems

Machine-learning techniques have been successfully used in various fields such as robotics, image processing and finance. These techniques are used to automatically understand the internal structure of data and make new predictions about it. Although machine-learning is a well established and recognised field, there has been very little application to computer systems in comparison to other disciplines.

Automatic program optimisation has been investigated for decades and numerous static models embedded within compilers have been developed. The recent application of machine-learning to the problem of automating the generation of optimising compilers has the potential to change this field. Recent results show that machine-learning models, generated automatically, already out-perform the human-coded analysis present in today's compilers. However, much work remains to be done in this area since these models still require extensive training for every new architecture encountered.

Recently, predictive models have been proposed for the design space exploration of new microprocessors. However, architects remain sceptical about the usefulness of these approaches; such techniques suffer from the large number of simulations required for training. Architects may therefore avoid using statistical approaches for the design of new processors, relying on more conventional analytic techniques. The adoption of machine-learning in this field faces many challenges. For these techniques to have greater take-up, architects need to be convinced that machine-learning provides the necessary tools to leverage the current design methodology and make the whole design process faster.

This thesis investigates the use of machine-learning for the efficient design of processors and optimising compilers. The reminder of this chapter introduces the problem, lists the contributions of this work and presents the overall structure of the thesis.

1.2 The Problem

Processor Design During the design process of general purpose processors, the architect must first determine the overall architecture of the processor. This involves defining the different instructions supported, the structure of the pipeline and the organisation of the memory hierarchy, for example. This task is typically performed by an experienced designer.

Once this architecture has been established, a certain number of parameters need to be tuned in order to achieve the design goals and requirements. These parameters include elements such as the number of functional units, the size of the register file or the sizes of the different caches. This phase, known as *design space exploration*, involves running numerous simulations to test alternative possible implementations. This exploration typically gathers information about important metrics such as performance or energy and ideally results in a design point that fulfils all the requirements and constraints.

However this exploration phase can take a long time since simulators are typically slow and the number of parameter combinations large. The designer has to carefully balance each of the parameters and investigate their complex interactions in order to find a good design point. The space of all possible designs, *i.e. the design space*, quickly becomes impractical to explore. What is needed are techniques that can automate this process as much as possible and provide the designer with a small set of promising design points where he can focus his efforts.

Co-Design In the embedded world, the set of programs that will be executed on the final system is often known in advance. Architects can take advantage of this when they design a new processor and, therefore, optimise the processor towards these applications. For instance if all the programs are using integer computation only, there is no need to integrate floating point units into the processor. The end-result is typically a processor with high efficiency and low energy consumption for a particular domain of applications.

The embedded world is dominated by time-to-market constraints. Therefore it is crucial to develop new microprocessors as efficiently as possible to maintain a competitive advantage. In addition, cost issues and the tight power budget mean that a significant amount of time is spent tuning these processors and the associated applications.

However, in the current embedded processor design methodology, the design process is often conducted in two separate phases. First a processor is designed for some target programs and, in a later stage, these programs might be optimised by hand or by the compiler for the newly developed processor. Clearly this is a sub-optimal way of designing systems. The compiler team may not be able to deliver a compiler that achieves the architect's expectations. More fundamentally, if one knew the performance that an eventual optimising compiler could achieve on any architecture, then a completely different architecture may be chosen. This inability to directly investigate the combined architecture/optimising compiler spaces, *i.e. the co-design space*, means tomorrow's architectures are being designed based on yesterday's compiler technology.

Optimising Compiler Once a processor has been designed, a new compiler must then be built. This generally involves using the compiler infrastructure of the previous generation of processors and adapting it for the new target. Significant engineering effort has been spent in developing retargetable compilers. Such compilers offer the possibility of easily and quickly adapting a generic infrastructure to the newly developed processor. However, because of their generic nature, these compilers need to be tuned by hand for each target architecture in order to extract the performance available.

Companies that design processors spend much effort in developing the toolchain necessary to use the system. Once developed, this toolchain, which integrates the compiler, is generally used for all subsequent implementations of the architecture. Therefore the original investment can be amortised over time. However, because this compiler may have been designed independently of the specific details of the microarchitecture, it might fail to extract the level of performance available in the processor when compiling new programs. Therefore these compilers can be suboptimal.

In recent years, the emergence of *machine-learning compilers* has tried to tackle this problem. These compilers, which integrate machine-learning techniques, learn an optimal optimisation strategy for a given architecture and then predict the correct set of optimisations to apply when compiling new programs. However, these compilers need to be retrained whenever changes occur in the microarchitecture. This means building one compiler for every possible design implementation, which is clearly infeasible.

1.3 Contributions

This thesis presents new techniques, based on machine-learning, that address the issues encountered during the design of microprocessors and the generation of their corresponding optimising compiler.

In this thesis a novel approach to efficiently explore the design space of new microprocessors is presented. By building upon prior work, new machine-learning models are investigated that make use of information across programs. The key contribution lies in the way knowledge is transferred across programs. By exploiting existing similarities between programs, the total number of simulations required to explore the design space can be reduced significantly.

In addition, the co-design space of embedded processors is also considered for the first time. This co-design space combines the processor design space with the compiler optimisation space. A machine-learning model is built that can automatically *predict* the performance of an optimising compiler across an arbitrary microarchitectural space without having to tune the compiler first. This allows the designer to accurately determine the performance of any architecture as if an optimising compiler were available. Given a small sample of the architecture and optimisation space, this model can then predict the performance of a yet-to-be-built optimising compiler using information gained from a non-optimising baseline compiler.

Ultimately, this has the potential to drive a change in the current methodology of designing embedded processors.

Finally a machine-learning optimising compiler is presented that can achieve a significant portion of the best performance available in the compiler space, for any microarchitecture. This compiler is named TALC: the Trans-Architecture Learning Compiler. Given a new microarchitecture, it automatically determines the right optimisation settings to apply for any new program with just one profile run. This approach is based on machine-learning where a model is learnt off-line "at the factory". The learning process is a one-off activity whose cost is amortised across all future uses of the compiler on any variation of the processor's base architecture. Given this approach, a new compiler does not need to be developed when the processor microarchitecture changes. This allows compilers to become fully integrated in the design space exploration of new processors, helping designers to fully evaluate the potential of any new architecture. In addition, this enables the design of parametrised embedded processors that can be shipped with this compiler. Customers that acquire such a design do not need to tune the compiler again for their specific implementation since the compiler is generic and will know, given the specific microarchitectural parameters chosen by the customer, how to compile optimally for it.

1.4 Structure

This thesis is organised as follows.

Chapter 2 introduces the different machine-learning techniques used throughout this thesis and discusses the evaluation methodology.

Chapter 3 presents the related work. Prior work on design space exploration is discussed, which includes simulation methodologies and the use of predictive models. Then work related to compiler optimisation space exploration is reviewed and, in particular, the use of machine-learning techniques to build optimising compilers.

Chapter 4 investigates two machine-learning models that can be used to efficiently explore the design space of new processors. The first of these models makes a prediction for any new program based on data gathered from a training set. The second model extends this and predicts the average behaviour of a complete benchmark suite. This allows a significant reduction in the number of simulations needed to conduct design space exploration. This chapter is based partially on the work published in [Duba 07a].

Chapter 5 explores the co-design space of the combined microarchitecture design and compiler optimisation space. This exploration demonstrates that compiler optimisations have a significant impact at the design stage and should not be omitted. A machine-learning model is then developed that automatically predicts this co-design space and helps the designer to make better decisions. This chapter is based on the work published in [Duba 08].

Chapter 6 develops a machine-learning compiler that adapts its optimisation strategy to the particular program being compiled on any microarchitecture from the design space. This proposed compiler, TALC, is the first to optimise a new program for any microarchitectural variation.

Chapter 7 finally concludes this thesis by summarising the contributions, providing a critical analysis of this work and discussing future work.

1.5 Summary

This chapter has introduced this thesis, outlining the problems encountered when designing processors and generating an optimising compiler. It has advocated the use of machine-learning to improve the design process and automatically generate an optimising compiler. The contributions of this work have been listed and an outline of the thesis described. The next chapter provides a short introduction to the machine-learning methodology used throughout this thesis.

Chapter 2

Machine-Learning and Evaluation Methodology

2.1 Introduction

This chapter gives a short overview of the machine-learning techniques and the evaluation methodology used in this thesis. However, it is not exhaustive, further information can be found in [Bish 06], which contains a complete description of the different techniques presented.

This chapter is organised as follows: first the terminology is defined in section 2.2, then section 2.3 describes unsupervised learning techniques and section 2.4 follows with regression techniques. Finally section 2.5 describes the methodology used to evaluate the different machine-learning models developed in this thesis.

2.2 Terminology

The term *machine-learning* is used to define techniques that allow computers to learn. These techniques, mostly based on statistical approaches, focus on extracting information from data *automatically*.

An observation, or a data point, is defined as a pair $< \mathbf{x}, y >$ where $\mathbf{x}$ represents an input vector and y the output value. The input vector is referred as the *features* while the output value is referred to as the *response*.

The *feature space* is defined as an abstract space where each sample is represented as a data point in a n-dimensional space. The dimension n of this space is determined by the total number of features.

Typically a relation exists between the features $\mathbf{x}$ and the response y denoted $\mathbf{x} \rightarrow y$. The task of any machine learning technique consists of determining this relation. This allows one to make predictions for new, unseen observations based on past observations.

2.3 Unsupervised Learning

Unsupervised learning techniques only make use of the feature space to analyse the inherent structure of the data. This contrasts with supervised learning methods, such as the regression techniques presented later in section 2.4, where the output space is used.

This section first presents the *Principal Components Analysis* (PCA) technique which is typically used to reduce the dimensionality of the feature space. Then, two clustering techniques, namely K-Means and hierarchical clustering are presented. Clustering techniques are an effective, yet simple, way of analysing data. They typically use the input space information to segment or group the observations according to the distance between each data point. This distance measure is typically computed using the Euclidean distance but other distance measures can be used as well.

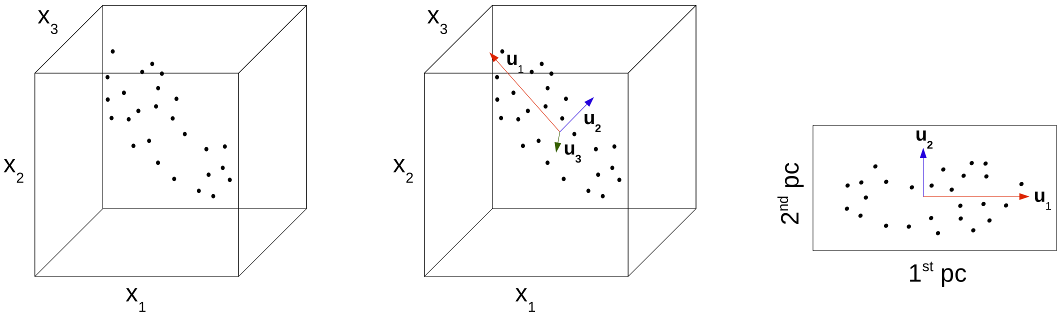
2.3.1 Principal Components Analysis

Sometimes the feature space contains too many dimensions. Some of these dimensions convey very little information and are often redundant. Therefore, it is desirable to reduce the dimensionality of the feature space in order to help discover the underlying data structure and improve the performance of the predictive models.

PCA (Principal Components Analysis) [Pear 01] is a technique that identifies the main components that are responsible for the observed variance in the data. Therefore it can be used to reduce the dimensionality of the input space. It transforms the original data into a new space, using an orthogonal linear transformation. This transformation ensures that the greatest variance observed in the data lies on the first coordinate, the second greatest on the second and so on. The idea is that only a few of these new components are necessary to express the original data and to keep much of the original variance.

Let $\mathbf{X}$ be a matrix of size $m \times n$, where the rows represent m observations and the columns n features. To extract the main components, the correlation matrix $\mathbf{C}$ is first computed. This correlation matrix is simply built by considering the pair-wise coefficients of correlation between the original variables (defined later in section 2.5.3). Then, the eigenvalues $\lambda_1 \dots \lambda_m$ of $\mathbf{C}$ and the associated unit eigenvectors $\mathbf{u_1} \dots \mathbf{u_m}$ are computed. The eigenvectors and eigenvalues satisfy:

$$\mathbf{C} \cdot \mathbf{u_i} = \lambda_i \cdot \mathbf{u_i} \tag{2.1}$$



(a) Input space **(b) Three main components** **(c) Projected space**

Figure 2.1: Example of applying PCA to a three-dimensional space (a). Three main components are first computed (b). Then, only the two main components are kept and the input space can be projected on a two-dimensional space (c).

if there is a non trivial solution **u**. The *characteristic equation*, defined as:

$$\det(\mathbf{C} - \lambda \cdot I) = 0 \tag{2.2}$$

is used to compute the eigenvalues $\mathbf{u_i}$, where det is the determinant of the matrix and I the identity matrix. Once the eigenvalues are known, the computation of the eigenvectors follows from equation 2.1. These eigenvectors correspond to the principal components and the eigenvalues to the associated variance.

The original data can then be expressed as a new matrix $\mathbf{X}' = [\mathbf{u}_1 \dots \mathbf{u}_m]^T \cdot \mathbf{X}$. By using only the first few principal components p, one can reduce the dimensionality of the data while conserving most of the variance. The transformed data can then be expressed as $\mathbf{X}' = [\mathbf{u}_1 \dots \mathbf{u}_p]^T \cdot \mathbf{X}$ where the total variance is equal to $\sum_1^p \lambda_i$. Typically the number of main components, p, is determined so as to keep as much of the total variance of the original data as possible.

Figure 2.1 shows how this technique can be used to reduce the dimensionality of a feature space. In this example, the input space is three-dimensional determined by the axis x_1, x_2 and x_3 as shown in figure 2.1(a). The three main components u_1, u_2 and u_3 corresponding to the data points are shown in figure 2.1(b). As can be seen the components u_1 and u_2 account for the largest variance. In figure 2.1(c) the three-dimensional space has been projected using only the two main components, conserving much of the original data structure.

2.3.2 K-Means

K-Means is one of the simplest clustering techniques. This algorithm clusters m observations into k classes, where $k < m$. The objective of this technique consists of minimising the total

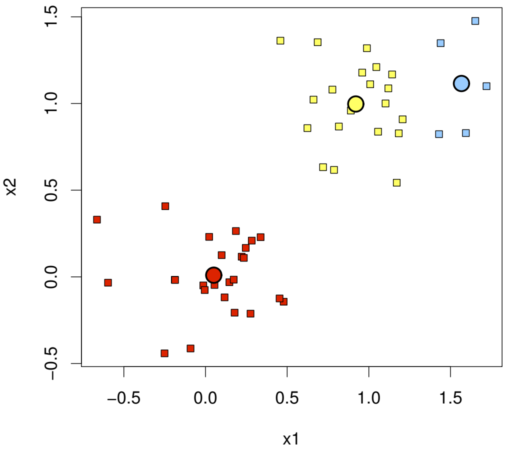


Figure 2.2: Application of the K-Means algorithm to an arbitrary dataset represented by the square. The plain circles represent the centre of each cluster. Each data point belongs to the cluster whose centre is the closest to it.

intra-cluster variance defined as:

$$\sum_{i=1}^{k} \sum_{\mathbf{x}_j \in C_i} (\mathbf{X}_{,j} - \mu_i)^2 \tag{2.3}$$

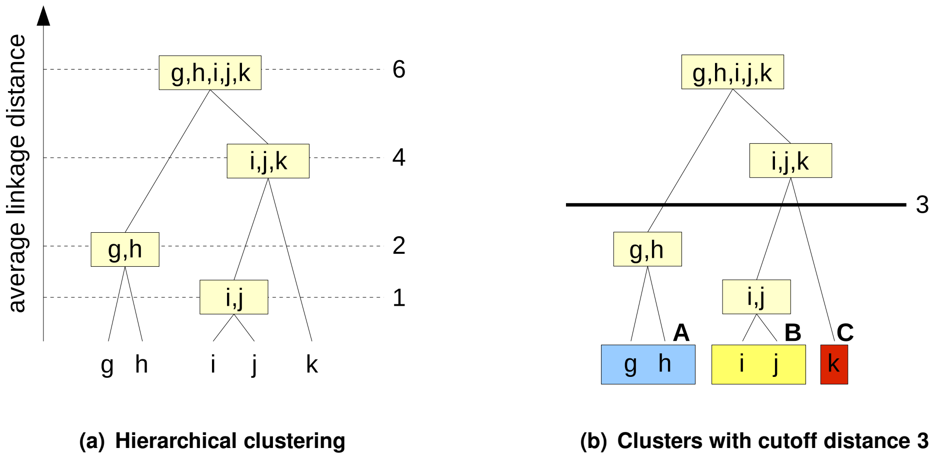
where C_i represents cluster i and μ_i its centre.

In this work, Lloyd's algorithm [Lloy 82] is used to find the clusters. It starts by randomly assigning each observation to a cluster. It then calculates the k centres which are simply the mean of the observations that belong to each cluster. Then it reassigns each observation to the closest cluster's centre. The centres are then computed again and this process is repeated until it converges to a stable solution.

The result of this algorithm can be seen in figure 2.2 on a arbitrary dataset. The number of clusters k has been fixed to three in this case. As can be seen, this technique clusters points that are close to each other in the input space. The centres of the clusters resulting from the application of the algorithm are marked with the plain circles.

2.3.3 Hierarchical Clustering

Hierarchical clustering [Lanc 67] takes a slightly different approach to the clustering problem. Instead of separating the data into different groups, it builds a hierarchical tree of distances between each data point. The algorithm used in this thesis is an agglomerative method. It produces a sequence of partitions $P_n, P_{n-1}, \ldots, P_1$ where the first partition, P_n, has n clusters



(a) Hierarchical clustering

(b) Clusters with cutoff distance 3

Figure 2.3: Example of hierarchical clustering. The result of the algorithm using average linkage distance is shown (a). Three clusters A, B and C are created when a cutoff distance of 3 is chosen (b).

each containing one data point and the last, P_1, has one cluster containing all the data points. It starts with each point assigned to one cluster. Then it iteratively merges the pairs of clusters that are the closest to each other.

The distance measure used to determine how close clusters are to each other is the average distance. The distance between two clusters A and B, called the *linkage* distance, is computed as:

$$\mathrm{D}(A,B) = \frac{\sum_{a\in A}\sum_{b\in B} d(a,b)}{|A|\cdot|B|} \tag{2.4}$$

where $d(a,b)$ is the Euclidean distance between two points a and b, and $|A|$ and $|B|$ are the cardinalities (number of elements) of clusters A and B respectively.

Figure 2.3 shows an example of hierarchical clustering. As can be seen in figure 2.3(a), elements g and h are separated by a distance of 2 whereas elements i and j are separated by a distance of 1. On average the distance between k and the elements i and j is 4. Given this tree, creating clusters is simply a matter of defining a cutoff distance. For instance, if a cutoff distance of 3 is chosen, figure 2.3(b), then three clusters A, B and C are determined. This implies that all the elements within the same cluster will be, on average, at a distance of less than 3 between each other.

This technique is particularly useful in analysing similarities between data points. In fact this type of clustering is used in chapter 4 to analyse program similarities and isolate the programs that differ significantly from the others.

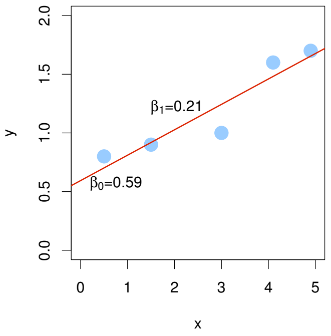


Figure 2.4: Example of linear regression where the resulting regression line is shown for five points. This line is defined as $y = \beta_0 + \beta_1 \cdot x$ where $\beta_0 = 0.59$ and $\beta_1 = 0.21$.

2.4 Regression Techniques

The regression problem consists of finding the function $f : \mathbf{x} \longrightarrow y$ that links the input variables $\mathbf{x}$ to the output y. Using this function, an estimation or *prediction* $\hat{y}$ of the real value y can be made. First, linear regression is presented followed by the K-Nearest Neighbours technique. Then Artificial Neural Networks are introduced and finally Support Vector Machines for regression are presented.

2.4.1 Linear Regression

This form of regression assumes a linear relationship between the input and the output. It uses a linear combination of the input $\mathbf{x}$ to predict the output y. This combination is expressed as a weighted sum, whose weights β are determined so as to minimise the squared error between real outputs $\mathbf{y}$ and the predictions $\hat{\mathbf{y}}$. This sum is computed as follows:

$$\hat{\mathbf{y}} = \beta_0 + \beta_1 \cdot \mathbf{X}_{,1} + \cdots + \beta_m \cdot \mathbf{X}_{,m} \tag{2.5}$$

The task of linear regression consists of finding the optimal weights β_j that minimise the squared error defined as:

$$\sum_{i=1}^{n} \sum_{j=1}^{m} (\mathbf{X_{i,j}} \cdot \beta_j - y_i)^2 \tag{2.6}$$

It can be shown that the weights β that minimise the total squared error are given by:

$$\beta = (\mathbf{X} \cdot \mathbf{X}^T)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y} \tag{2.7}$$

Figure 2.4 shows an example of linear regression. The red line minimises the total squared error. In this example the line that estimates the data is defined by $\hat{y} = \beta_0 + \beta_1 \cdot x$ with $\beta_0 = 0.59$ and $\beta_1 = 0.21$. The weight β_0 is in fact the intercept and β_1 the slope of the linear equation.

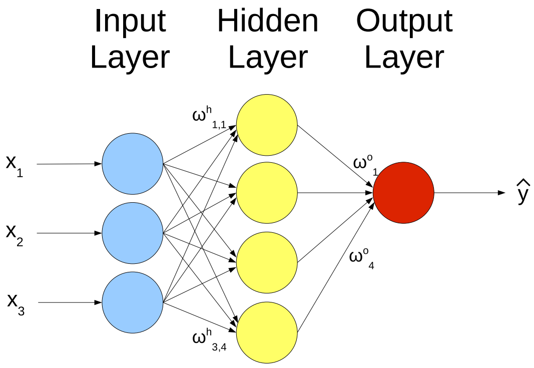


Figure 2.5: Example of an artificial neural network where a simple multi-layered neural network is shown. This network is composed of three layers; input, hidden and output.

2.4.2 K-Nearest Neighbours

K-Nearest Neighbours (KNN) is amongst the simplest types of regression models, classified as an instance-based learning technique. The prediction for a new data point is simply made by taking the responses' average, or outputs' average, of the k nearest data points from the training set. The average is typically computed as a weighted sum where the weights are associated with a distance measure between the new data point and the k nearest ones.

2.4.3 Artificial Neural Networks

Artificial Neural Networks (ANNs) can be used when non-linearity is desired. They use a network of neurons to map the input variables to a response or prediction. Each neuron in the network is connected by weighted edges, as can be seen in figure 2.5 which represents a typical feed-forward multi-layered neural networks.

Feed-forward network The output of each neuron is computed as follows:

$$f(\mathbf{x}) = g(\sum_i \omega_i \cdot \mathbf{x}_i) \tag{2.8}$$

where $g()$ is an activation function.

The activation function $g()$ is defined differently depending on the layer. The tangent hyperbolic function $g(x) = tanh(x)$ is typically used for the hidden layer since it produces values between -1 and 1, necessary to normalise the output. In the case of regression, the output activation function is the identity function, allowing extrapolation. Note that the input neurons are in fact just forwarding the input x_i.

It follows that the prediction $\hat{y}$ made by the network is:

$$\hat{y} = \sum_i (\omega_i^o \cdot tanh(\sum_j \omega_{j,i}^h \cdot x_i)) \tag{2.9}$$

where ω^o are the weights associated with the output layer and ω^h the weights of the hidden layer, as can bee seen in figure 2.5.

Back-propagation The training phase of the network consists of finding the weights ω in order to minimise the prediction error. Typically the back-propagation algorithm is used. This algorithm initialises the weights to random values. Then it computes the prediction error and propagates this back to each individual neuron using the *learning rule* defined in the next paragraph. For each neuron, it then computes what the output should have been and updates the neuron's weights accordingly. The algorithm is repeated until the error is below a certain threshold.

Learning rule This rule determines how the weights of each neuron are updated. For the output neurons, the change of the weight is defined by:

$$\Delta\omega_i^o = \eta \cdot \delta^o \cdot x_i^h \cdot g'(\hat{y}) = \eta \cdot \delta^o \cdot x_i^h \tag{2.10}$$

$$\delta^o = (y - \hat{y}) \tag{2.11}$$

where η is the learning rate constant and x_i^h the output of the i^{th} hidden neuron.

In the case of the hidden neurons, the change is:

$$\Delta\omega_{j,i}^h = \eta \cdot \delta^h \cdot x_j \cdot g'(x_i^h) \tag{2.12}$$

$$\delta^h = \delta^o \cdot \omega_i^0 \tag{2.13}$$

These learning rules are derived from the fact that the squared error of predictions $\hat{y}$ with the actual outputs y needs to be minimised.

2.4.4 Support Vector Machines for Regression

With Support Vector Machines (SVM) for regression [Smol 03] the basic idea is to map the input data $\mathbf{X}$ into a high-dimension space via a nonlinear mapping. Then a simple linear regression can be performed in this new space. In particular, the goal is to find a function $f(x)$ that has at most ε deviations from any of the training targets y.

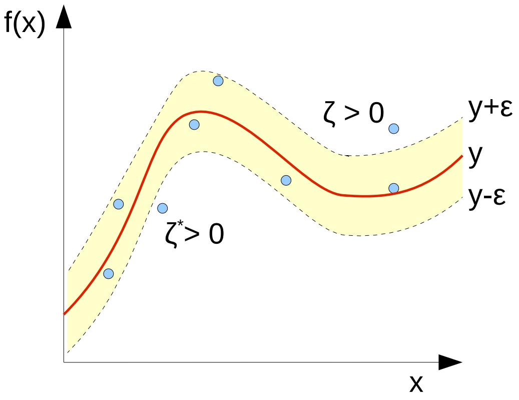


Figure 2.6: Example of SVM for regression showing the regression curve y with the deviation ε. The point above the curve has the associated slack variable $\zeta > 0$ while the point below the curve has the slack variable $\zeta^* > 0$.

The problem can be formulated as:

$$\begin{array}{ll} \text{minimise} & \frac{1}{2}||\omega||^2 + C \cdot \sum_{i=1}^{N}(\zeta_i + \zeta_i^*) \\ \text{subject to} & \begin{cases} (\omega^T \cdot \phi(x_i) + b) - y_i \leq \varepsilon + \zeta_i \\ y_i - (\omega^T \cdot \phi(x_i) + b) \leq \varepsilon + \zeta_i^* \\ \zeta_i, \zeta_i^* \geq 0, i = 1, \cdots, N, \varepsilon \geq 0 \end{cases} \end{array} \tag{2.14}$$

where $< x_i, y_i >$ represents a data pair and N the number of training samples, b is a constant, ζ_i and ζ_i^* are slack variables to cope with otherwise infeasible constraints of the optimisation problem, ω is a vector of coefficients and $C > 0$ the capacity constant. This constant determines the trade-off between the largest deviation ε and the *flatness* of the hyperplane. A high value of C means that the model will tend to overfit.

The mapping to a higher-dimensional space is performed with the help of mathematical functions called kernels. The most popular kernel is the radial basis function defined as:

$$\phi = e^{-\gamma||\mathbf{x}-\mathbf{c}_i||^2} \tag{2.15}$$

for $\gamma > 0$.

Figure 2.6 illustrates the SVM for regression. As can be seen the points that are within the deviation around the regression curve have the slack variables ζ and ζ^* equal to zero. For these points, the SVM for regression ensures that their deviation is at most ε. Depending on the value of the capacity constant C, in equation 2.14 above, a few points are allowed to deviate by more than ε from the regression curve. These points have their associated slack variables ζ or ζ^* greater than 0.

2.5 Evaluation Techniques

This section now presents the different methods employed throughout this thesis to evaluate the accuracy of the models developed and the relationship between the feature space and responses.

2.5.1 Validation of the Models

One important aspect of any machine-learning technique is its validation. The method known as *cross-validation* was used in this thesis to evaluate all the machine-learning models built. This technique ensures that the data used to test the accuracy of a model has not been used during the training phase.

In particular two well established techniques were used; *leave-one-out cross-validation* and *repeated random sub-sampling validation*. These validation techniques are discussed below.

Leave-one-out cross-validation This technique is used when the total number of data points is small. It cycles through all the data points and builds the training set by leaving out only one data point. So if the data contains N points, the training set will be composed of the $N-1$ points and the test set will consist of the unique N^{th} point left out. Once the accuracy has been assessed for each test point, an average is usually computed which gives the final accuracy of the scheme.

Another application of this validation technique consists of selecting an entire group of data points that share a common property to leave out for validation. Consider for instance the case where several data points are collected per program. In this situation, all the data points generated from a particular program form a group. Therefore, when leave-one-out cross-validation is applied, it uses all the points from one program to validate the model and the data points of all the other programs to train. In this thesis this is how leave-one-out cross-validation is applied.

Repeated random sub-sampling validation When the number of data points is large, another technique can be applied to validate a model. This technique randomly selects the training data points and use the remaining for testing. This selection and training process is repeated a certain number of times (typically 20) and the accuracy is then averaged. The advantage of using this technique is that it also gives an estimation of the prediction's variance since the training set is selected randomly.

2.5.2 Mean Error

The *relative mean absolute error* (rmae) is used to evaluate the prediction error of a model. It is defined as:

$$\frac{1}{m}\sum_{i}^{m}\left|\frac{\hat{\mathbf{y}}_i-\mathbf{y}_i}{\mathbf{y}_i}\right| \tag{2.16}$$

This measure gives the average error between the predicted output $\hat{y}$ and the real output y for each data point in the test set. Note that this value is normalised by the real output in order to ease the comparison between different datasets. An rmae of 100% means that the prediction is double the real value.

2.5.3 Coefficient of Correlation

Another way of measuring the performance of a model is by looking at the correlation between the predicted outputs $\mathbf{y}$ and real outputs $\hat{\mathbf{y}}$. In addition, this coefficient can also be used for analysing the relationship between the features and the responses. For this reason it is defined in generic terms using two variables X and Y.

The coefficient of correlation between two variables X and Y is defined as:

$$\rho_{X,Y}=\frac{\mathbf{cov}(X,Y)}{\sigma_X\cdot\sigma_Y}, \tag{2.17}$$

where σ_X and σ_Y are the standard deviations of variables X and Y respectively and $\mathbf{cov}(X,Y)$ is the covariance of variables X and Y. These two values are defined as:

$$\begin{aligned}\sigma_X &= \sqrt{\tfrac{1}{m}\textstyle\sum_i (X_i-\overline{X})^2}\\ \mathbf{cov}(X,Y) &= \tfrac{1}{m}\textstyle\sum_i^m (X_i-\overline{X})\cdot(Y_i-\overline{Y})\end{aligned} \tag{2.18}$$

The correlation coefficient only takes values between -1 and 1. The larger this value is, the stronger the relationship between the two variables (ignoring the sign). At the extreme, a correlation of 1 means that both variables are perfectly positively correlated; one variable can be expressed as the product of the other (linear relation). A correlation of 0 means that there is no linear relationship between these two variables.

2.5.4 Mutual Information

The mutual information gives an indication of how much information two variables share. This measure is useful, for instance, when evaluating the information contained in the features about the target output.

The mutual information of two variables X and Y is formally defined as:

$$\mathrm{I}(X;Y)=\mathrm{H}(X)-\mathrm{H}(X|Y) \tag{2.19}$$

where $\mathrm{H}(X)$ and $\mathrm{H}(X|Y)$ represents the entropy of X and its conditional entropy respectively. The entropy and the conditional entropy are formally defined as:

$$\begin{aligned} \mathrm{H}(X) &= -\textstyle\sum_{x \in X} p(x) \cdot \log(p(x))) \\ \mathrm{H}(X|Y) &= -\textstyle\sum_{x \in X} p(x) \cdot \mathrm{H}(Y|X = x) \end{aligned} \tag{2.20}$$

Intuitively, if the base of the logarithm function is chosen to be 2, the entropy $\mathrm{H}(X)$ of a variable X determines how many boolean questions one should ask, on average, in order to determine the value of the variable X. Mutual information therefore measures the information that X and Y share: how knowing one of these variables reduces the uncertainty about the other.

2.6 Summary

This chapter has introduced the machine-learning methodology used to build the models utilised in this thesis. This chapter has also presented the evaluation methodology used to evaluate such models. The next chapter discusses related work.

Chapter 3

Related Work

This chapter presents the research relevant to this thesis. The first section describes commonly used simulation methodologies that focus on speeding up simulation time. Section 3.2 then introduces different performance models that tackle the problem of efficient microarchitectural design space exploration. These models typically reduce the number of simulations needed for exploration. Section 3.3 looks at the compiler optimisation space and how researchers have explored this whilst section 3.4 presents existing solutions for automatic development of compilers that can optimise code effectively. Finally section 3.5 summarises this chapter.

3.1 Simulation Methodologies

As seen in the introduction chapter, one of the main issues in designing new microprocessors is the large overhead induced by excessive simulation time. In this context, shorter simulation time means that many more alternative designs can be evaluated leading to better microprocessors. This section reviews the different techniques that were proposed in the literature to speed up simulation time to make microarchitectural design space exploration affordable.

3.1.1 Statistical Simulation

The idea of statistical simulation consists of a extracting program's characteristics and then using them to generate a synthetic trace. This synthetic trace is constructed so that its characteristics match those of the original program while being much shorter. Then, when design space exploration is performed, this synthetic trace can be executed in place of the original program, reducing simulation time drastically. Technically this approach first runs the program and collects a dynamic execution trace. This trace is then analysed and a new, shorter and simpler one is generated. This synthetic trace is finally simulated symbolically within a modified simulator. The simulation time is therefore shorter since the synthetic trace is much smaller than the original one and no actual computation is required because of the probabilistic nature of the simulation.

The statistics collected from the dynamic execution trace contain both intrinsic program characteristics and locality events. The intrinsic characteristics are architecture-independent and usually include dynamic instruction mix, register dependencies between instructions and basic block size distributions. Conversely the locality events are directly related to the microarchitecture and includes branch misprediction rate or cache misses which are collected for different cache and branch prediction parameters.

One of the major benefits of using statistical simulation for early design space exploration is the large reduction in simulation time. Compared with full cycle-accurate simulation, researchers have typically reported a simulation time reduced by two orders of magnitude with less than 10% error. Oskin *et al.* developed HLS [Oski 00], extended with HLSpower [Rao 02] to allow power modelling. Later Nussbaum and Smith [Nuss 01] extended this work by studying the effects of different models of synthetic trace generation and evaluated how the model tracks the changes in a simple microarchitecture design space. Based on this work, Eechhout *et al.* [Eeck 04] proposed the use of statistical flow graphs to characterise the program's control flow and later explored a processor design space using different search strategies [Eyer 06a].

The major drawback of statistical simulation is that the simulator must be modified in order to incorporate the statistical models. Furthermore, this technique relies on the extraction of program characteristics, making it more difficult to implement than other techniques such as statistical sampling, presented in the next section. Moreover part of the statistics collected depend directly on the underlying architecture. For instance, the locality event related to the caches needs to be captured for all possible cache configurations one wants to explore. This implies that new features need to be collected when changes occur in the microarchitecture. For all these reasons, statistical sampling is usually preferred over statistical simulation. The next section presents this approach.

3.1.2 Statistical Sampling

Pioneered by Conte *et al.* [Cont 96], statistical sampling is a common and widely accepted method that reduces simulation time by only simulating a portion of the program. Clusters of consecutive instructions are identified from the full execution trace of the program. Once selected, only these representative clusters are simulated accurately. Before actually performing a cycle-accurate simulation of each cluster, a warm-up is required to initialise the cache structures and the state of the branch predictor.

Several variations of this technique exist that differ mainly in the way the clusters are selected. In the original work developed by Conte *et al.* [Cont 96], the clusters were obtained at random intervals. More efficient ways of selecting these clusters have been proposed, such as the widely used Smarts [Wund 03] or SimPoint [Sher 02].

Smarts [Wund 03] segments the program into intervals with equal numbers of instructions. For each interval, it simulates, in cycle-accurate mode, a few thousand instructions to warm-up the dynamic structures and then starts recording IPC (Instructions Per Cycle) for a few thousand instructions. The rest of the interval is then simulated in functional mode until the next one. Because Smarts relies on statistical sampling theory, it can estimate the IPC error rate and recommend a higher sampling rate if the error rate is outside the confidence interval.

SimPoint [Sher 02] takes a different approach. The program is first segmented in intervals that contain the same number of instructions, typically 10 million each. Then, each interval is run on the target architecture using functional simulation and basic block execution frequencies are collected, forming basic block vectors. These vectors are then used to cluster the intervals using K-Means and a representative interval for each cluster is selected. Finally only the representative intervals are simulated in cycle-accurate mode (after a warm-up period) and the corresponding results are weighted based on the number of intervals in the cluster.

In a recent study of simulation techniques [Yi 05] for estimating IPC, it was demonstrated that there is in fact little difference between Smarts and SimPoint. Both techniques perform similarly in terms of error (around 3%) even though Smarts is usually more accurate. However, SimPoint exhibits a better speed versus accuracy trade-off and as such is usually prefered if architects are ready to sacrifice a little bit of accuracy for simulation speed. Savings in terms of simulation time are typically between one and two orders of magnitude compared to full program simulation.

Statistical sampling can be used in conjunction with the techniques presented in this thesis. In fact in chapter 4, where a microarchitectural design space exploration is conducted, SimPoint is used to considerably reduce the simulation time of the experiments. SimPoint is simple to use as opposed to techniques such as statistical simulation and, as seen in this section, offers a slightly better advantage in terms of speed whilst maintaining similar accuracy.

3.2 Performance Estimators for Design Space Exploration

The design space considered by architects is often too large to be explored exhaustively. For this reason it is desirable to only simulate a part of the space in order to understand it and infer, in a second step, results for the rest of the space. The previous section has highlighted the different techniques that can be used to reduce simulation time by analysing programs and only simulating representative parts of them. This section now looks at ways of simulating only part of the design space by building estimators that model the total design space. It shows how the use of knowledge extracted from a sample of the space can be used to reduce the total number of simulations needed to thoroughly explore the design space.

3.2.1 Analytic Models

Analytic models have been proposed as a way to conduct design space exploration of some key parameters of the microprocessor. These models are typically built from the observations made by the designer. From these observations, a relationship between the microarchitectural parameters and the performance can be determined. This relationship is expressed in the form of an analytical model; *i.e.* a set of equations from which the parameters leading to the best performance can be derived analytically.

Among many existing prior works, Emma and Davison [Emma 87] modelled an in-order pipelined processor by analysing data dependencies from a program trace. By using different trace reduction techniques that simplify the data dependencies, they were able to accurately predict the performance of the processor.

Noonburg *et al.* [Noon 94] used a model based on probability matrices. Based on the execution trace of a program, they computed the level of parallelism available in the program in terms of data and control parallelism. They also extracted information about the machine parallelism decomposed into branch, fetch and issue parallelism. Once built, their model is able to predict IPC with relatively good accuracy given a program trace.

Michaud *et al.* [Mich 99] took a different approach and built a simple model that focuses on the instruction window and issue mechanisms. Their model expresses ILP (Instruction Level Parallelism) as a function of the window size, allowing true design space exploration. Hartstein *et al.* [Hart 02] developed a similar approach for finding the optimal pipeline depth for a superscalar processor.

More recently Karkhanis and Smith [Kark 04] developed a more complete model that estimates cache miss rate and branch misprediction rate. This model was later extended by the same authors [Eyer 06b] to divide the instruction execution flow into intervals. These intervals are delimited by the different miss events.

All these techniques use the knowledge of an expert designer to derive a model by hand. An in-depth knowledge of the microarchitecture is therefore necessary in order to achieve high accuracy. However it becomes increasingly difficult to adapt these models to the emergence of more complex architectures. For each new feature added to the microarchitecture, new changes to the analytic model are required. This implies new efforts needed from the human expert in order to extend the model. While this approach gives insight into what is happening in the microarchitecture, it is unsuitable for the automated design space exploration of complex microarchitectures.

3.2.2 Benchmark Characterisation

Another way of performing efficient design space exploration consists of analysing the programs and extracting key characteristics from them. Using this characterisation, it is then possible to group similar programs and reduce the total number of simulations.

Saavedra and Smith [Saav 96] extracted dynamic program information to estimate execution time and find similar programs. This information includes the number of instructions executed for each type, such as arithmetic operations or memory operations, the distribution of basic block size and some control flow information. Based on this, the performance of the program can be estimated using some simple analytic models. In addition, this same information can be used to cluster the programs that are similar, so that only the representative benchmarks need to be simulated. This leads to important savings in the total number of simulations required to perform an exploration of the design space.

Recently Eeckhout *et al.* extended this work by adding more program features [Eeck 02]; in particular branch prediction statistics, ILP and cache miss rates. Based on this new set of features, they used a clustering technique, K-Nearest Neighbours, allowing them to predict the performance of any new unseen program [Host 06] for different systems/architectures. However, as chapter 4 will show, this technique does not work when considering microarchitectural changes. It appears that the features used are not sufficiently informative to distinguish between different microarchitectures.

While benchmark characterisation has been used for performance prediction, none of these works referred to in this section have ever demonstrated its accuracy over a realistic microarchitectural design space. Instead they have shown that for a fixed architecture it is possible to build such a model and assume it would work across a design space of microarchitectures. As chapter 4 demonstrates, these techniques actually fail at accurately predicting the design space of new programs. One of the main reasons is that the extraction of a finite set of program features cannot accommodate all possible design spaces. In addition, these features need to be extracted by hand, using a human expert. As a result, any addition or major change to the microarchitecture requires the addition of new features to capture the behaviour of the resulting design space.

3.2.3 Single-Program Microarchitectural Predictor

Another approach to performance estimation consists of building a machine learning model that directly uses the microarchitecture parameters such as pipeline width, register file size or cache sizes as an input. For each program, an individual model is built using training data; *i.e.* simulation runs of the program on a few different microarchitectures sampled from the design space. Once trained, these models are used to make a prediction of any microarchitecture parameter combination.

Many researchers have recently proposed such models. All these models work in a similar way and the only difference between them lies in the type of machine learning technique used. The simpler of these are based on linear regressors [Jose 06a]. More powerful models such as artificial neural networks [Ipek 05, Ipek 06], radial basis functions [Jose 06b] and spline functions [Lee 06, Lee 07a] were also developed, all showing similar accuracy [Lee 07b].

The disadvantage of using such models is their high training cost. For each program one wishes to predict the design space for, a significant number of simulations are required to train the model. Therefore the number of simulations is proportional to the number of programs in the benchmark suite. This is clearly not realistic when the number of programs in the benchmark suite is large. Chapter 4 will present two techniques that can be used to either predict the design space of a new program or predict the design space of a complete benchmark suite using an order of magnitude fewer simulations than state-of-the-art approaches.

3.2.4 Trans-Program Microarchitectural Predictor

To address the large training cost of the predictive models presented in the previous section, researchers have tried to reuse information gained from the run of previous programs. They exploited the fact that programs share similarities. Thus the information from the design space gained from a few programs can be *transferred* to some new unseen ones.

Khan *et al.* [Khan 07] developed a model that uses reactions to characterise programs. With this approach each program is characterised by a set of performance (or reactions) obtained from runs on a few selected architectures. A single model is then built that uses this additional information as an input with the microarchitecture parameters to make new predictions. The work from Khan *et al.* is similar to the work presented in chapter 4, since it characterises programs by looking at their behaviour in the design space. However major differences exist. Firstly they used a one-fit approach where a single model based on an Artificial Neural Network is used. This contrasts with the model developed in this thesis where the problem is decomposed by combining smaller program-specific models to achieve higher accuracy. Secondly, their approach only focuses on predicting the design space of a new program. They do not address the issue of training cost. In comparison the model developed in chapter 4 goes beyond the prediction of a single program and predicts the behaviour of a whole benchmark suite, making it ideal for design space exploration. Finally, by looking carefully at their work, it tends to show that the actual reactions are not very useful. Indeed the prediction accuracy of their model when no reactions are used from the new unseen program is very close to the accuracy obtained when reactions are used. In other words, their model seems to always make the same predictions, independently of the program. This implies that either all the programs behave in exactly the same way in their setup, or that their design space does not contain much variation. Conversely, chapter 4 thoroughly evaluates the design space and the accuracy of the model, demonstrating that it actually works.

3.3 Compiler Optimisation Space Exploration

The previous sections have looked at the related work for microarchitectural design space exploration. This section now considers the compiler side and in particular how to optimise programs by conducting a search of their optimisation spaces. Several techniques were proposed in the literature to efficiently search the optimisation space. A couple of domain-specific techniques are discussed in section 3.3.1. Then the commonly used iterative compilation is reviewed in section 3.3.2 followed by pruning strategies in section 3.3.3. Finally, the use of performance estimators to speed up iterative compilation is presented in section 3.3.4.

3.3.1 Domain-Specific Optimisations

Some specific systems have exploited domain-specific knowledge in order to efficiently compile and optimise code for a given platform. ATLAS and SPIRAL are two example of such systems which consist of a set of low-level functions targeted at some specific domain grouped in a library.

ATLAS [Whal 97] is a self-tuning linear algebra library. This framework recompiles itself depending on the specification of the underlying hardware. In particular, it considers optimisations such as loop tiling and instruction scheduling which enable data prefetching. To discover the optimal values that control these optimisations, ATLAS uses micro benchmarks that stress different aspects of the architecture such as the data cache or floating-point registers. These benchmarks are used to perform a search of the optimal optimisation parameters that are used to generate the optimised version of the library.

SPIRAL [Pusc 05] is another example of a self-tuned library. It automatically generates high-performance code for digital signal processing. It exploits domain-specific knowledge to search the parameter space at compile-time. The domain-specific knowledge comes in the form of notations that express implementation details that are specific to the machine. At compile-time a separate search phase is conducted in order to tune the parameters using iterative compilation, described in section 3.3.2.

These two systems both require domain-specific knowledge and the use of iterative compilation to optimise themselves on the target system. They have to be retuned for each new platform. This contrasts with the machine-learning model developed in chapter 6 where the compiler is built only once and optimises across a range of microarchitectures using just one profile run for any new program.

3.3.2 Iterative Compilation

Iterative compilation, or feedback-directed compilation, optimises a single program on a specific microarchitecture by searching its optimisation space. This technique was pioneered by Bodin *et al.* [Bodi 98] where different loop tiling and unrolling factors were considered for the matrix multiplication problem. Later this work was extended [Kisu 00] to other programs for the same loop transformations of each individual loops in the program. They showed that large speedup in execution time can be found using iterative compilation over standard static techniques that use heuristics to determine the best tile size or unroll factor.

Cooper *et al.* [Coop 99] looked at the phase ordering problem which consists of finding an optimal sequence of code transformations to optimise a given target metric. They were among the first to use a genetic algorithm to search the optimisation space of each program to reduce code size, leading to impressive reductions. Recently the same authors conducted an extensive study of the search space [Alma 04] of the possible sequences of transformations. Different search algorithms were used to search the space showing that many local minima exist, some close to the global minimum and some far away from it. Hence they advocated the use of multiple hill-climber runs.

Vuduc *et al.* [Vudu 04] looked at the problem of optimising a matrix multiplication library by tuning various optimisation parameters using iterative compilation. In particular the search space of the loop tiling size was evaluated. The space was explored randomly and a statistical stopping criterion was used to determine when the search should stop. A confidence value is also given that estimates how far from the real optimum the result is.

Haneda *et al.* [Hane 05] used a statistical approach to automatically determine the best compiler settings for a given application. The notion of *Orthogonal Array* was used where each column represents a compiler setting and each row an experiment to be performed. The value in each cell indicates whether the corresponding flag is enabled or disabled. Using the rows of this array, the application is compiled with the specific flag settings corresponding to each row and the execution times are recorded. Then a statistical technique is used to infer which compiler flags are beneficial and the values for these important flags are fixed for subsequent runs. The procedure is repeated until the values of all flags have been fixed. The authors presented their results and compared them to the default flag settings of the compiler. Although their approach shows some improvement over the default compiler settings, the number of compilations and runs needed by their technique is large. In addition they did not compare their technique with the most basic search strategy; random search.

Finally Pan and Eigenmann [Pan 06] evaluated the two previous techniques with their own algorithm to find the best compiler settings for a given program. Their algorithm searches the space by iteratively eliminating settings with the most negative effect from the search space. They showed, within their experimental setup, that the three techniques achieve in fact a similar speedup of 1.07 on average over the default compiler settings. However the number of executions needed to search the space is consistently lower than for the two other approaches.

All the techniques mentioned in this section show a requirement for a search of the optimisation space for each new program one wishes to compile. This contrasts drastically with techniques that use prior knowledge to *predict* the best set of compiler settings without actually searching the optimisation space of the program. The next section looks at extended search strategies that make use of prior knowledge to speed up the search.

3.3.3 Search Space Pruning

A few researchers have extended iterative compilation by using different pruning techniques. The basic idea is that not all of the optimisations need to be integrated in the search space since some transformations or sequences of transformations might have very little impact.

Triantafyllis *et al.* [Tria 03] explored the space of compiler settings iteratively. They first find, at compiler-construction time, a small set of promising compiler settings that perform well on a given set of code segments. A search tree is then built from these settings and traversed to find good combinations for the frequently executed code segments for a new application, drastically limiting the space. In addition, the authors also developed a performance estimator that uses a simple analytic model. They showed that the results of the search are similar in cases where either the estimator is used or real runs are performed. They showed that, on average, an exhaustive search of the space leads to a speedup of 1.10 over the default compiler settings for execution time. Their exploration methodology achieves a speedup of 1.05 on average; 50% of the total available.

Other researchers looked at the phase ordering problem at the function level, trying to find the best sequences of transformations for each function of a given program. Kulkarni *et al.* [Kulk 04] used their previously developed VISTA compiler infrastructure [Zhao 02] to search effective optimisation phases. By analysing the code produced by different transformations, the authors detect transformation sequences that do not change the code, *i.e.* dormant phases. Sequences that do transform the code, *i.e.* active phases, are represented in the form of a tree. This tree can be later used during search to avoid unnecessary compilations. Using this pruning technique coupled with a genetic algorithm, they conducted a search for the best phase order at a function level leading to the best trade-off in terms of speed and code size. They found that over 84% of the executions can be avoided by identifying these cases where some phases are dormant or equivalent code is produced.

Pruning the optimisation space can dramatically speed up iterative compilation. However, it still suffers from the fact that a search is required for each new application one wants to compile. The next section reviews prior works that focused on reducing the evaluation cost of one compilation by building performance estimators.

3.3.4 Performance Estimation

Performance estimators are an efficient way of reducing the cost associated with iterative compilation. A typical search of the optimisation space consists of repeating the cycle of compilation followed by a run until a good solution is found. A performance estimation can be used to replace the run of the newly compiled application and therefore save time; this estimation is cheap compared to the time required to run the program.

Analytic Models Zhao *et al.* developed an approach based on an analytic model named FPO [Zhao 03] to estimate the impact that different loop transformations can have. They used their model to decide whether to apply three transformations for each loop in their programs; loop interchange, loop tiling and loop reversal. The model makes the right decision more than 80% of the time. The authors extended this work by creating more models for different optimisations such as partial redundancy elimination and loop invariant code motion [Zhao 05]. However when compared with a simple heuristic that controls when to apply these optimisations, their model achieves only marginal improvement. Moreover the practicality of such an approach might be questioned since for each optimisation a new model must be developed by hand.

As already mentioned in section 3.3.3, Triantafyllis *et al.* [Tria 03] also used an analytic model to reduce the required time to evaluate different compiler optimisations for different code segments. They showed that when exploring the optimisation space using their estimator, the same result is achieved as when performing real runs. In fact these analytic models are really useful for searching the optimisation space, even though they provide only a rough estimate of the real execution time. Indeed the key element is to be able to distinguish between good and bad optimisations rather than knowing the exact execution time when conducting an exploration of the space.

Yotov *et al.* [Yoto 03] investigated a model-driven approach for the ATLAS library and compared it with iterative compilation for finding the optimal compiler settings. They developed an analytic model that simply uses the machine description (cache size for instance) to compute the optimal parameters of the optimisations. Using this model, the time needed to generate the library, optimisation and code generation, is typically reduced by 30% to 70% percent in total. For one, out of the three platforms tested, a difference in performance of 20% is observed between the model and the use of micro benchmarks, showing that this approach can bring important savings in terms of compilation time.

A few researchers tried to predict the performance that an optimisation would have by looking at the compiled code only. This approach has the potential to reduce the time required to explore optimisation spaces. To overcome the high costs associated with iterative compilation, Cooper *et al.* developed ACME [Coop 05] which uses the concept of virtual execution. Instead of actually compiling and running each transformation sequence, a single profile run is performed. Based on this run, an execution frequency for each basic block can be extracted which is used to count the number of times each instruction will be executed. An estimation of the actual execution time is made by simply summing up all these numbers. This sum can then be updated accordingly for any sequence of transformations that adds or removes instructions from a basic block. Using this technique the authors showed that the time spent performing a search can be reduced by a few factors while maintaining the same accuracy. However this approach is limited since only simple data-flow transformations can be modelled in this way; it does not take into account various effects such as branch predication or cache misses.

Empirical Models A different approach taken by Cavazos *et al.* [Cava 06a] uses an empirical model to make predictions about the behaviour of optimisation sequences. The model is first trained offline using a set of training programs. Then, based on characteristics extracted from the new program one wants to compile, the *reactions* as the authors named them, the model makes performance predictions for any given sequence of transformations. The program is characterised automatically using a set of responses; *i.e.* a few selected transformations are applied to the new program and the performance is recorded, leading to a feature vector used to characterise the program. This way of characterising the program's behaviour offers the advantage of being independent of the architecture or optimisation used. This is in fact an essential element of the work presented in chapter 4 for the problem of microarchitectural design space exploration.

This concept of using empirical models to estimate the performance of sequence of transformations was further used by Dubach *et al.* [Duba 07b]. The authors used code features extracted at the source code level to characterise the effects of code transformations. Using the features extracted before and after the sequence of transformations is applied, the model predicts accurately the performance that the sequence of transformations would achieve, should the program be run. The authors demonstrated that this model can be used to predict the performance of new transformations not seen during training, since the only input used by the model are the actual code features.

Vaswani *et al.* [Vasw 07] modelled the co-design space of compiler optimisation settings and microarchitecture design parameters by using different techniques such as linear regression or radial basis function. Their model takes as an input the microarchitectural configuration and the desired optimisation flags and produces a prediction for the execution time. The authors reported error rates between 5% and 10% for the predictions. However, as it will be shown later in chapter 5, this model fails to capture interactions between compiler optimisations and the microarchitecture. Consequently its use is limited and it cannot be used to develop an optimising compiler that works across the architecture space.

3.4 Automatic Compiler Construction

The previous section has looked at the different compilation techniques that can be used in order to extract the maximum performance available in the compiler optimisation space by searching this space. While these techniques have shown that large performance improvement can be achieved, none of them have actually managed to automatically develop an optimising compiler. They all require a search of the optimisation space. Section 3.4.2 will look at existing work in the area of automatic optimising compiler generation but first a review of retargatable or portable compilers is conducted.

3.4.1 Portable Compilation

The integration of compiler and architecture development is not new and has been the focus of prior research over the last 10 years. Frameworks such as Buildabong [Fisc 01], Trimaran [Trimaran 00] or Pico [Abra 00] allow automatic exploration of both compiler and architecture spaces. The compiler and the simulator live side by side and are often tightly coupled within these frameworks, allowing great flexibility in terms of space exploration.

Other researchers have focused on creating portable compilers such as VPO [Beni 88] or LLVM [Latt 04]. These compilers maintain independence between the optimisation passes and the target architecture. However, these infrastructures focus purely on portability from an engineering point of view: developing tools and optimisations that can be reused across many architectures. This is fundamentally different from the work that is presented in this thesis where the compiler automatically learns how to tune the application for any architecture. The next section will introduce the related work in this area.

3.4.2 Machine-Learning Optimising Compilation

The term *machine-learning optimising compiler* can be used to describe a compiler whose optimisation strategy has been learnt automatically and is able to adapt to any new program one wants to compile. This is drastically different from typical approaches that either build a fixed optimisation strategy by hand or perform a search of the optimisation space for every new program, as seen in previous sections.

Moss *et al.* were among the first researchers to apply machine learning techniques inside a compiler. Their first work [Moss 98] involved scheduling local instructions within a basic block using different supervised learning techniques. They showed that it was possible to beat the scheduling heuristics present in a production compiler using an entirely automated process. Moreover their technique was able to achieve almost the same level of performance compared to one of the best instruction schedulers available at that time; the DEC heuristic scheduler.

Monsifrot *et al.* used use machine-learning in order to automatically build compiler heuristics [Mons 02]. In their paper, they showed how the heuristic that controls the loop unrolling optimisation can be automatically created using decision trees, a simple classification technique. Compared with the default heuristic of the fortran compiler used in their experiments, the automatically generated heuristic performs better on average. This is a rather strong result since they were able to achieve this automatically, as opposed to the default heuristic that was certainly tuned by compiler experts over a long period of time.

Stephenson *et al.* investigated the use of meta optimisations [Step 03] by tuning the compiler heuristics using genetic algorithms. In particular they looked at the heuristics that control hyperblock formation, register allocation and data prefetching. Because they conducted their experiments on a per program basis, where the heuristic was tuned independently for each program, this is more like an optimisation space exploration rather than heuristic tuning. However, they did evaluate their results with cross-validation, where the heuristic was tuned on a set of training programs and tested on a different set. Unfortunately, it appears that the results obtained are not conclusive, despite claims from the authors. For instance, the approach using cross-validation could only achieve a speedup of 1.01, out of a possible 1.36 speedup, in the case of data prefetching. This clearly shows that their technique failed at generating a compiler heuristic that works for any new program encountered.

In a more recent piece of work, the same authors looked at tuning the unrolling factor using supervised classification techniques [Step 05] such as K-Nearest Neighbours and SVM. However only a modest improvement over the default heuristic was reported.

In addition to their work on iterative compilation for the matrix multiplication optimisation problem, Vuduc *et al.* [Vudu 04] developed a statistical approach that decides at runtime which implementation is the best suitable for a given matrix input size. They compared different models such as linear regression or support vector machines showing that this latter model outperforms all others and makes the right choice 88% of the time.

Long and O'Boyle developed an instance-based learning model [Long 04] for loop optimisations of Java programs. Once trained on the target platform, their model predicts, given some program features extracted statically, how to transform the loops in the program in order to get maximum performance on that particular platform. They showed that their model achieves most of the speedup available when compared to an exhaustive exploration of the optimisation space.

Cavazos and Moss [Cava 04] also applied machine learning techniques to the problem of deciding whether to perform instruction scheduling for a given basic block within a virtual machine; Jikes RVM [Alpe 99]. Instruction scheduling is a time-consuming compiler optimisation that tries to reorder machine instructions in order to reduce execution time. Since compilation time is part of the application execution time within the context of a virtual machine, significant reductions in execution time can be achieved by applying these costly optimisations only when the resulting code is likely to lead to good performance. For this purpose the authors of this work applied a supervised technique to learn whether to apply instruction scheduling, using features extracted from the basic blocks. Later Cavazos and O'Boyle extended this concept and applied logistic regression [Cava 06b] to the problem of finding the best set of optimisations to apply for each method within the Jikes RVM. They used features extracted from the bytecode to infer which optimisations to apply to each method.

Agakov *et al.* [Agak 06] introduced a machine-learning technique to focus iterative compilation for the search of optimal transformation sequences for small kernel-like applications for embedded systems. Using features extracted from the program, the model is able to predict in which area of the space the optimum lies. Then a standard iterative compilation is performed on these areas. By comparing with exhaustive search, they showed that more than 85% of the total speedup available can be reached with as few as 10 iterations, whereas random search needs an order of magnitude more iterations to reach the same level of performance.

This work was later extended in [Cava 07] where the model uses performance counters instead of program features as a way to characterise new programs. They evaluated their approach using benchmark suites such as SPEC 2000 and MiBench that contain fairly large applications. In particular they showed that performance counters are superior to static code features. The reason lies in the fact that code features are difficult to define for large applications; they tend to characterise local code well but fail at globally characterising an entire program. The major reason being that aggregating local features is a difficult task. For this reason, chapter 6 will make use of performance counters to develop a new machine-learning compiler that can make one-shot predictions of the best compiler settings for any new program or architecture.

To summarise, a few researchers have looked at ways to automatically generate an optimising compiler. Some took the approach of tuning or creating heuristics while others have focused on using an external model, *i.e.* a machine-learning model, to drive the compilation process. However, all these prior works have focused on deriving an efficient optimisation strategy for a fixed architecture. This contrasts with the work presented in chapter 6 where an optimising compiler is developed that can adapt to a full microarchitectural space.

3.5 Summary

This chapter has presented prior work related to microprocessor design, compiler optimisation space exploration and portable optimising compilers. The work in the field of microprocessor design has mostly been limited to simulation methodology and the use of per-program models. In contrast, the work presented in chapter 4 of this thesis develops performance models that exploit knowledge across programs.

Prior work on compiler optimisation exploration has mainly focused on searching the optimisation space of a program. As seen throughout this chapter, many search strategies were proposed and some models developed to focus the search in promising areas of the space or directly predict the set of optimisations to apply. However all these techniques were targeted at a specific program or architecture. In contrast the work presented in chapter 5 focuses on exploring the compiler optimisation space across a range of programs and microarchitectures. Chapter 6 develops a truly portable optimising compiler. This compiler uses a model that predicts how to compile any new unseen program on any new unseen microarchitecture in order to get maximum performance.

Chapter 4

Exploring and Predicting the Microarchitectural Design Space

4.1 Introduction

The design of new microprocessors is often associated with long-running detailed simulations. Because of the large number of parameters that can be tuned in a typical general purpose processor, the space of all possible designs, *i.e.* the design space, quickly becomes impractical to explore. For this reason performance predictors have been recently proposed in the literature [Ipek 06, Jose 06a, Lee 06, Lee 07a, Lee 07b] as a means to quickly and accurately model the design space of each individual program. While this is clearly a good step towards efficient design space exploration, it is still not sufficient when a large number of programs are considered.

To overcome this problem, researchers have looked at ways of reducing the number of benchmarks by analysing program similarities [Eeck 02, Host 06]. The benchmark suites are reduced by maintaining a subset of programs that differ significantly from each other, thus reducing the time spent exploring the design space. The advantage of this technique is that the features used to characterise the programs are independent of the architecture space under consideration, therefore they only need to be extracted once. However, as demonstrated in this chapter, this technique is not accurate enough to enable efficient design space exploration.

This chapter presents a novel approach to design space exploration that *transfers* information across programs. It is organised as followed; first the simulation environment and the benchmarks are described in section 4.2. Then the resulting design space is analysed in detail in section 4.3 where it is shown that some programs behave similarly. Using this observation, section 4.4 presents a model that reuses information from previously seen programs to predict the design space of new ones. Later, in section 4.5, this model is extended for the prediction of the average behaviour of a benchmark suite. Section 4.6 then shows a practical usage scenario where a search of the design space is performed. The superiority of this model is then demonstrated against a state-of-the-art approach that uses microarchitectural-independent program features in section 4.7. Finally section 4.8 concludes this chapter.

4.2 Experimental Setup

This section describes the experimental methodology used throughout this chapter. In particular it details the simulation environment, the benchmark suites used and the microarchitectural design space.

4.2.1 Simulation Environment

SimpleScalar [Burg 97] is a well established out-of-order superscalar processor simulator in the research community. It was chosen in this chapter to conduct an exploration of the design space of a typical general purpose processor, using the *Alpha* instruction set. While this simulator does not correspond directly to any real machine, it is widely used in the research community allowing meaningful comparison with existing work. The specific version of SimpleScalar used (v2.0) is based on Wattch [Broo 00] in order to obtain power consumption estimates. In addition, Cacti (v4.2) [Tarj 06] was used to accurately model the energy and access latencies of the microarchitectural components to make the simulations as realistic as possible. So, for example, as the size of the data cache is increased, so is the static energy it consumes each cycle, the dynamic energy it consumes on each access and the number of cycles it takes to access it.

When a program is run, the simulator reports both the number of cycles and the energy consumed (in nJ) gained from Cacti and Wattch. Those values are used to evaluate program performance and power consumption. In addition to those metrics, architects are typically interested by the tradeoff between energy and cycle time (delay). For this purpose, the energy-delay (ED) and energy-delay-squared (EDD) products were also evaluated. These metrics defined as:

$$ED = Energy \cdot Delay \tag{4.1}$$

$$EDD = Energy \cdot Delay \cdot Delay \tag{4.2}$$

reflect the efficiency of the processor configuration, the lower the value the better.

Program	Input	Simulated instr.	Total instr.
ammp	*NA*	130m	319b
applu	*NA*	230m	224b
apsi	*NA*	290m	348b
art	1	110m	42b
bzip2	source	200m	109b
crafty	*NA*	200m	192b
eon	cook	180m	81b
equake	*NA*	180m	132b
facerec	*NA*	200m	211b
fma3d	*NA*	200m	268b
galgel	*NA*	190m	409b
gap	*NA*	210m	269b
gcc	166	270m	47b
gzip	graphic	160m	103b
lucas	*NA*	220m	142b
mcf	*NA*	160m	62b
mesa	*NA*	160m	282b
mgrid	*NA*	150m	419b
parser	*NA*	200m	547b
perlbmk	704	170m	67b
sixtrack	*NA*	50m	471b
swim	*NA*	200m	226b
twolf	*NA*	190m	346b
vortex	lendian1	220m	119b
vpr	route	150m	84b
wupwise	*NA*	210m	350b

Table 4.1: The 26 SPEC CPU 2000 benchmarks with their corresponding input used, the total number of instructions simulated with SimPoint and the actual total number of instructions present in each benchmark.

4.2.2 Benchmarks

When architects design a new microprocessor, they wish to evaluate the performance of many different possible designs or implementations. This evaluation must be conducted by testing many different workloads on the architecture. For that purpose, well established benchmark suites exist that try to be as representative as possible of a domain of applications that will be run on the final developed processor. Since this chapter deals with the design of general purpose processors, the SPEC CPU 2000 benchmark suite [Henn 00] is a natural choice. This is a widely used and standardised benchmark suite supported by industry.

The entire SPEC CPU 2000 benchmark suite was compiled for the *Alpha* target with the highest optimisation level and run using the *reference* input set. Since it was not manageable to run those programs in their entirety, SimPoint [Sher 02] was used to reduce the number of simulated instructions to a reasonable amount whilst maintaining high accuracy. An interval size of 10 million instructions and a maximum of 30 clusters were selected per program. Experiments were run warming the cache and branch predictor for 10 million instructions before performing the actual detailed simulation. Table 4.1 shows the list of programs from SPEC CPU 2000 as well as the reference input used. The total number of instructions simulated and present in each benchmark is also shown.

Program (cont.)	**Input**	**Sim. instr .**
basicmath	small	57m
bitcnts	small	42m
qsort	small	148m
susan_c	large	1.5m
susan_e	large	3m
susan_s	small	28m
cjpeg	small	36m
djpeg	large	10m
lame	small	146m
madplay	small	43m
tiff2bw	small	67m
tiff2rgba	large	61m
tiffdither	small	431m
tiffmedian	small	215m
lout	small	107m
dijkstra	small	75m
patricia	small	91m
ispell	small	10m
say	small	48m
search	large	0.2m
bf_d	small	42m
bf_e	small	42m
pgp	*NA*	1.4m
pgp_sa	*NA*	93m
rijndael_d	small	34m
rijndael_e	small	32m
sha	small	15m
rawcaudio	small	37m
rawdaudio	small	29m
crc	small	34m
fft	small	21m
fft_i	small	40m
toast	small	51m
untoast	small	17m

Table 4.2: The 34 MiBench benchmarks used with their corresponding chosen input size and the total number of instructions executed.

In addition to SPEC CPU 2000, the MiBench benchmark suite [Guth 01] was also included for the later sections of this chapter. MiBench is a multimedia benchmark suite with many of these applications typically running on desktop machines equipped with general purpose processors.

Since the MiBench programs are much smaller than the ones from SPEC CPU 2000, all of them were run in their entirety except for *ghostscript* which would not compile correctly. Here too, the programs were compiled for the *Alpha* target with the highest optimisation level. For each program, the input sizes were chosen in order to be as close as possible to 100 million instructions executed per program without going beyond this limit whenever possible. This was done to ensure enough instructions were being executed while bounding simulation time to a reasonable amount. Table 4.2 shows the programs used from MiBench and the total number of instructions simulated for each benchmark. For MiBench the number of instructions simulated and present in the benchmark is the same since the programs were run in their entirety, without using SimPoint.

4.2.3 Microarchitecture Design Space

The microarchitectural design space considered contains 63 billion different configurations, created by varying 13 different parameters within the SimpleScalar simulator. These are listed in table 4.3. They are similar to those other researchers investigated [Ipek 06, Lee 06], allowing

Parameter	**Value Range**	**Num**	**Baseline**
Width	$2,4,6,8$	4	4
ROB size	$32 \to 160 : 8+$	17	96
IQ size	$8 \to 80 : 8+$	10	32
LSQ size	$8 \to 80 : 8+$	10	48
RF sizes	$40 \to 160 : 8+$	16	96
RF rd ports	$2 \to 16 : 2+$	8	8
RF wr ports	$1 \to 8 : 1+$	8	4
Gshare size	1K $\to$ 32K : $2*$	6	16K
BTB size	1K, 2K, 4K	3	4K
Branches allowed	$8,16,24,32$	4	16
L1 Icache size	8K $\to$ 128K : $2*$	5	32K
L1 Dcache size	8K $\to$128K : $2*$	5	32K
L2 Ucache size	256K $\to$ 4M : $2*$	5	2M
Total		63bn	

Table 4.3: Microarchitectural design parameters that were varied with their range, steps and the number of different values they can take. The step value can be either an increment (+) or a product (*). Also included is the baseline configuration.

meaningful comparisons with previous work. The left-hand column describes the parameters and the second column gives the range of values each parameter can take. Also shown is the step size between the minimum and maximum values. The third column enumerates the number of different values for each parameter.

A baseline configuration, shown in the fourth column of table 4.3 was chosen. This baseline is in fact similar to the Intel Core microarchitecture [Inte 07, Sand 07]. It is a balanced microarchitectural configuration, allowing four instructions to dispatch and issue each cycle, with modestly-sized instruction and data caches.

It was decided to keep some parameters related to cache structure and branch prediction mechanism constant, as shown in table 4.4(a). Parameters such as associativity and block size are expected to have little impact on performance in regard to the other parameters varied. Also the latencies of the different functional units were kept constant to simplify the experimental setup. However should these additional parameters be integrated in a processor study, it is expected that the techniques presented later in this chapter would still be applicable.

The number of functional units were varied with the width of the pipeline as shown in table 4.4(b). So, for a 4-way machine for instance, four integer ALUs, two integer multipliers, two floating point ALUs and one floating point multiplier/divider were used.

Parameter	Configuration
BTB associativity	4-way
L1 Icache	32B block size, 4-way
L1 Dcache	32B block size, 4-way
L2 Ucache	64B block size, 8-way
FU latencies	IntALU 1 cycle, IntMul 3 cycles, FPALU 2 cycles, FPMul/Div 4/12 cycles

(a) Constant

Parameter	Number			
Machine width	2	4	6	8
IntALUs	2	4	5	6
IntMuls	1	2	2	3
FPALUs	1	2	3	4
FPMulDiv	1	1	2	2

(b) Related to width

Table 4.4: Microarchitectural design parameters that were not explicitly varied, either remaining constant or varying according to the width of the machine.

Although the total design space studied has 63 billion different configurations, some of them do not make architectural sense within this setup. For example, the reorder buffer should not be smaller than the issue queue or load/store queue. These configurations were thus removed resulting in a total design space of 18 billion points. In reality other constraints might need to be modelled and the space might be further pruned based on the particularity of the architecture under consideration, but this is outside the scope of this work.

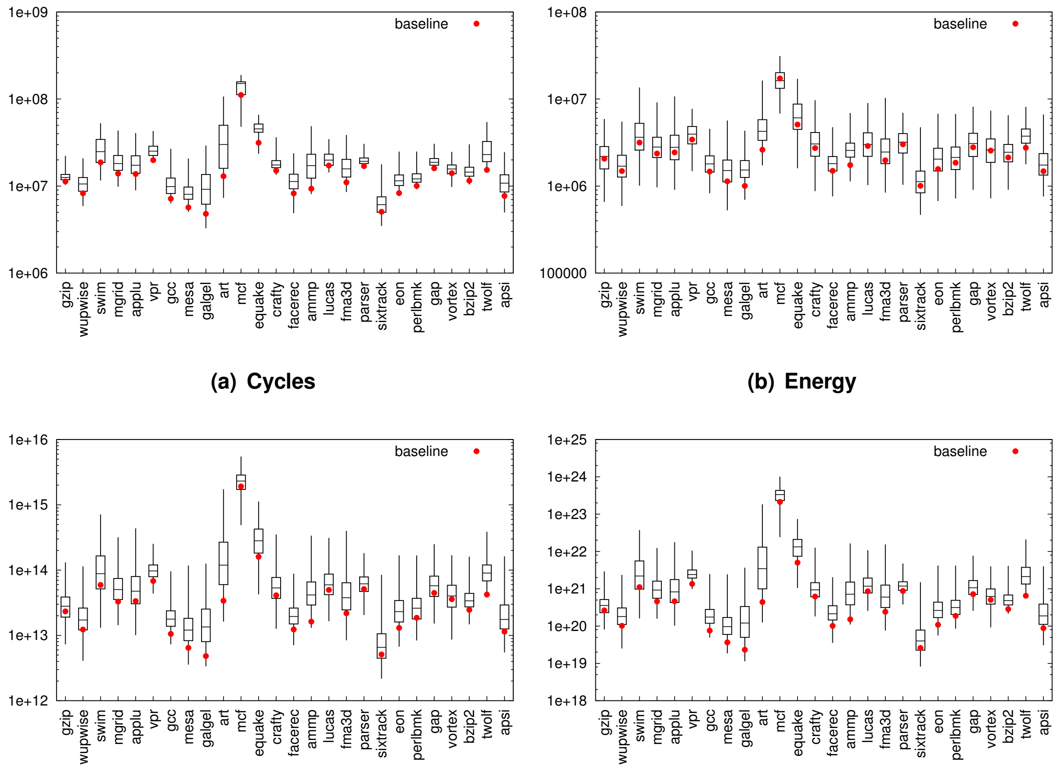
4.2.4 Evaluation methodology

Since there are 18 billion design points in the design space, it is infeasible to simulate all these points. Therefore, a subspace of 3000 points was selected using uniform random sampling. Each benchmark was simulated on these sampled architectural configurations leading to a total of 180,000 unique simulations (60 programs × 3000 designs).

All predictors were trained using these randomly selected configurations from the design space and validated using cross-validation. Since the selection of the training simulations was performed randomly, each experiment was repeated 20 times, using different random simulations each time.

4.3 Analysis of the Design Space : SPEC CPU 2000

This section analyses the design space for the SPEC CPU 2000 benchmark suite. This analysis is performed on the sampled space containing 3000 design points for each program of SPEC CPU 2000. The purposes of this analysis are multiple. First, it presents the individual spaces of each program and shows that they are not trivial (*i.e.* flat). Secondly, it shows that similarities do exist among the programs, although they differ in many respects. This is a key insight that



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.1: Characteristics of the design space for the SPEC CPU 2000 programs. Cycles, energy, ED and EDD are shown for a normalised execution of 10 million instructions. Each graph shows the median, quartiles for 25% and 75%, minimum and maximum values for each benchmark, using a logarithmic y-axis. Also shown is the performance of the baseline architecture for each program.

will be used later by the techniques presented in sections 4.4 and 4.5. Finally, it shows that the behaviour of the benchmark suite across the design space exhibits some variation.

This contrasts with many prior works [Ipek 06, Jose 06a, Lee 07a] in the field of performance prediction that simply showed prediction accuracy without really showing any characteristic of the space they predicted for. It is indeed straightforward to build predictors for “flat” spaces. This section, however, demonstrates that the space considered in this work is not trivial.

4.3.1 Variation in the Design Space per Program

Figure 4.1 shows some important statistics of the space on a per-program basis for cycles, energy, ED and EDD. For each benchmark, the maximum value, the 75% quartile, the median, the 25% quartile and the minimum are shown. The choice of reporting the quartile and the medium values (instead of the standard deviation and the mean) were motivated by the fact that these statistics are more robust than their counterpart when dealing with a sample of a real population. In addition, the baseline configuration is also shown on the same figure.

From these graphs it is clear that there are large differences in terms of program behaviour. For instance the median energy value (figure 4.1(b)) of program *mcf* is an order of magnitude higher than the one from *sixtrack*. Since all the metrics reported were normalised for 10 million instructions for each program, the difference observed between the programs are due to intrinsic program differences (some programs might have better data locality or more parallelism available than others for instance and hence they might make a better usage of the resources available).

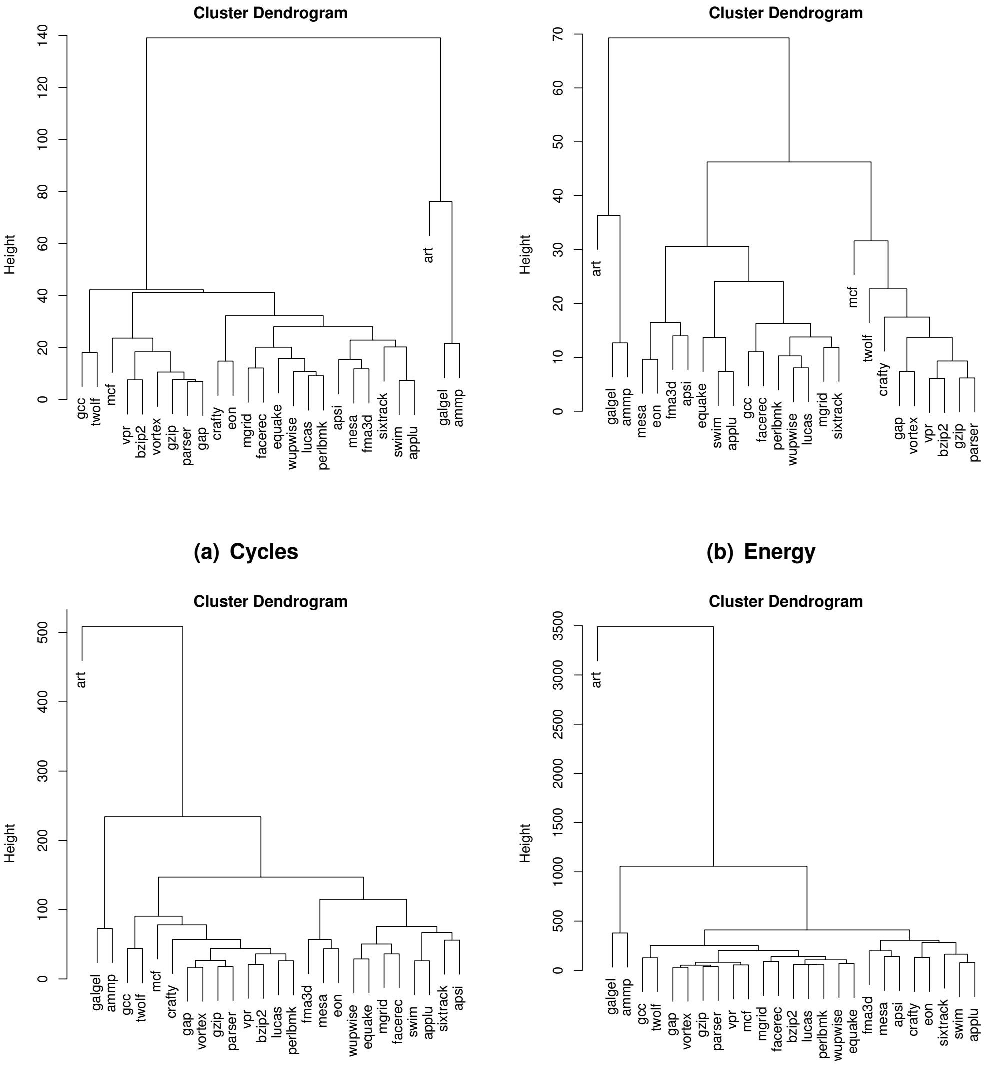
Looking at each program individually, significant differences in terms of variability of their space is observed. For instance when considering cycles (figure 4.1(a)) the program *gzip* has a very low variability and the difference between the slowest and the fastest configuration is only a factor two. On the other hand, programs such as *art* have a big variability; the difference between the minimum and maximum is more than an order of magnitude. For such programs, the choice of microarchitecture clearly makes a difference and this shows the importance of selecting the right design.

Finally it is interesting to notice the behaviour of the different programs on the baseline architecture. For instance for ED (figure 4.1(c)), it can be seen that the baseline configuration is a very good choice for program *twolf* while for *swim* for instance, it is three times worse than the best one. This means that a configuration that is good for one program is not necessarily good for another program.

4.3.2 Program Similarities

Since the main part of this chapter is devoted to learning across programs, this section shows that there are similarities between the programs of SPEC CPU 2000. The similarity between programs is expressed using the distance between their design spaces in terms of responses: the values directly extracted from the output space. To measure this distance, a vector is built for each program containing the values of the target metric (cycles, energy, ED or EDD) for every design point of the sampled space (3000 samples). The Euclidean distance between these vectors, containing each 3000 values, is then calculated and serves as the distance measure. This differs from prior work that measured program similarities [Phan 05] using dynamic features, as opposed to directly using the responses surface of the design space.

Because it is not desirable to show the distance between each possible pair of programs, a hierarchical clustering technique was applied. This technique offers a very convenient way



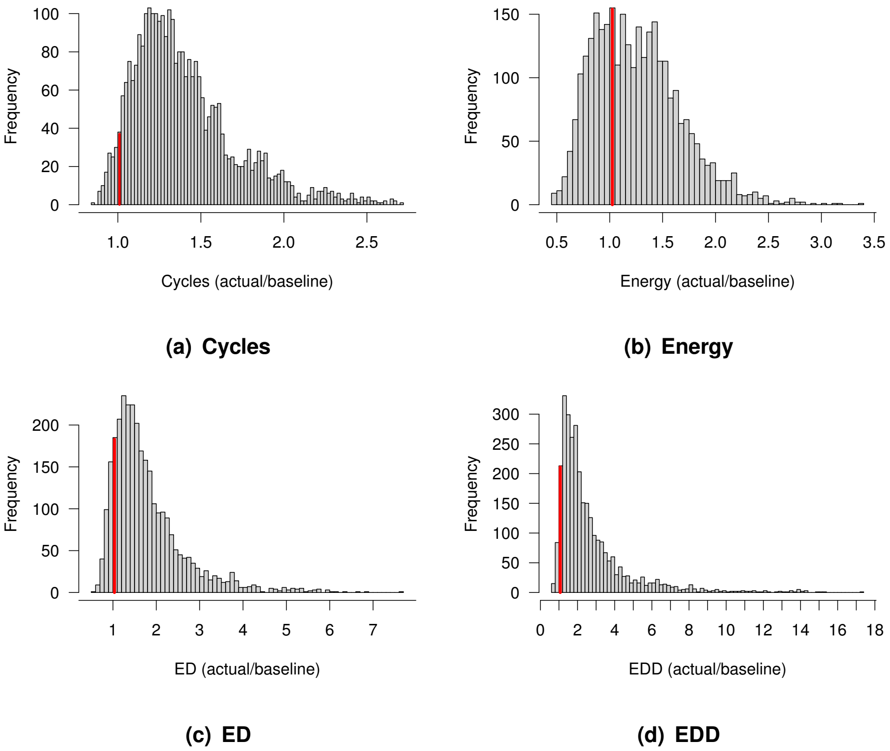
(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.2: Result of the hierarchical clustering using the Euclidean distance measure. The average distance between the design space of any two group of programs can be determined by looking at the height of the branch that connects them. For instance for cycles, there is an average distance of 80 between *art* and the two programs *galgel* and *ammp*, that differ themselves by a distance of 20.



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.3: Distribution of the design space for cycles, energy, ED and EDD. The baseline configuration lies within the top 10% of the space for cycles, ED and EDD, and within the top 20% of the space for energy.

of representing program similarities and has already been applied by other researchers in the field [Josh 06]. Figure 4.2 shows for each metric a dendrogram resulting from the hierarchical clustering. The height gives the average distance between the programs contained in two branches. For instance for ED, the program *art* is on average at a distance of 500 to all the others. The higher the separation, the less similar the programs are.

For all the metrics, it can be observed that program *art* is very different from the others. Furthermore it can be seen that *mcf* is significantly different from the others for energy. It is thus expected that these programs will be difficult to predict because of their unique behaviour.

4.3.3 Average Behaviour

Having looked at the behaviour of each of the SPEC CPU 2000 programs individually, this section looks at the average behaviour of the full benchmark suite. To avoid giving more weight to some programs in the calculation of the average, the performance of each program was normalised against the performance obtained on the baseline architecture. Hence the geometric mean was used to compute the average performance, since it deals with ratios.

As can be seen in figure 4.3, most of the configurations achieved between one and two times the performance of the baseline architecture (with value 1.0). Those with a value less than 1.0 being better than the baseline. It also shows that there are more configurations worse than the median than there are better ones (the histogram ends with a long tail on the right). Considering the design space for energy, it contains many more configurations close to the minimum than for the other metrics. Therefore, it is expected that it will be statistically easier to find a value in the space close to the minimum.

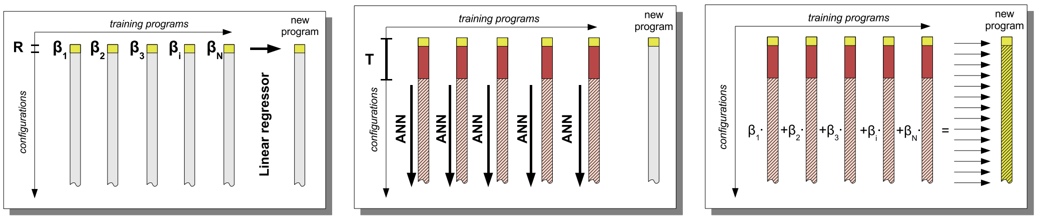
In addition, the baseline configuration lies comfortably within the top 10% of this space for cycles, ED and EDD, and within the top 20% of this space for energy. This verifies that this baseline configuration is a well balanced design point and that finding a better design point is not trivial.

4.4 Architecture-centric Predictor

In this section, a model named the "architecture-centric predictor" is presented that learns across programs. This contrasts with prior approaches [Ipek 06, Jose 06b, Lee 06] that only considered each program independently. This model predicts the performance of any new unseen program for any point in the design space. The advantage of this model over its predecessors is that it requires one order of magnitude less training simulations, while achieving the same prediction accuracy.

This architecture-centric model was built based on the observation that some similarities exist between the different program spaces as seen in the analysis of SPEC CPU 2000 (section 4.3.2). These similarities can be exploited to make accurate predictions for the design space of any new program. This section presents a model capable of efficiently combining the microarchitectural design spaces of existing programs to make useful predictions for any new, unseen program.

The idea of predicting performance for one program from other similar ones is not new and was already exploited by Hoste *et al.* [Host 06]. However, as it is shown later in this chapter, this approach simply fails at predicting the microarchitectural space with sufficient accuracy. In fact Hoste *et al.* only applied their technique for ranking architectures at a system level. Their reason was that ranking is an easier task and does not required very accurate predictions. This ranking mechanism, however, has its limitation when applied to the design space exploration of microarchitectural configurations. Indeed, architects want to know more than just a ranking of the different possible designs. Ideally, they would like to have access to



(a) Linear mapping

(b) Single-program predictors

(c) New program predictions

Figure 4.4: Overview of the architecture-centric model. First, a linear combination is learnt from the training programs to the new program using **R** responses (a). Then the space of each training program is predicted using Artificial Neural Networks (ANNs) built using **T** samples (b). Finally the entire space of the new program can be predicted using the linear combination of the training program (c).

the complete design space, allowing them to see what are the specific impacts of the different architectural parameters on the target metrics. For these reasons a new approach is needed and is presented in the following sections.

4.4.1 Overview

The technique presented in this section consists of combining linearly the design spaces of training programs in order to predict the design space for any new program. This combination corresponds in fact to a weighted sum of the design spaces of the training programs. A linear regressor is used to determine the optimal weights β_i using only a small number of runs **R**, called **responses**, from the new program as seen in figure 4.4(a). These responses correspond in fact to the values of the target metric extracted from **R** runs of the program on a few selected microarchitectures. Once the weights β_i have been determined using the **R** responses, it is possible to predict the design space of the new program by simply applying a weighted sum to the corresponding design points of the training programs.

However, if one wants to predict any design points for the new program, the whole design space of each training programs must be known, which is not realistic. To overcome this issue, single program predictors are built using Artificial Neural Networks (ANNs) as shown in figure 4.4(b). Each program in the training set is simulated on **T** microarchitectures selected randomly. Once this data is gathered, the design space of each training program can be predicted using the newly built ANNs.

Finally, to predict any microarchitectural design point of the new program, the ANNs of each training program are used to derive predictions. Then these predictions are summed up

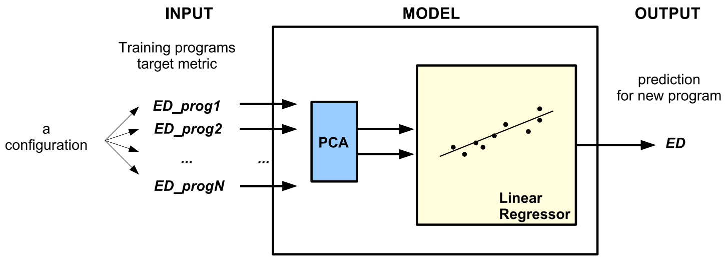


Figure 4.5: Mapping of the training programs to the new program. Given a configuration, the values of a given target metric (for instance ED) is extracted for each training programs. Then PCA is applied in order to reduce the number of those values. They are then fed into the trained linear regressor that produces a prediction for the new program for this particular configuration.

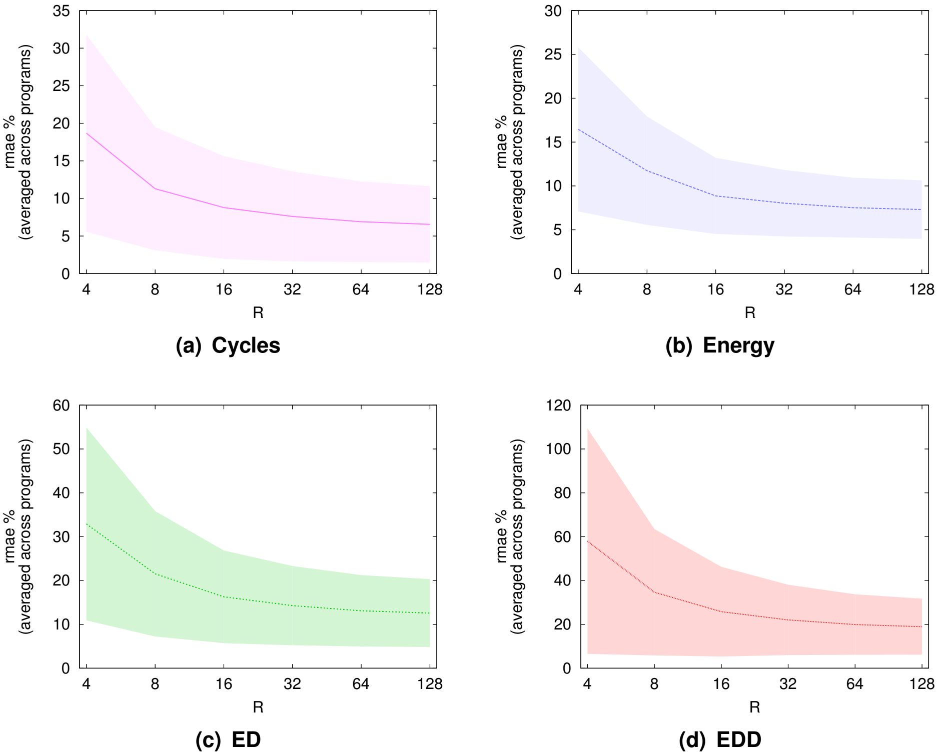
using the weights β_i as can be seen in figure 4.4(c). The following sections describe in more details each step necessary to build this model and evaluates the optimal parameter values for **R** and **T**.

4.4.2 Finding the Linear Mapping

As presented in the overview, the first step towards building the model consists of determining a mapping between the new program and the training programs. This mapping is expressed as a linear combination of the design spaces of the training programs. However, since some of the training programs might correlate with each other, it is desirable to first remove this correlation. This reduces the number of weights to determine and, therefore, increases the accuracy of the linear regressor. Principal Components Analysis (PCA) was applied by keeping 99% of the variance in the input space, resulting in about nine principal components being retained when applied to SPEC CPU 2000. It follows that the number of weights β_i to be determined was reduced. This is in fact very important when using a linear regressor since the number of weights has to be smaller than the number of samples used to determined them.

Figure 4.5 gives an example of how a prediction for ED is made for a new program based on the design spaces of the training programs. For a given architectural configuration, the ED values of each training program are fed as an input to the model. This vector of values is then reduced using PCA. The linear regressor uses these new transformed values to make a prediction for the new program based on the weights determined during the training phase.

To find the optimal weights β_i of the linear model, a few responses **R** from the new program are needed. These responses are nothing but the target metric of interest (for instance ED) for a few configurations. The number of responses **R** hence need to be fixed. To determine the optimal value of **R**, leave-one-out cross-validation was used on the 26 SPEC CPU 2000



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.6: Error averaged across all programs as a function of **R**, the number of training simulations required to compute the linear mapping to the new program. The standard deviation of the error across the programs is shown in the shaded area.

programs. Turn-by-turn, each program is considered as the new program and the remaining 25 ones compose the training set. Figure 4.6 shows the error of the model as a function of **R**, averaged across all the programs. As observed, the error decreases with the number of responses **R**. For values of **R** greater than 32 the decrease is only marginal, which seems to make **R** =32 an ideal choice.

Figure 4.7 shows the coefficient of correlation as a function of **R**. There is a high correlation for values of **R** greater or equal to 16. It can also be observed that for smaller values of **R**, the correlation for cycles and energy differ; cycles is more difficult to predict than energy. The reason for this difference is due to the fact that performance is more likely to be influenced by the program behaviour while the energy consumption is less specific to the program but related to the microarchitecture. Lower values of **R** fail to capture enough of the program behaviour for performance, while for energy a value as small as 4 is sufficient to get a good correlation.

Based on those observations, the number of responses **R** was fixed to 32 since it leads to a low error rate and a good correlation. It means that it is possible to predict any point

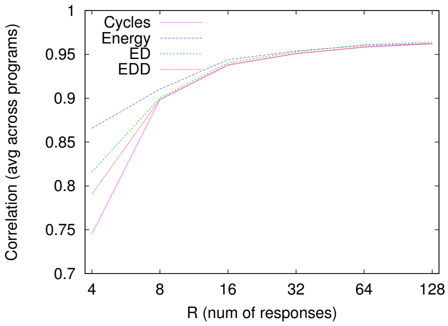


Figure 4.7: Coefficient of correlation averaged across all programs for different number of responses **R**.

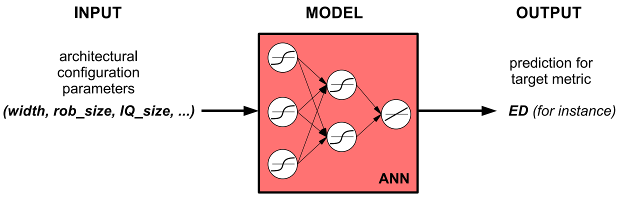
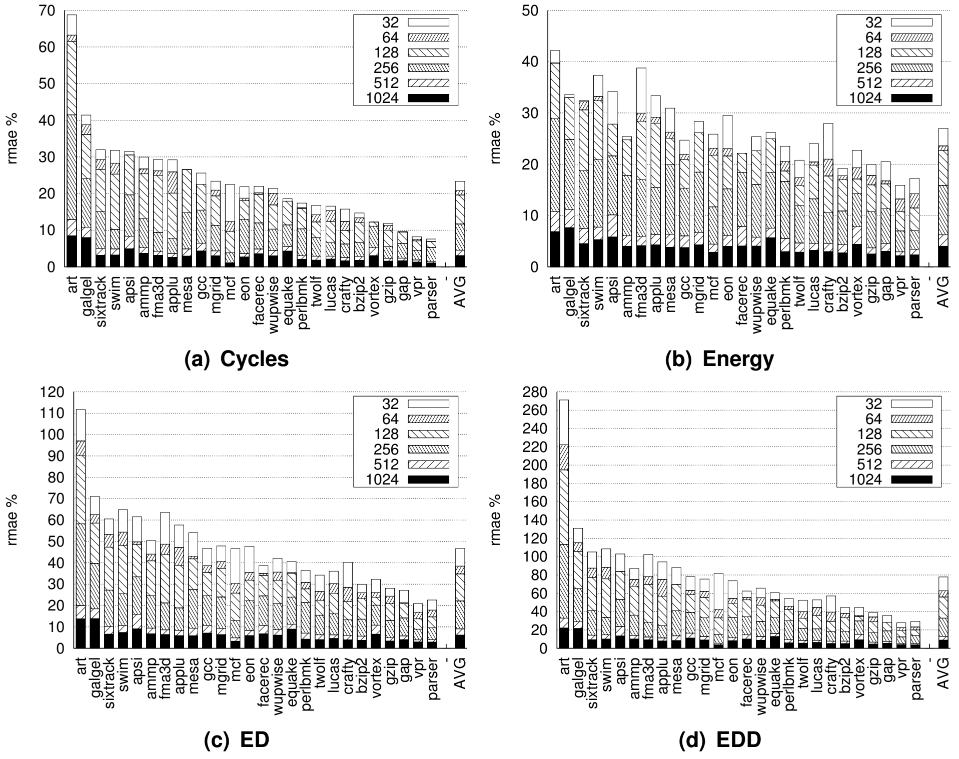

INPUT
MODEL
OUTPUT
architectural configuration parameters
(width, rob_size, IQ_size, ...)
ANN
prediction for target metric
ED (for instance)

Figure 4.8: Model of the single-program predictor. Once trained, the Artificial Neural Network (ANN) makes predictions for any configuration parameters given as an input.

in the design space of a new program with great accuracy using only 32 simulations from it. However, one problem with this approach is that it is only possible to predict configurations that have been seen before for the training programs. The next section describes how to solve this issue by predicting the design space of each training program individually.

4.4.3 Single-program Predictors

Program-specific predictors were used to predict the individual design space of each training program. Those models use as their input the parameters of the architecture configuration and make predictions for any values of these parameters. For each program, a model was built using **T** training samples from the design space. Predicting the performance of different programs across a large design space was studied by many researchers [Ipek 06, Jose 06b, Lee 06]. The specific implementation used in this chapter is based on Artificial Neural Networks (ANNs) [Ipek 06]; this choice was motivated by their ease of use. Other implementations such as radial basis functions or spline functions could have been possible but it was shown they achieve equivalent accuracy [Lee 07b]. Figure 4.8 shows how a single-program predictor is used. It takes as an input the microarchitectural parameters, such as pipeline width, reorder buffer queue size or instructions issue queue size. Then the trained ANN is used and



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.9: Average accuracy of the artificial neural networks broken down per program for various training sizes. The x-axis shows the individual benchmarks and the average. The y-axis shows the prediction error.

outputs a prediction for the target metric of choice (cycles, energy, ED or EDD).

Technically, the ANN used is based on a multi-layer perceptron, composed of an input layer, one hidden layer and an output layer. The number of neurons in the hidden layer was determined empirically. The best performance was reached with ten hidden neurons. The tangent hyperbolic activation function was chosen for the hidden layer whilst a linear function was chosen for the output layer. Once built, the models were trained using the backpropagation algorithm with a maximum number of iterations fixed to 2000.

Because of the nature of ANNs, the input data was normalised between -1 and 1. This normalisation process involves first transforming the microarchitectural parameters that vary by a factor of two, such as the cache size, with the logarithmic function. In a second step, all the inputs are scaled according to the minimum and maximum values they can take. The same scaling technique is applied to the output data (*i.e.* cycles, energy, ED or EDD) using only the values observed in the training set.

The training phase of the models involves gathering some training data. Since this training data corresponds in fact to real simulations, it is desired to determine what is the optimal num-

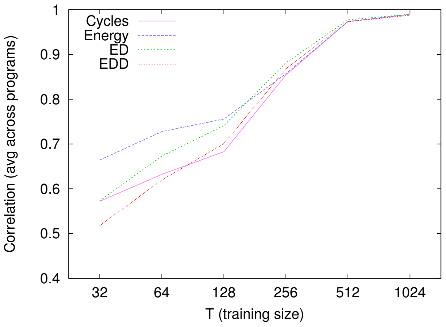


Figure 4.10: Coefficient of correlation averaged across all programs for different training sizes for cycles, energy, ED and EDD.

ber of simulations needed for each program. Obviously the amount of training data influences the accuracy of the model; the more training data the more accurate the model is.

Figure 4.9 shows the mean error of the models as a function of **T**, the training size, for each program of the SPEC CPU 2000 benchmark suite for the different target metrics. Because random initialisation of the weights of the ANNs can lead to variation in their performance, each ANN was trained 20 times. The performance was then averaged for each program and validated using cross-validation. As expected, the error of the models decreases as the number of training samples **T** is increased. It can be observed that for less than 512 training samples, large variation across the programs occur. For instance, if the ED metric is considered, a maximum error of 58% is reached for the program *art* when only 256 training points are used. With **T** =512 the maximum error is just 20%. On average the error achieved when using 512 samples is below 10% for ED, which seems to be a good tradeoff between the number of simulations required for training and the accuracy achieved by the model.

To further validate this choice, a study of the correlation between the predictions and the actual values was performed. Figure 4.10 shows the coefficient of correlation averaged across all the SPEC CPU 2000 programs for different training sizes and target metrics. A correlation of 0.98 is achieved with 512 training data points, while increasing this to 1024 points brings little further improvement. This further confirms that the choice of **T** =512 as the number of trainings required per program represents the best tradeoff.

4.4.4 Predicting a New Program

Having determined the optimal values of **T** =512 and **R** =32, the program-specific predictors and the linear mapping can now be considered together, as shown in figure 4.11. To make a prediction for any microarchitecture, the configuration parameters are given to the model as

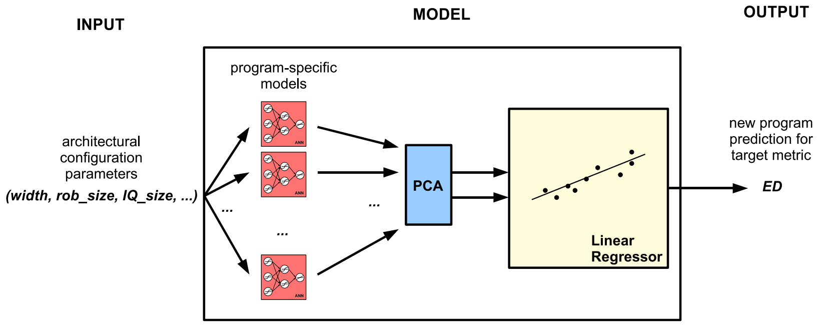
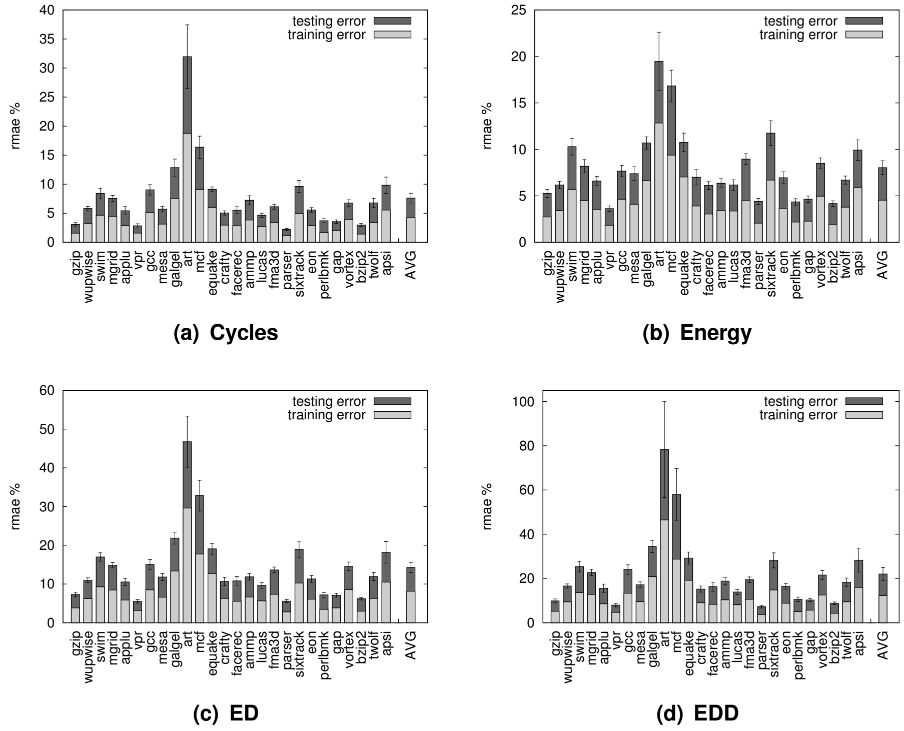

INPUT
MODEL
OUTPUT
program-specific models
architectural configuration parameters
(width, rob_size, IQ_size, ...)
ANN
PCA
Linear Regressor
new program prediction for target metric
ED

Figure 4.11: The complete model that can predict the performance of any design point for any new program. For each of the training programs, a prediction is made for a particular architecture configuration using the ANNs. These predictions are then reduced to a handful of values using PCA and given as an input to the linear regressor. This linear regressor finally makes a prediction for the new program using the weights found during the training phase.

the input. Then the program-specific models are used to determine the predicted value for each program in the training set. Finally those predictions are reduced to a few values using PCA and combined using the linear mapping in order to make a prediction for the new program.

The accuracy of the final model is evaluated for each program of the SPEC CPU 2000 benchmark suite. As before, this process was repeated 20 times using leave-one-out cross-validation. Figure 4.12 shows the training and testing error achieved by the model for each of the four metrics evaluated. The training error is derived from the error made by the model on the training data ($\mathbf{R} = 32$ for each program) whilst the testing error is the error made when testing the model on the remaining unseen data. The testing error will be referred simply as the error from now on.

The model achieves an average error of 8% for cycles and enegy, 14% for ED and 21% for EDD. Some programs have a bigger error in comparison with others. For instance program *art* has an error of 32% for cycles and 19% for energy and program *mcf* an error of 16% for cycles and 17% for energy. As seen in section 4.3.2, these programs are indeed very different from the others. Therefore, it is difficult to use the knowledge gathered from the training programs. This explains why the model's error is high for these programs when compared with others. Interestingly, it is possible to use the training error as an indicator of whether the model is going to work well or not: the higher training error is, the higher testing error. Therefore, if the architecture-centric model is expected to lead to a high error for a particular program, a single-program predictor could be used instead. In fact as an extension to the work presented in this chapter, one could build the model incrementally by adding more programs to the training set as new programs are encountered that differ significantly from the ones already seen.



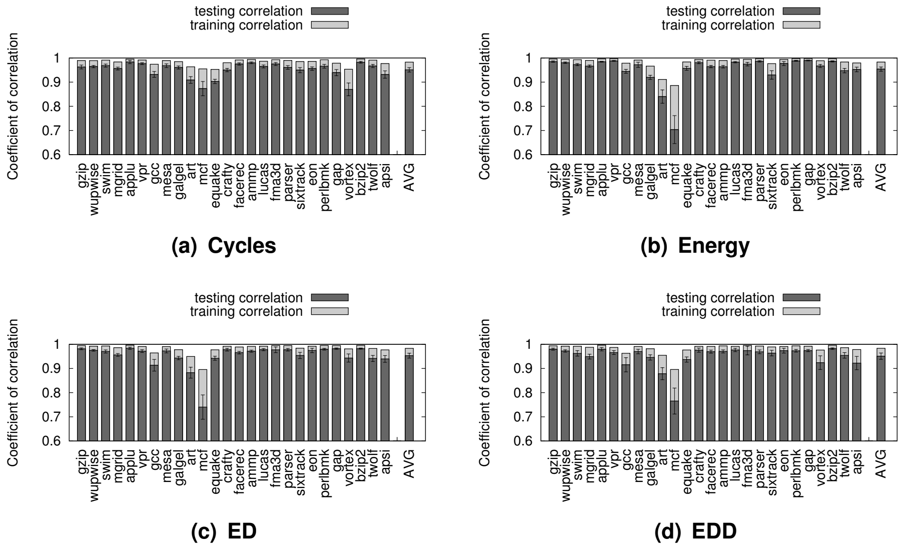
(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.12: Training and actual mean error for each program of SPEC CPU 2000 (the lower the better). The actual error corresponds to the prediction error when testing on the remaining points of the space not used for training. The standard deviation is also shown since the training has been repeated 20 times picking each time different training samples.



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.13: Coefficient of correlation for each program of SPEC CPU 2000 based on either the training set or the testing set (the higher the better).

Figure 4.13 shows the coefficient of correlation for the model. An average correlation of 0.95 for all the four metrics is achieved. As before, programs *art* and *mcf* perform slightly worst than the others. Nonetheless, most of the programs exhibit correlation above 0.9, independently of the metric of interest. This shows that the model developed in this section is actually working across the vast majority of the SPEC CPU 2000 applications.

4.4.5 Predicting MiBench From SPEC CPU 2000

So far, leave-one-out cross-validation was used to verify and assess the performance of the model developed. While this validation technique is well founded, one could argue that because the validation was performed within the same benchmark suite, SPEC CPU 2000, the technique might in actual fact not work for programs from other benchmark suites.

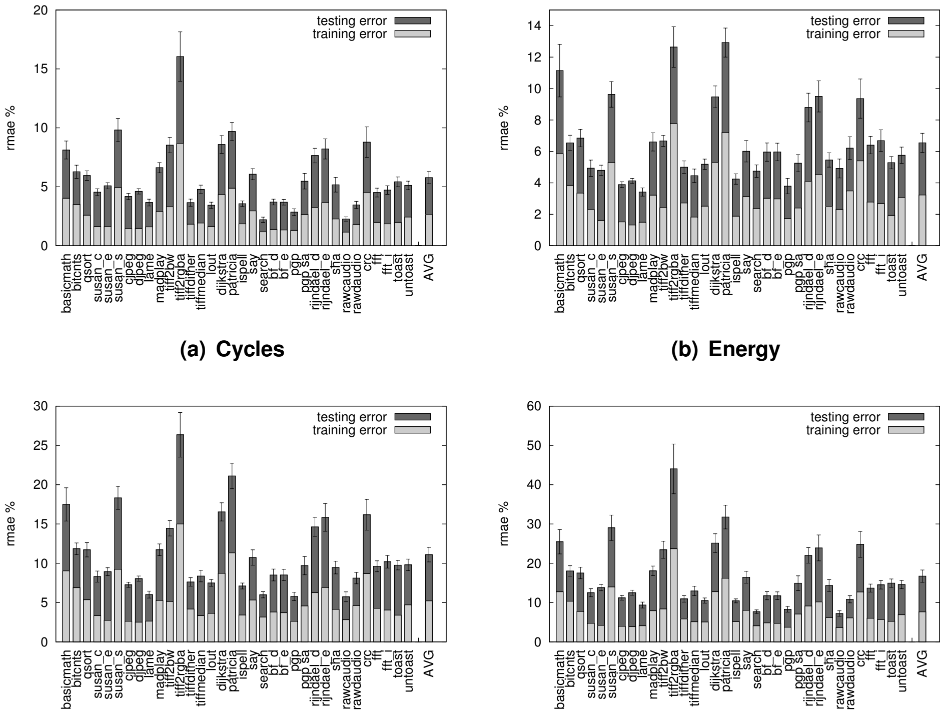
To verify that the model is able to make accurate predictions for programs outside the benchmark suite used for training, this section uses SPEC CPU 2000 to predict each of the programs from the MiBench benchmark suite. Moreover, since MiBench benchmarks are mainly targeted at embedded systems, it enables testing of the models across a different application domain.

Figure 4.14 shows the error of the model when predicting MiBench from SPEC CPU 2000. The average prediction error is about 6% for cycles, 7% for energy, 12% for ED and 18% for EDD. These errors are slightly lower than the errors found when using leave-one-out cross-validation on SPEC CPU 2000. This can be explained by the fact that for SPEC CPU 2000, a few programs (*art* and *mcf* for instance) have a very different behaviour from the others. Therefore, the model's accuracy for these programs is lower than for the others, resulting in an increase in the average error. However if one dismisses these two programs, there is no fundamental difference in terms of error between these two benchmark suites.

When correlation is considered, figure 4.15, the same conclusion can be drawn; MiBench has no program that behaves significantly different from the others. In a nutshell, the prediction of MiBench from SPEC CPU 2000 is as accurate as predicting the programs of SPEC CPU 2000 using leave-one-out cross-validation.

4.4.6 Summary

This section has described and evaluated a model, the architecture-centric predictor, that learns across programs. Once trained, this model makes accurate predictions for any new program encountered using just 32 simulations from it. This is achieved by exploiting program similarities. This model is capable of predicting the microarchitectural design space for any new unseen program.



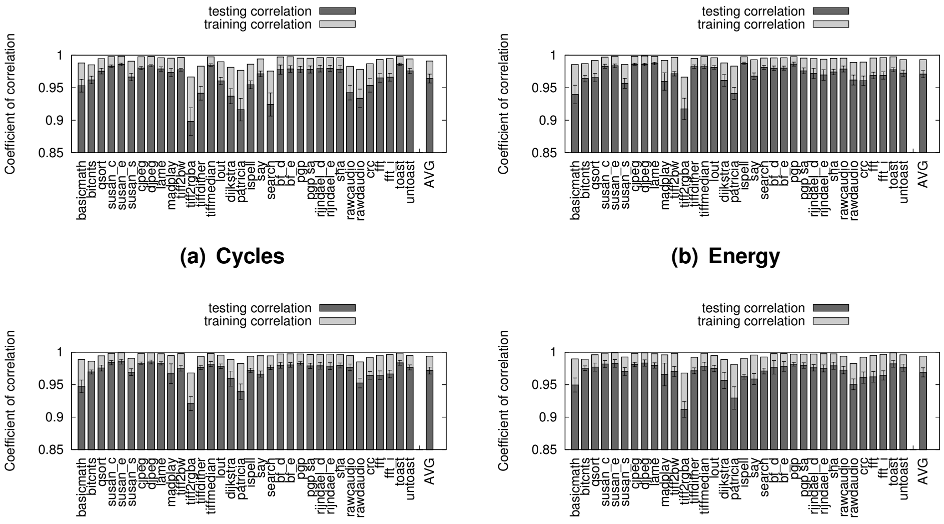
(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.14: Training and actual mean error of the model trained on SPEC CPU 2000 for each program of MiBench (the lower the better).



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.15: Coefficient of correlation of the model trained on SPEC CPU 2000 for each program of MiBench based on either the training set or the testing set (the higher the better).

However, if a program differs significantly from anything seen so far, the model might fail to accurately predict its space. Fortunately, by examining the training error it is possible to detect such exceptional cases (two programs out of 60 in our experimental setup). If such cases happen, the model can suggest building a single-program predictor using more simulations.

This section also demonstrated that the model is not restricted to any specific application domain. As shown, it is possible to predict with great accuracy all the programs composing the MiBench benchmark suite from the ones of SPEC CPU 2000. The next section looks at how this model can be extended to the prediction of the average behaviour of a complete benchmark suite.

4.5 Benchmark Suite Predictor

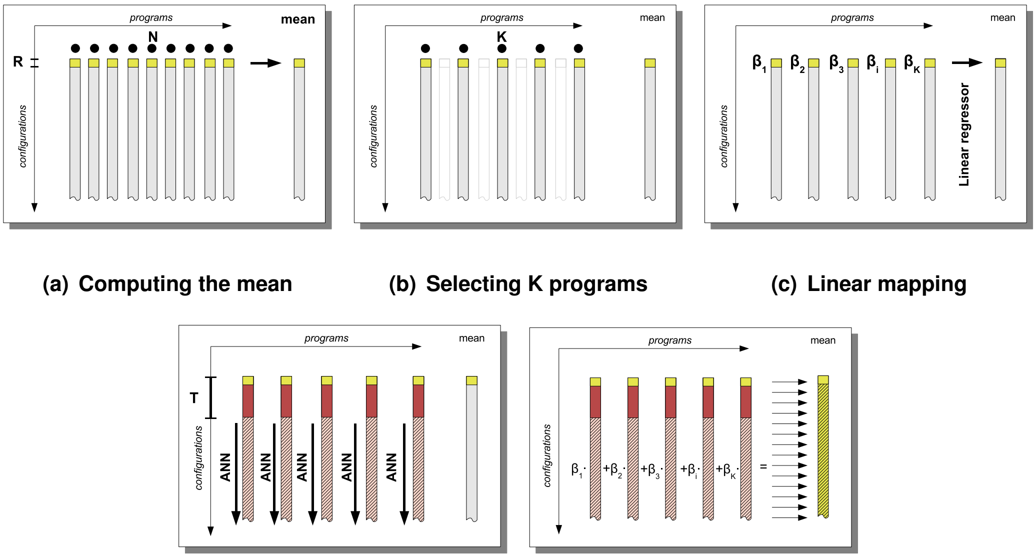
The previous section presented a model that predicts the entire design space of any new program using other training programs. While the cost of training can be amortised over time, when more programs need to be predicted, architects might in fact already have a specific set of programs that they want to use to explore the microarchitectural space. This case can arise when using different inputs or when compiler optimisations are considered for instance.

What is really needed is a model that can cut the number of simulations required when a large number of programs are used. This section develops a model that predicts the average behaviour of the whole set of applications, as opposed to predicting each new program individually. Indeed a key element in design space exploration consists of quickly discarding uninteresting configurations and to focus only on the most promising ones. For this purpose, a model can be built and used to predict the average behaviour of a set of programs using a reduced number of simulations. Each program can then be simulated individually on those selected designs.

This section extends the architecture-centric model presented in the previous section to predict the average behaviour of a whole benchmark suite, where all the programs are known in advance. This new model is called the “benchmark-suite predictor”. In this setup, the amount of training needed to build the model, *i.e.* number of simulations, can be reduced to a strict minimum while maintaining the prediction accuracy.

4.5.1 Overview

This benchmark suite predictor is built in four different stages, as shown in figure 4.16. First **R** runs are performed on **R** randomly selected configurations for each of the **N** benchmarks within the suite. These **R** runs are similar to the responses concept introduced in the previous



(a) Computing the mean

(b) Selecting K programs

(c) Linear mapping

(d) Single-program predictors

(e) Predicting the mean

Figure 4.16: An overview of the benchmark suite predictor. First **R** randomly-selected simulations for each program are run to calculate the mean (a). Then **K** programs are selected to represent the whole suite, reducing the training requirements (b). Then a combination of these **K** programs to the mean is found using linear regression (c) and artificial neural networks are trained using **T** training samples each (d). Finally using the linear mapping the mean can be predicted for any point in the design space (e).

section. However, in this case these **R** responses are used to compute the average behaviour of the benchmarks for the metric of interest, as shown in figure 4.16(a). The average behaviour can be characterised by different means which includes the arithmetic, geometric, harmonic, or any other statistical property, such as the standard deviation.

Once this mean has been calculated for a few configurations, a greedy algorithm is used to select **K** representative programs from the **N** programs present in the suite. This algorithm is described in figure 4.16(b). The **K** programs are selected such as to best represent the whole benchmark suite. Then a weighted sum of their execution time is used to approximate the mean of the whole benchmark suite. Since the real mean for the **R** configurations is known, this information can be used to map from the **K** programs to the actual mean, as shown in figure 4.16(c). In other words, the average performance of the **N** programs of the benchmark suite is expressed as a linear combination of just **K** programs, where **K** <**N**. Information about the **N**-**K** dropped programs is in fact represented within this mapping.

Finally, as for the previous developed model, ANNs are used to predict the individual spaces of each of the **K** representative programs (figure 4.16(d)), using **T** =512 randomly-selected training simulations as seen in section 4.4.3. These predictors are then combined using the linear mapping to predict the average behaviour of the full benchmark suite for all remaining configurations as shown in figure 4.16(e).

In the following subsections, the benchmark suite predictor is described in more details and the optimal parameters for **R** and **K** are determined using the SPEC CPU 2000 benchmark suite.

4.5.2 Computing The Mean

As discussed in the previous section, the first step in building the benchmark suite predictor consists of randomly selecting **R**, a small number of configurations from the design space, and simulating all the programs on these configurations. From these runs, the mean can then be computed for each of the **R** configurations (figure 4.16(a)). The **R** configurations are chosen from the space using uniform random sampling.

The model can be used to predict any type of means (or other statistical metrics) required by the user. In this section most of the results are shown for the geometric mean, defined as

$$geometric\ mean = \mu_g = \sqrt[N]{\prod_{i}^{N} x_i}$$

where **N** is the number of benchmarks in the suite and x_i is the metric of interest for benchmark i. The choice of the geometric mean is motivated by the fact that the target metric is normalised for each program to a baseline architecture, hence giving a ratio. However, later in section 4.5.5 results will show that the model is not limited to the prediction of the geometric mean. Other metrics such as the arithmetic and harmonic means, as well as the standard deviation across the programs can be accurately predicted by the model.

4.5.3 K Representative Programs

Having computed the mean for **R** configurations from the **R** runs, the next step consists of selecting the **K** representative programs. Since programs share similarities, it is possible to capture the average behaviour of a benchmark suite using only a subset of its programs (figure 4.16(b)). As mentioned in the overview, the choice of this subset is made using **R** responses from each program. Based on these **R** values, it is possible to find **K** programs that best represent the whole benchmark suite.

A greedy algorithm, shown in figure 4.17, was developed to select the most **K** representative programs. This algorithm assumes that for a given value of **K**, the optimal subset of **K**

```
Input: N programs from the benchmark suite, number of K programs to keep
Output: P : the set of K most representative programs
$P$ = {all N programs};
for $i \leftarrow N$ to $K$ do
    foreach program, p, from remaining set $P$ do
        Remove $p$ from $P$;
        Estimate error using R training simulations;
        Add $p$ back into $P$;
    Remove $p$ that produces the min. error from $P$.
```

Figure 4.17: The greedy algorithm that selects the **K** most representative programs from the whole benchmark suite.

representative programs is included in the optimal subset of $\mathbf{K}+1$ programs. Hence the algorithm proceeds iteratively by removing one-by-one each program and record the associated training error. At each step it removes the program that lead to the minimum recorded training error. The rational being that the programs removed are the ones whose behaviour are already captured by the remaining ones and hence lead the smallest increased in error when removed.

Ideally the selection process is repeated for all the four different target metrics (cycles, energy, ED and EDD) that the model predicts. However, because the selection process can only be done once, one of these four metrics need to be used to guide the selection of the **K** representative programs. ED was chosen since it represents the tradeoff between cycles and energy, ensuring that the choice of the representative programs remains the same independently of the metric of interest. This contrasts with prior work where the selection of the representative programs was based on their configuration-independent features [Eeck 02] rather than the configuration-dependent behaviour, *i.e.* the **R** responses.

Table 4.5 shows which programs of SPEC CPU 2000 are often selected as part of the **K** representative programs, when **K** is fixed to 10. The algorithm was executed 20 times using each time different random $\mathbf{R} = 32$ responses from each program in the benchmark suite. As it can be seen six programs appear more than half of the time within this set of representative programs. This means that those programs are essential to characterise the benchmark suite. The reason why the selected programs differ from one run to another run of the algorithm is due to the fact that many of these programs are similar. Hence choosing one or another similar does not affect the accuracy. As expected, those programs that are often selected are not similar. If they were, they could be replaced by other similar ones and not occur that often. Looking back

Program	Frequency
equake	16/20
art	15/20
mesa	13/20
apsi	12/20
galgel	11/20
mcf	11/20

Table 4.5: Programs appearing more than half of the time in the **K** $= 10$ most representative programs of SPEC CPU 2000. The greedy algorithm was run 20 times using different **R** $= 32$ responses each time.

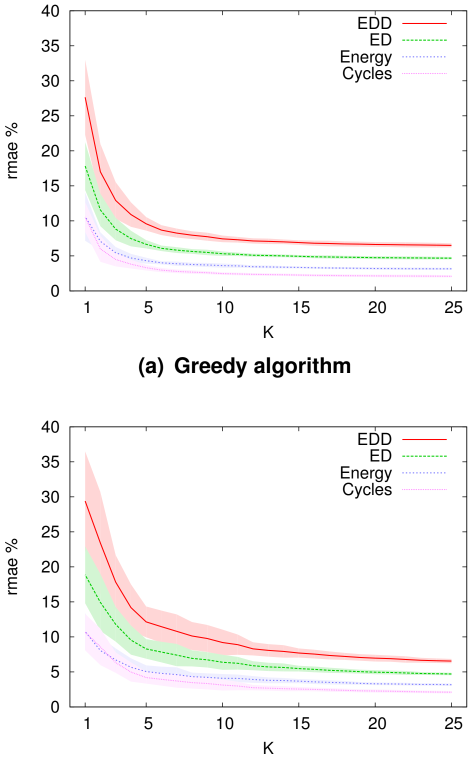
at figure 4.2(c), it is clear that the distances between these six programs are high on average: they are at least at an average distance of 85 from each other. The selection made by the greedy algorithm clearly identifies programs that belong to different clusters.

Having presented the greedy algorithm and its output, its efficiency is evaluated for all possible values of **K** (for SPEC CPU 2000) against a random selection of **K** programs. The final decision on the value of **K** depends on the number of programs in the benchmark suite and the prediction error that the user requires. The number of responses **R** was set to 32 since it is the optimal number found for the architecture-centric predictor (section 4.4.2). This choice of **R** is again discussed and further evaluated in the next section. The single-program predictors used for each representative programs were trained using **T** $= 512$ samples (section 4.4.3).

The mean prediction error for the greedy algorithm and random selection is shown in figure 4.18 and the associated correlation in figure 4.19. As can be seen the greedy algorithm outperforms random selection for small values of **K** (< 5) in terms of error and correlation. Furthermore, the standard deviation of the greedy algorithm is significantly lower than random selection overall. Indeed the random selection process is more likely to give different results each time it is invoked. These results show that it is possible to capture the behaviour of a whole benchmark suite using only a fraction of its programs.

4.5.4 Mapping K Programs To The Mean

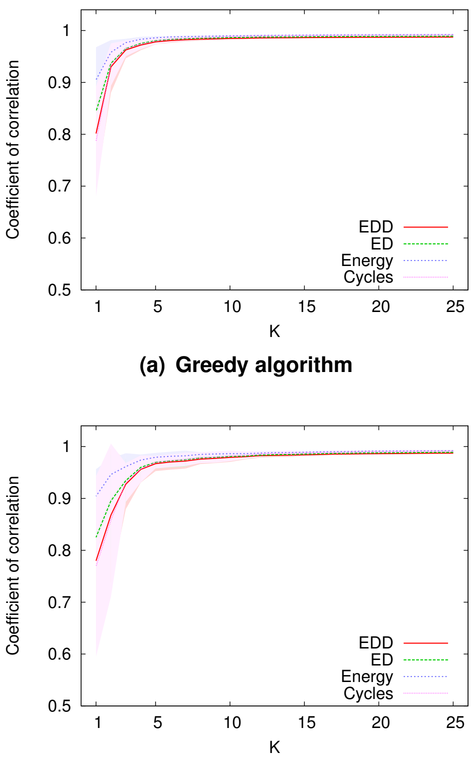
Having seen how to select **K** representative programs, they must now be combined to predict the behaviour of the benchmark suite for a given metric (figure 4.16(c)). To this end, a linear mapping is learnt between the **K** representative programs and the mean for the **R** configurations for which the exact mean value is known (section 4.5.2). This mapping accounts for the programs dropped.



(a) Greedy algorithm

(b) Random selection

Figure 4.18: Mean error and its standard deviation (in the shaded areas) as a function of **K**, the number of representative programs needed to build the linear model. The greedy algorithm performs better than random selection for small values of **K**, and has a much lower standard deviation overall.



(a) Greedy algorithm

(b) Random selection

Figure 4.19: Correlation and its standard deviation (in the shaded areas) as a function of **K**, the number of representative programs needed to build the linear model. As for the error, the greedy algorithm performs better and has a lower standard deviation.

Given **R** values of the performance metric for each of the **K** programs and **R** values for the actual mean, an estimator $\hat{\mu}$ of μ which uses **K** rather than all **N** (**K** $\ll$ **N**) values is built to estimate the mean. For means such as the arithmetic or harmonic, which are already in a sum form, this is straightforward. However, since the geometric mean is in product form, it first needs to be transformed into a sum by applying the natural logarithm function :

$$\begin{aligned} ln(\mu_g) &= ln(\sqrt[N]{\prod_{i}^{N} x_i}) \\ &= \frac{1}{N} \cdot \sum_{i}^{N} ln(x_i) \end{aligned}$$

Hence the linear model can be expressed as :

$$ln(\hat{\mu_g}) = \sum_{i}^{K} \beta_i \cdot ln(x_i)$$

where β_i is the weight for benchmark i and x_i is the target metric for benchmark i. If all the information is available to the model ($\mathbf{K} = \mathbf{N}$), all the weights (β_i) have a value of $\frac{1}{N}$ and the estimation is perfect *i.e.* $\hat{\mu_g} = \mu_g$. When the number of programs, **K**, used to estimate the weights β_i is smaller than **N**, then the weights β_i are updated accordingly to account for the removed programs.

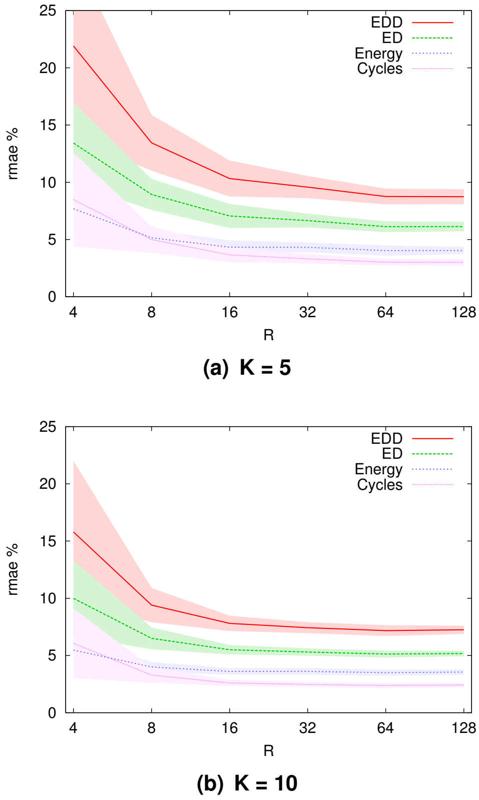
For the same reason as in section 4.4.2, PCA is applied beforehand to reduce the number of weights β_i needed to be estimated by the linear regressor. By keeping 99% of the variance, the total number of weights to estimate is typically reduced to nine.

The quality of the linear mapping between programs and the geometric mean depends on the number of training simulations **R** available per program. Until now this value was set to $\mathbf{R} = 32$ based on the optimal value found for the architecture-centric predictor in section 4.4.2. However since the benchmark-suite predictor is a different model, this parameter is now reevaluated. Since the **R** configurations are randomly selected for each benchmark, the training process was repeated 20 times and the average and variance reported. Figure 4.20 shows the error of the model and figure 4.21 its correlation as **R** is varied between 4 and 128 simulations per program for two arbitrary choices of **K**, the number of representative programs. It can be observed that for values of **R** greater or equal to 16 the accuracy of the predictor reaches a stable plateau for $\mathbf{K} = 10$. However when $\mathbf{K} = 5$ the accuracy can be improved by having more responses **R**. Hence the choice of $\mathbf{R} = 32$ was also verified to be a good value for the benchmark-suite predictor, leading to an error rate of 7% and a correlation of 0.98 for ED when using only five representative programs.

4.5.5 Predicting Different Metrics

The specific choice of which mean to use in any particular situation depends on the metric being averaged and is orthogonal to the focus of this work. This section demonstrates that the results showed for predicting the geometric mean also hold for other type of means. Furthermore, it shows that this technique can be used to predict the standard deviation of the benchmark programs from this mean, allowing the user to distinguish between microarchitectural configurations with similar mean values, but different behaviour across the benchmarks.

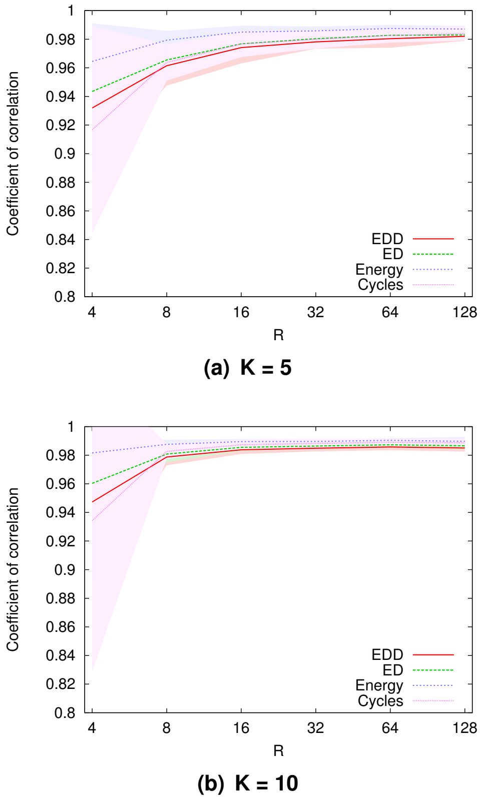
The type of mean to use depends on the metric the user needs to predict. For instance if the user wants to predict the absolute number of cycles, then the arithmetic mean must be used. If the metric to predict is a ratio, such as the number of cycles for a given configuration over a



(a) K = 5

(b) K = 10

Figure 4.20: Error and its standard deviation as a function of **R**, the number of training simulations required to compute the linear mapping to the geometric mean for two values of **K**, the number of representative programs retained.



(a) K = 5

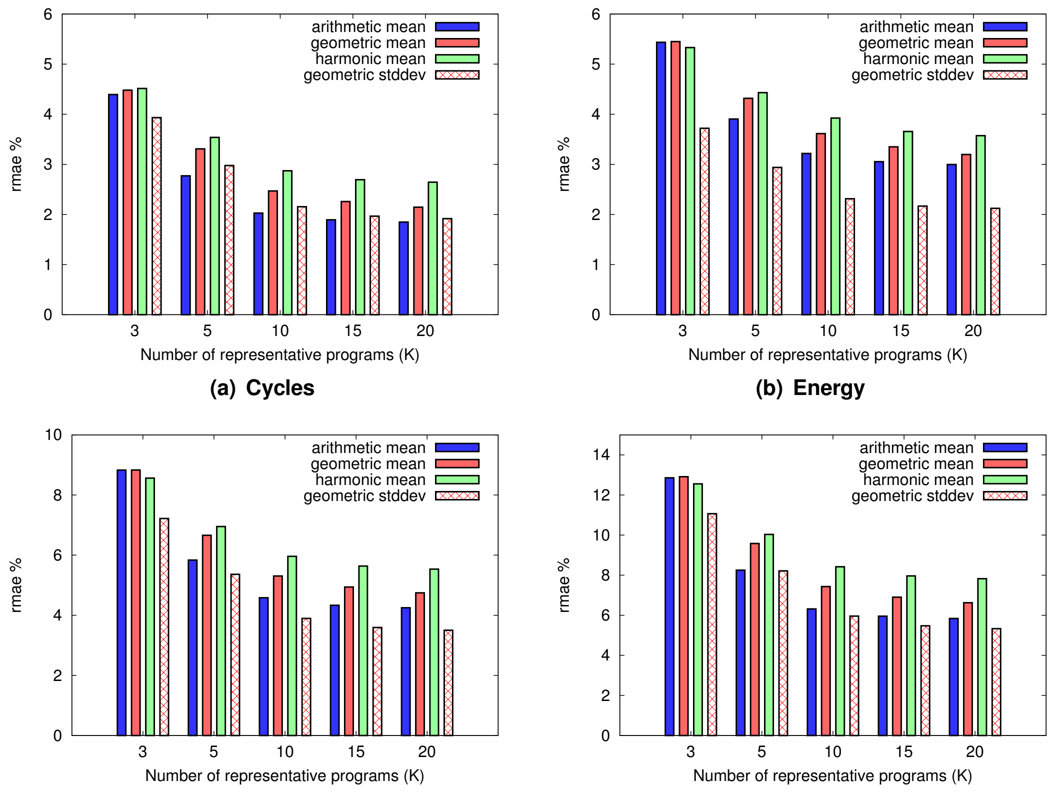
(b) K = 10

Figure 4.21: Coefficient of correlation and its standard deviation as a function of **R** for two values of **K**.

baseline, then the right metric to use is the geometric mean as seen throughout earlier sections. And finally, if the measured metric is a speed, such as IPC, then the harmonic mean is the right choice. For this reason, the three type of means considered and their corresponding formulae are :

$$\begin{aligned}
arithmetic\ mean &= \mu_a = \frac{1}{N} \cdot \sum_i^N x_i \\
geometric\ mean &= \mu_g = e^{\frac{1}{N} \cdot \Sigma_i^N ln(x_i)} \\
harmonic\ mean &= \mu_h = \frac{N}{\Sigma_i^N \frac{1}{x_i}}
\end{aligned}$$

When it comes to predicting the standard deviation across the benchmarks, a specific for-



(a) Cycles

(b) Energy

(c) ED

(d) EDD

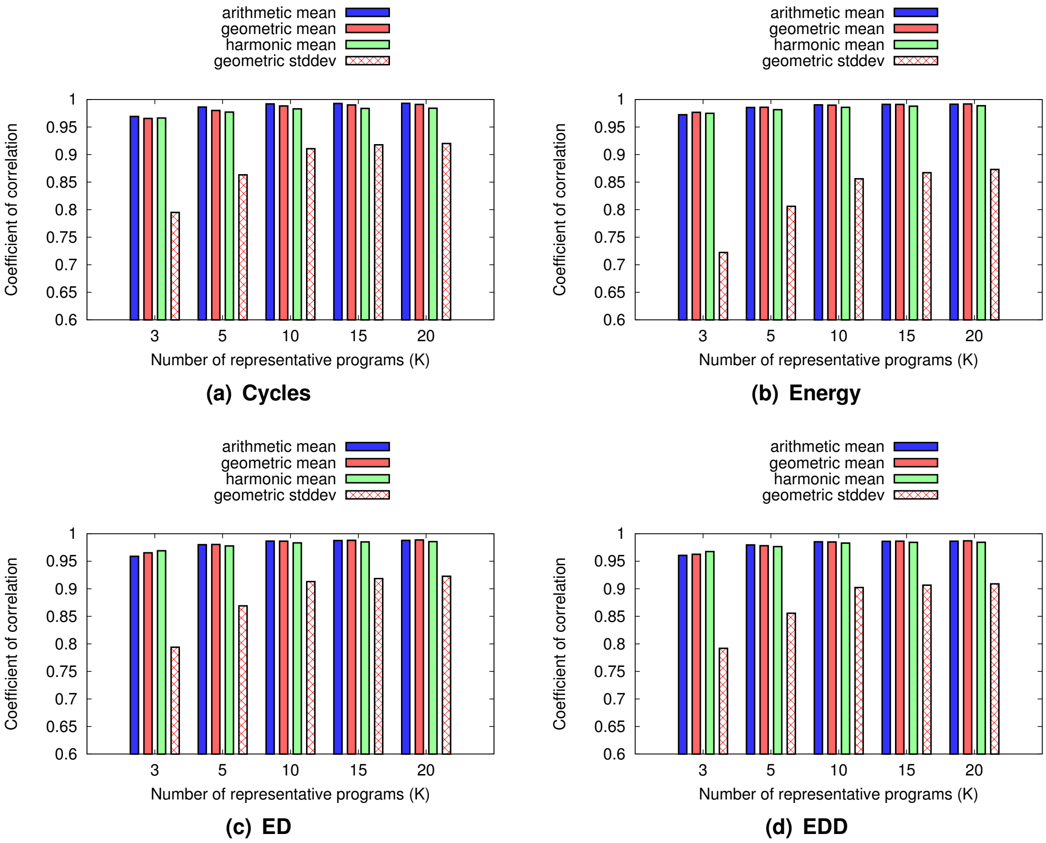
Figure 4.22: Error for the arithmetic, geometric and harmonic means as the number of representative programs **K** is varied, . Also shown is the prediction error of the standard deviation of the geometric mean across the different benchmarks.

mula must also be used depending on the chosen mean metric. To simplify it was decided to simply focus on the geometric standard deviation defined as :

$$\textit{geometric standard deviation} = \sigma_g = e^{\sqrt{\frac{1}{N}\cdot\Sigma_i^N (ln(x_i)-ln(\mu_g))^2}}$$

Figure 4.22 shows the results from predicting the three means and the standard deviation for cycles and energy for various values of **K**, the number of representative programs. As can be seen, predicting the arithmetic mean produces the lowest error of about 2% for cycles and 3% for energy (when $\mathbf{K} \geq 10$). The other two means have a slightly higher error rate since their formula is more complicated to compute. Despite this, the model still achieves a low error of around 3% for cycles and 4% for energy when the harmonic mean is considered (the most difficult to predict). A good prediction accuracy is achieved with just $\mathbf{K} = 5$ representative programs, independently of the mean used.

Looking at the prediction error of the standard deviation for cycles, energy, ED and EDD,



(a) Cycles **(b) Energy**

(c) ED **(d) EDD**

Figure 4.23: Correlation for the arithmetic, geometric and harmonic means as the number of representative programs **K** is varied. Also shown is the correlation for the standard deviation of the geometric mean across the different benchmarks.

it seems at first that predicting the standard deviation is easier than predicting the means. However, this analysis would not be complete without looking at the correlation. Figure 4.23 shows indeed that the coefficient of correlation for the standard deviation is consistently lower than for the means. Considering energy for instance, the correlation is only 0.7 with $\mathbf{K} = 3$ for the standard deviation where it is above 0.95 for the three means. However the correlation improves as more representative programs are selected. When $\mathbf{K} = 10$ a reasonable correlation of 0.85 for energy and 0.9 for cycle, ED and EDD is achieved. This result is in fact intuitive: since the standard deviation is a measure of how each program differ from the average behaviour of the whole suite, it is expected to need more than a couple of programs to make a good estimation.

4.5.6 Summary

This section described the benchmark-suite predictor, a model that predicts the average behaviour of a benchmark suite for any particular design point. This model gathers $\mathbf{R} = 32$ responses from all the **N** programs present in the benchmark suite and uses them to compute the mean. Then it selects **K** representative programs based on these **R** responses and learns a linear mapping between those **K** programs and the mean of interest. As shown, the predicted values can be any kind of mean or other statistical property of the whole benchmark suite for cycles, energy, ED and EDD.

The total cost of this predictor for a benchmark suite containing **N** programs is $512 \cdot \mathbf{K} + 32 \cdot (\mathbf{N} - \mathbf{K})$ simulations. As seen, when predicting the whole SPEC CPU 2000, a good accuracy is reached using only $\mathbf{K} = 5$ representative programs. In the following sections, a practical example of using the predictor to search the design space is presented and a comparison in terms of error and simulation cost is conducted with two state-of-the-art techniques.

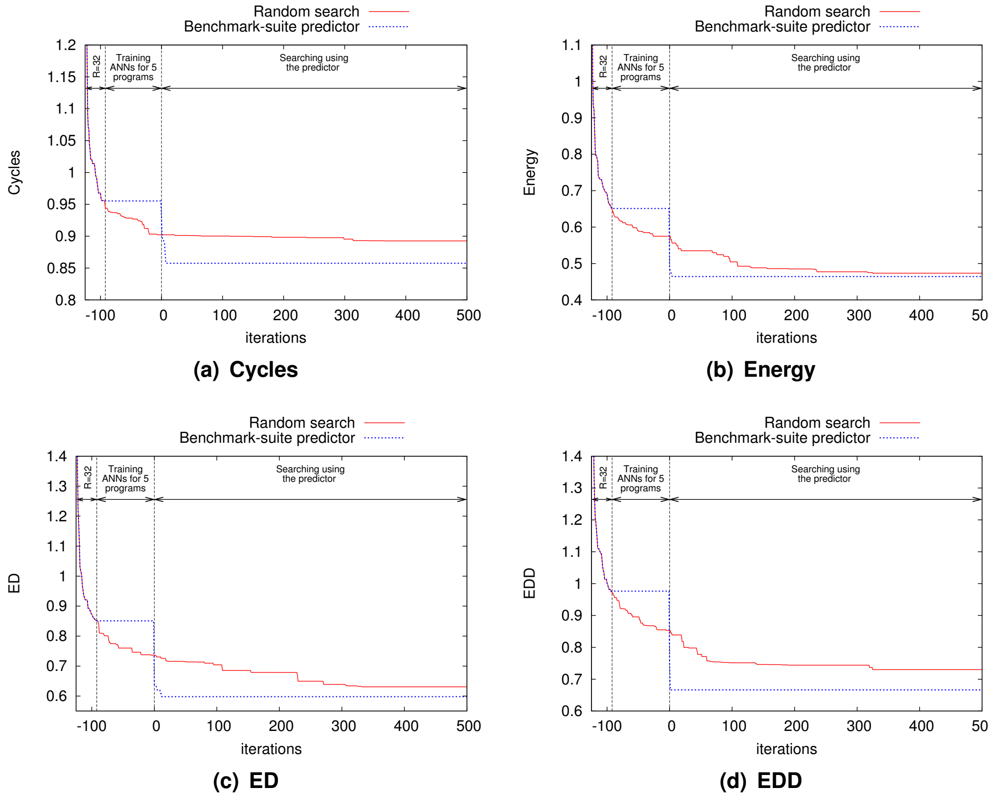
4.6 Searching the Space

The previous section presented and evaluated the benchmark-suite predictor using the mean absolute error and the coefficient of correlation. While those metrics are important to measure and compare the efficiency of predictors, it is also important to keep in mind one of the reasons for creating these models in the first place: exploring the design space. This section shows the performance of the benchmark-suite predictor when used for searching for the best configurations in the space.

4.6.1 Searching For The Best Configurations

Searching a design space using a predictor can be done in several ways, ranging from applying local search techniques such as hill climbers, global search strategies such as simulated annealing or using genetic algorithms. If the predictor is accurate and fast, the whole design space could be ideally ranked by the predictions and the search could start with the design point leading to the best prediction. However, due to the massive design space considered in this work and to simplify the evaluation of the search, it was decided to consider solely the 3000 random samples from the space analysed throughout this chapter.

For a given metric of interest, each of these 3000 configurations from the sample space were given a rank according to the prediction made by the benchmark-suite predictor. The search can then take place by simulating the configuration with the highest rank, then the second highest, and so forth until the performance does not improve anymore. This technique



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.24: Searching the design space for the best configurations for cycles, energy, ED or EDD. The result of random search is shown as well as the result using the benchmark-suite predictor. The predictor first extracts $\mathbf{R} = 32$ responses from each program, then some simulations are used to train the ANN models for each of the $\mathbf{K} = 5$ representative programs. Finally the search using the predictor starts at iteration 0, once the training done. The search quickly converges towards the minimum in less than 10 iterations (the flat dotted blue line).

was compared with a purely random search of the space. The benchmark-suite predictor was built using $\mathbf{K} = 5$ representative programs since this leads to a good accuracy as demonstrated in the earlier sections.

Because some simulations are needed to obtain the $\mathbf{R} = 32$ responses and to train the model for the $\mathbf{K} = 5$ representative programs, the random search starts in fact 124 iterations earlier than the search using the predictor. In other words, an advantage is given to random search corresponding to the simulations needed to train the model. An iteration consists of the evaluation of the design space for the 26 programs that compose the SPEC CPU 2000 suite. Thus 124 iterations corresponds to 3224 simulations ($124 \cdot 26 = 3224$) which is roughly the amount necessary to gather $\mathbf{R} = 32$ responses ($32 \cdot 26 = 832$) and to train $\mathbf{K} = 5$ ANN models using $\mathbf{T} = 512$ simulations each ($832 + 5 \cdot (512 - 32) = 3232$).

Figure 4.24 shows how random search and the benchmark-suite predictor perform when searching the design space for the best microarchitectural configuration for cycles, energy, ED and EDD (relative to the baseline configuration). Because of the intrinsic randomness involved with random search, the search was conducted 20 times and at each iteration the median value was computed. The same was performed 20 times for the benchmark-suite predictor since the training phase involves the random selection of the training samples (visible in figure 4.24). First the $\mathbf{R} = 32$ responses are extracted for each program; both search techniques have obviously the same performance at this stage. Once the responses are gathered, another training phase is needed this time to train the single-program models for the $\mathbf{K} = 5$ selected programs. At this stage, the search is stopped for the benchmark-suite predictor and random search is taking the lead. Once this training period is over, a search using the predictor can effectively start.

It can be seen in figure 4.24, once the model is trained, it needs only a few simulations to reach the minimum value in the sampled space. In fact less than 10 iterations are required for any of the metrics considered. This really shows that the predictor achieves a fairly accurate ranking of the space in almost no time; predictions for the 3000 samples can be made in a matter of seconds. It is interesting to notice that random search performs quite well for energy compared to the other metrics. This is due to the way the space is distributed, as already seen at the beginning of this chapter in figure 4.3(b). However the benchmark-suite predictor still largely outperforms random search even in this case.

4.6.2 Analysis of Best Configurations

The configurations found from the search and their associated parameters resulting in the best value for cycles, energy, ED and EDD in the sampled space are shown in table 4.6(a), alongside with the baseline configuration. The best values obtained for the corresponding configurations for each metric are shown in table 4.6(b).

Best Cycles The microarchitectural configuration giving the minimum number of cycles is obviously aimed towards performance, as table 4.6 shows. It reduces the number of cycles to 86% of the baseline, a 14% saving. However, it also increases energy consumption through the use of a wide pipeline, large register file and many ports into the register file. Energy consumption increases by 18% compared with the baseline configuration.

Best Energy The configuration with the minimum energy consumption has a much smaller register file and smaller L1 caches than the baseline. These structures usually use a lot of

Configuration	Width	ROB	IQ	LSQ	RF	Read	Write	Bpred	BTB	Br	Icache	Dcache	L2
Baseline	4	96	32	48	96	8	4	16K	4K	16	32KB	32KB	2MB
Best cycles	6	128	80	80	112	12	3	2K	4K	16	64KB	16KB	4MB
Best energy	2	144	40	72	48	2	1	32K	4K	32	8KB	8KB	4MB
Best ED	4	136	32	56	88	2	4	32K	2K	8	128KB	128KB	0.5MB
Best EDD	6	152	80	16	80	2	5	8K	4K	8	8KB	128KB	4MB

(a) Microarchitectural parameters

Configuration	Cycles	Energy	ED	EDD
Baseline	1.00	1.00	1.00	1.00
Best cycles	**0.86**	1.18	1.01	0.86
Best energy	1.46	**0.46**	0.68	0.99
Best ED	1.26	0.47	**0.60**	0.76
Best EDD	0.94	0.75	0.71	**0.67**

(b) Best values achieved

Table 4.6: The microarchitectural parameters of the baseline and the best configurations found from the sampled space with their corresponding values for cycles, energy, ED and EDD.

energy and hence smaller ones lead to a large energy saving; less than half (0.46) of the energy consumed by the baseline. However, this configuration leads to an increase in cycles of 46%. Although the reorder buffer, issue queue and load/store queue are larger than the baseline, they actually consume less energy because of the smaller pipeline width: only two instructions wide instead of four. This means that there are fewer ports into each structure and so fewer accesses in each cycle.

Best ED So far, the configurations giving the minimum cycles and minimum energy were considered. For the first, a 14% reduction in cycles resulted in a 18% increase in energy consumption. For the second, a 54% saving in energy meant a 46% increase in cycles. Neither of these processor configurations would be implemented in practise because designers aim for a tradeoff between performance and energy. Therefore, the configuration leading to the minimum ED is now considered.

The minimum ED value achievable is 0.60 as shown in table 4.6(b). This translates into a 26% increase in cycles for an energy reduction of 53%. Compared to the previous configuration that lead to the lowest energy consumption, it almost achieves the same energy savings with a much better performance. This time cycles has only increased by 26% as opposed to 46%. As can be seen in table 4.6(a) when compared to the best energy efficient configuration, this is achieved by having a slightly wider pipeline and a larger register file. The L1 caches are also made much bigger. All these larger structures help increasing performance and indirectly reduce the total energy consumed since less time is spent running the application. However due to the increase in static energy, something needs to compensate for this; the L2 cache is thus made much smaller since its static energy accounts for the largest portion of the energy consumed by the whole processor.

Best EDD As just seen for the ED metric where both cycles and energy are optimised together, big energy savings can still be achieved with limited performance impact. However, sometimes it is really not desirable to deteriorate performance and architects would prefer to have a bit less of energy savings and keep the same performance level. For this reason the EDD tradeoff metric is often used in practice.

Looking at the best configuration for EDD in table 4.6(b), an EDD value of 0.67 is achieved which translates into a 25% decrease in energy and a 6% decrease in cycles. In fact this configuration represents a really good choice since it is the only one out of the four that reduces energy significantly while achieving better performance.

As can be seen in table 4.6(a), this configuration shares some features of the fastest one such as a wide pipeline, big instruction queue and large L2 cache. However, the register file is smaller and especially the number of read ports into it. Given the wide pipeline, this reduction is likely to result in energy savings. The instruction cache is very small too, only 8KB, the same value as the configuration leading to the best energy savings. This certainly also allows reducing the energy consumed since the instruction cache is a very frequently accessed structure.

4.6.3 Summary

This section has shown that the benchmark-suite predictor can be used to search the design space for the best performance in terms of cycles, energy, ED or EDD. Using this predictor, the search converges quickly, in less than 10 iterations, for any of the target metrics evaluated. Another big advantage of this predictor is the fact that it can be used for more than searching for a minimum value. For instance constraints on cycles or energy can be added during the exploration. If the constraints or the goal are changed, the search can be reconducted with no additional simulation. This contrasts with typical design space exploration techniques that focus purely on search strategy and need to define the goal and constraints beforehand; any changes imply new simulations need to be run. The next section evaluates how the benchmark-suite predictor performs compared to two state-of-the-art techniques.

instr. mix
% loads
% stores
% control transfers
% arithmetic operations
% fp operations
% shift operations
% string operations
% sse instructions
% other instructions

ILP
32-entry window
64-entry window
128-entry window
256-entry window

register traffic
avg. num. of input operands
avg. degree of use
prob. register dep. =1
prob. register dep. ≤2
prob. register dep. ≤4
prob. register dep. ≤8
prob. register dep. ≤16
prob. register dep. ≤32
prob. register dep. ≤64

data stream strides
prob. local read = 0
prob. local read = 8
prob. local read ≤ 64
prob. local read ≤ 512
prob. local read ≤ 4096
prob. local read ≤ 32768
prob. local read ≤ 262144
prob. local write = 0
prob. local write ≤ 8
prob. local write ≤ 64
prob. local write ≤ 512
prob. local write ≤ 4096
prob. local write ≤ 32768
prob. local write ≤ 262144
prob. global read = 0
prob. global read ≤ 8
prob. global read ≤ 64
prob. global read ≤ 512
prob. global read ≤ 4096
prob. global read ≤ 32768
prob. global read ≤ 262144
prob. global write = 0
prob. global write ≤ 8
prob. global write ≤ 64
prob. global write ≤ 512
prob. global write ≤ 4096
prob. global write ≤ 32768
prob. global write ≤ 262144

footprint
unique 4KB pages accessed (instr.)
unique 32-byte block accessed (instr.)
unique 4KB pages accessed (data)
unique 32-byte block accessed (data)

branch predictability (PPM)
GAg: hist. bits = 4
GAg: hist. bits = 8
GAg: hist. bits = 12
PAg: hist. bits = 4
PAg: hist. bits = 8
PAg: hist. bits = 12
GAs: hist. bits = 4
GAs: hist. bits = 8
GAs: hist. bits = 12
PAs: hist. bits = 4
PAs: hist. bits = 8
PAs: hist. bits = 12

Table 4.7: Microarchitecture-independent features extracted for the features-based predictor.

4.7 Comparison With State-Of-The-Art

The previous sections have evaluated the different parameters of the model and demonstrated its efficiency to search the design space. This section conducts a comparison of the benchmark-suite predictor with two state-of-the-art techniques that can be used to predict a whole benchmark suite.

4.7.1 Feature-Based Predictor

A first and simple approach to reduce the number of simulations when performing the design space exploration of a microarchitecture consists of selecting only a subset of programs from the benchmark suite. In order to select the best representative subset of programs, Eeckhout *et al.* proposed the use of microarchitectural-independent features as a means to characterise programs [Eeck 02]. This contrasts with the model developed in this chapter that directly uses responses from the output space.

Recently, this technique has been applied for predicting performance of any program across different architecture systems [Host 06] using offline training. In this section, this technique was adapted by selecting a subset of programs based on the microarchitectural-independent features listed in table 4.7 extracted from the authors' own tool (MICA v0.1) derived from Pin [Luk 05]. First PCA was applied to the set of features to reduce its number and normalise them. Then using K-Means, the program were clustered into **K** clusters. One program was then selected per cluster and this set of selected programs represents the benchmark suite. ANNs were then built (using **T** =512 training samples) for each of these programs. The predictions made by these ANNs were finally combined using the weights obtained from the clustering (*i.e.* number of programs present in each cluster).

The total cost in terms of simulations for this technique is : $\mathbf{K} \cdot \mathbf{T} + 1 \cdot (\mathbf{N} - \mathbf{K}) = \mathbf{K} \cdot 512 + 1 \cdot (\mathbf{N} - \mathbf{K})$ where **K** is the number of representative programs and **T** (fixed to 512) is the number of training samples required to build the ANNs for each of the **K** programs. The term $1 \cdot (\mathbf{N} - \mathbf{K})$ accounts for the extraction of the features for the programs not kept as representative of the benchmark suite.

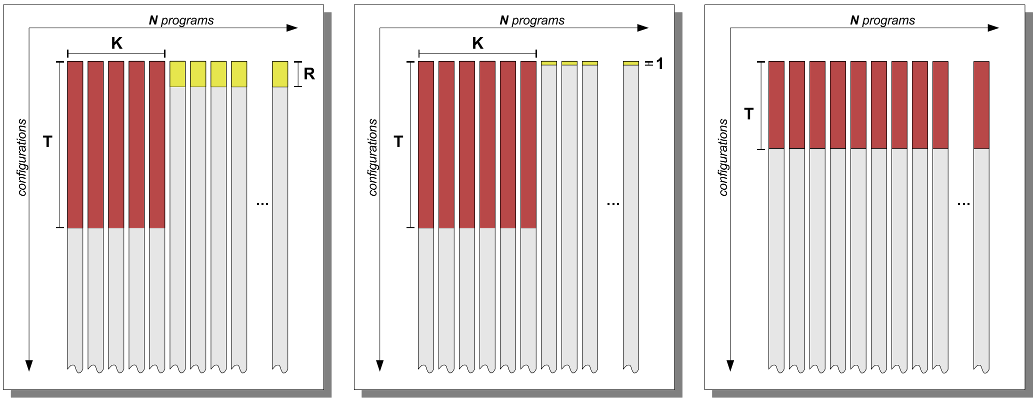
4.7.2 Single-Program Predictors

The second state-of-the-art technique consists simply of constructing a single-program predictor for each benchmark. The scheme proposed by İpek *et al.* [Ipek 06] was chosen, although any other related approaches [Jose 06a, Jose 06b, Lee 06, Lee 07a] could have been used since they were shown as equivalent [Lee 07b]. Since this technique predicts the space of each individual program, the predictions of the individual models were simply averaged to compare with the benchmark-suite predictor.

With this approach all the **N** programs from the benchmark-suite are considered. Hence the only way to vary the number of simulations consists of playing with the **T** parameter that controls the number of samples used to train the ANNs for each program. It follows that the total simulation cost for this technique is : $\mathbf{N} \cdot \mathbf{T}$.

4.7.3 Training Costs

Given a fixed training budget, the distribution of simulations across benchmarks varies for each prediction scheme, as shown in figure 4.25. The benchmark-suite predictor requires **K** programs to be selected, using $\mathbf{R} = 32$ responses from all the **N** programs. For these **K** programs, $\mathbf{T} = 512$ simulations are run for each of them to train the program-specific models. For



(a) Benchmark-suite predictor (b) Features-based model (c) Single-program predictors

Figure 4.25: Share of training simulations between programs for each prediction scheme. For the benchmark-suite predictor, all programs require 32 (**R**) simulations to select **K** representative programs and learn a linear mapping. The chosen **K** programs receive the remaining budget. In the features-based model, one simulation is required to extract features, then the remaining simulations are assigned to the chosen **K** benchmarks. For the single-program predictor, the entire training budget is shared equally between all **N** programs.

the features-based approach, **K** programs are selected using the features extracted from a run of each of the **N** programs. Then the program-specific models are built for these **K** programs, using also $\mathbf{T} = 512$ simulations. In the case of the single-program predictors, all the **N** programs receive a similar simulation budget **T** used to train each individual model.

Table 4.8 summarises for each technique the cost associated in terms of **N**, the number of program in the suite, **K**, the number of representative programs selected and **T**, the number of training samples for the single-program predictors. Since **N** is being fixed by the size of the benchmark suite, only the variables **K** and **T** can be chosen by the user in order to achieve a desired error rate or simulation budget. The last column of the table shows how those variables are set for a total budget of about 3200 simulations. As it can be seen the features-based approach contains one more representative program ($\mathbf{K} = 6$) than the benchmark-suite predictor ($\mathbf{K} = 5$). Since the features-based model does not need to run $\mathbf{R} = 32$ simulations for all the programs, this budget can be used to add an additional representative program.

4.7.4 Predicting The Whole SPEC CPU 2000 Suite

Having determined how to distribute a given simulation budget for the benchmark-suite predictor and the two state-of-the-art schemes, this section now evaluates the accuracy of these

Technique	Total cost	Fixed cost of ∼3200 simulations for N =26	
Benchmark-suite predictor	$512 \cdot \mathbf{K} + 32 \cdot (\mathbf{N} - \mathbf{K})$	**K** =5	$(512 \cdot 5 + 32 \cdot (26 - 5) = 3232)$
Features-based approach	$512 \cdot \mathbf{K} + 1 \cdot (\mathbf{N} - \mathbf{K})$	**K** =6	$(512 \cdot 6 + 1 \cdot (26 - 6) = 3092)$
Single-program predictors	$\mathbf{T} \cdot \mathbf{N}$	**T** =128	$(128 \cdot 26 = 3328)$

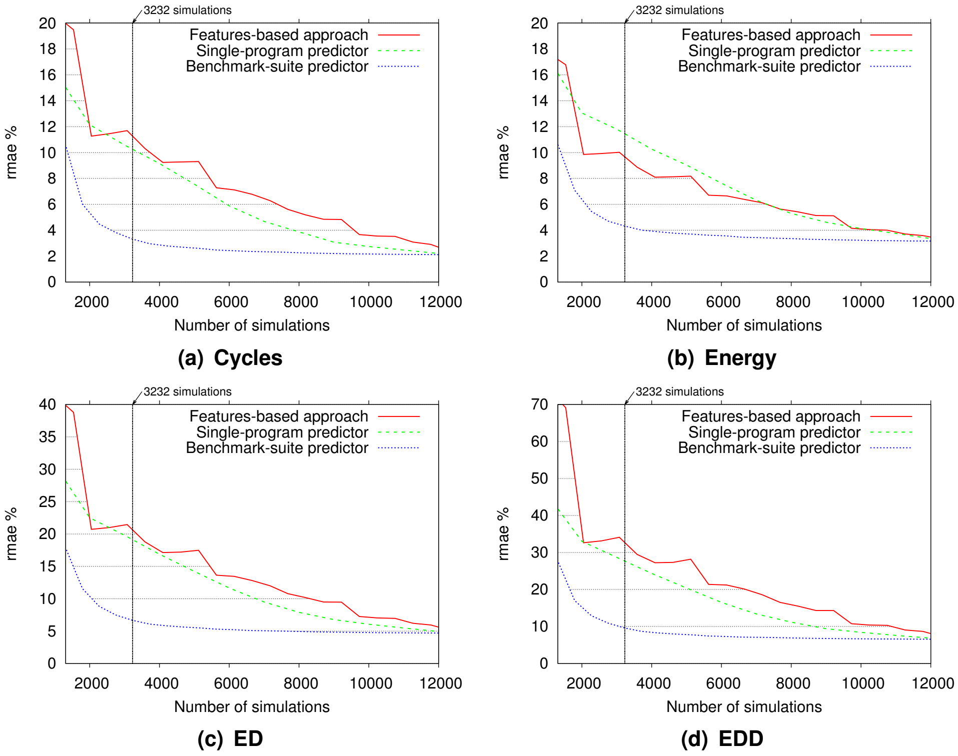
Table 4.8: Simulation costs for each technique. **N** is the total number of programs in the benchmarks suite (26 for SPEC CPU 2000). **K** is the number of representative programs and **T** the number of training samples for each single-program predictor.

models when predicting the whole of SPEC CPU 2000 for the sampled microarchitectural design space.

Figure 4.26 shows the error for each metric as the total number of training simulations is varied. As can be seen, the benchmark-suite predictor achieves the same accuracy as the features-based approach and the single-program predictor using fewer simulations. For example, when predicting ED, it achieves an error of just 6% using 3232 simulations, whereas the two state-of-the-art approaches require more than 10,000 simulations to reach the same accuracy.

The coefficient of correlation of the three different techniques is also shown in figure 4.27 for the different metrics. One can observe that both techniques that focus on reducing the number of simulations by keeping only the representative programs achieve a higher correlation than the single-program predictors technique. For all the metrics, the benchmark-suite predictor outperforms the features-based approach. However, for energy both techniques perform pretty similarly. This means that energy is less sensitive to the way program are being selected but relies much more on having an accurate modelling of the architecture space for each selected programs. Referring back to figure 4.2(b) at the beginning of this chapter, it can be observed that the dendrogram is much more balanced for energy than for the other metrics. In addition the distance between any two programs is smaller on average. This supports the claim that the energy value is less program related than for the other metrics.

In summary, all three techniques were compared based on their mean prediction error and correlation. Even though the features-based technique performs relatively well in terms of correlation, the benchmark-suite predictor always outperforms the two other approaches requiring less simulations and achieving lower error and higher correlation. The next section finally considers the benefit of using the benchmark-suite predictor when predicting benchmark suite of different sizes. It also shows how much savings can be made in terms of simulations.



(a) Cycles

(b) Energy

(c) ED

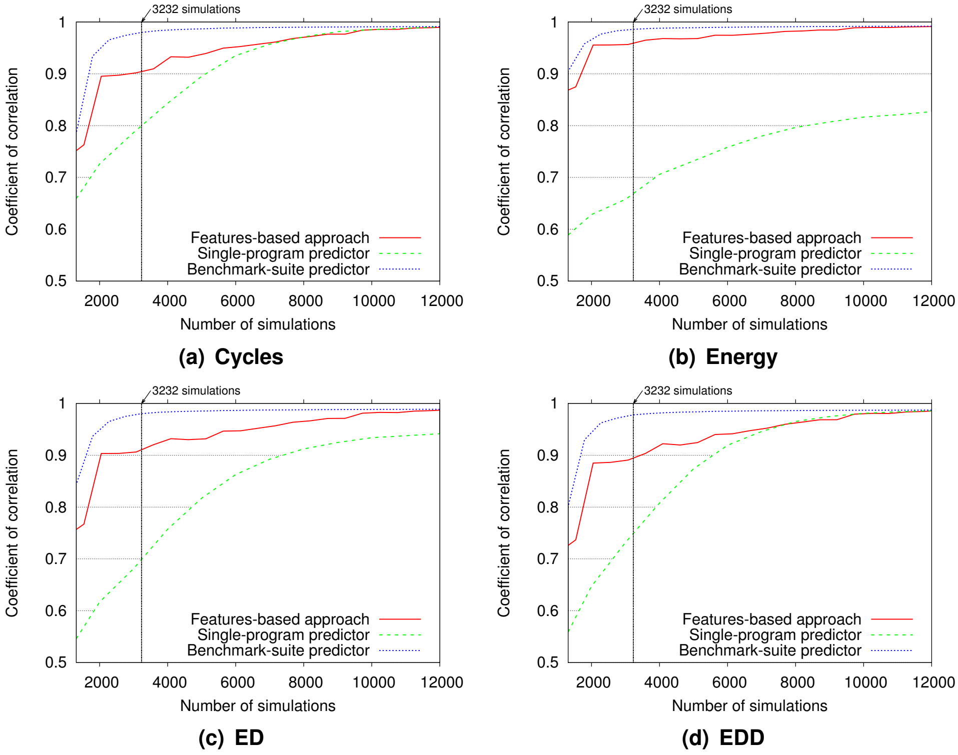
(d) EDD

Figure 4.26: Error as a function of the number of simulations to train both state-of-the-art schemes and the benchmark-suite predictor. This latest technique achieves the same or lower error as the state-of-the-art schemes with half the number of training simulations or fewer.

4.7.5 Larger Benchmark Suites

As seen in the previous section, the benchmark-suite predictor achieves a lower error rate using far less training simulation than the other two schemes. This section now considers how the number of training simulations varies for a fixed error rate, as different benchmark suites are predicted. The error rate was fixed to the values obtained from the benchmark-suite predictor when just $\mathbf{K} = 5$ representative programs were selected. As shown in section 4.7.3 this corresponds to 3232 simulations leading to an error of 3% for cycles, 4% for energy, 6% for ED and 8% for EDD. It is straightforward to fix the budget and pick other error rates if desired. Three benchmark suites of different sizes were chosen to compare the techniques: SPEC CPU 2000 Integer (12 programs), SPEC CPU 2000 (26 programs in total) and SPEC CPU 2000 combined with MiBench (60 programs in total).

Figure 4.28 shows the results using the benchmark-suite predictor and the two state-of-the-art approaches. As can be seen, for small benchmark suites, such as SPECint, all three predictors require a similar number of training simulations to achieve the fixed error rate, al-



(a) Cycles

(b) Energy

(c) ED

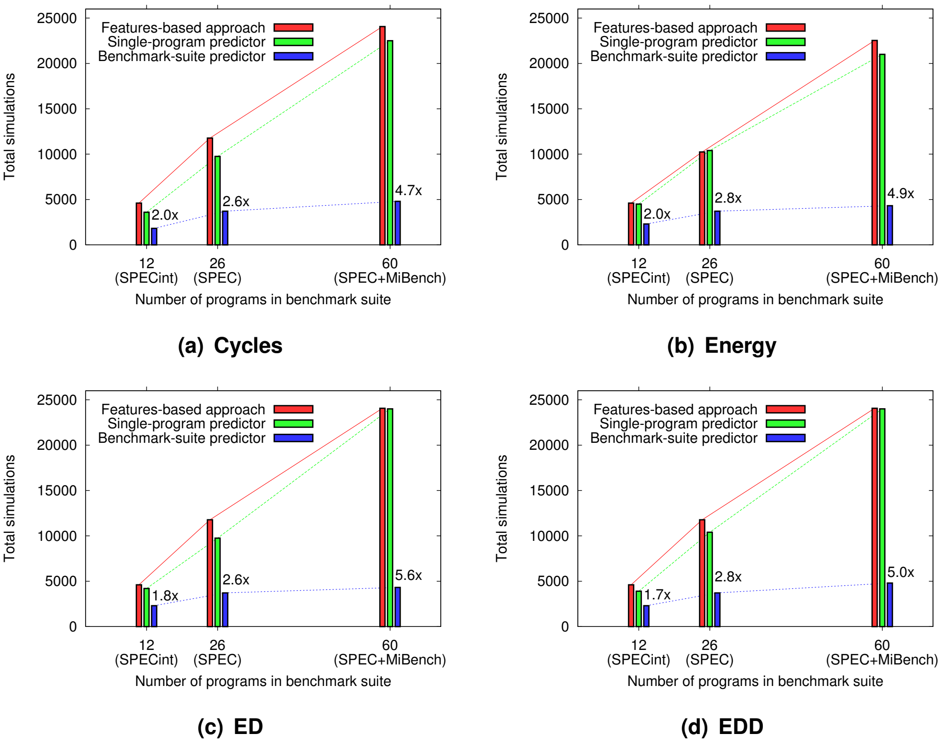
(d) EDD

Figure 4.27: Coefficient of correlation as a function of the number of simulations to train both state-of-the-art schemes and the benchmark-suite predictor.

though the benchmark-suite predictor always requires fewer than the other two approaches. However, for larger benchmark suites, the benefits of this scheme become clear. When predicting cycles for the whole of SPEC, the single-program predictor and features-based approach need 9750 and 11,776 respectively, whereas the benchmark-suite predictor requires just 3712, 2.6 times fewer. When predicting for combined SPEC and MiBench, it requires 4800 simulations compared with 22,500 and 24,064, which is 4.7 times fewer. As the curves in these graphs demonstrate, when moving from 26 to 60 programs, both other schemes require a further 10,000 new simulations whereas the benchmark-suite predictor requires fewer than 1000. This is, asymptotically, an order of magnitude fewer simulations than the state-of-the-art approaches as the benchmark suite size increases.

This asymptotic behaviour is explained by the fact that many programs are similar. Therefore, as large benchmark suites are considered, the number of optimal representative programs **K** tends to stabilise to a constant value; *i.e.* adding more programs does not affect the value of **K** and only requires a further $\mathbf{R} = 32$ simulations as opposed to the other approaches.

The same behaviour would be expected from the features-based approach since it uses a similar strategy of only selecting the relevant programs. However, surprisingly, it does not help



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 4.28: Total simulations needed to train each model for a fixed error when different benchmark suites are considered. The lines represent the trend and the labels at the top of each bar for the benchmark-suite predictor represent the savings compared to the best of the two other approaches. As the number of programs in the benchmark suite increases, all schemes require a greater number of training simulations. However, the benchmark-suite predictor needs an order of magnitude fewer additional simulations than the two other approaches. Moving from 26 to 60 programs, it requires fewer than 1000 new simulations whereas both other schemes require a further 10,000.

to reduce the simulation budget when compared to the single-program predictor. The reason is that it is actually not as accurate at identifying the really important **K** representative programs. Furthermore, it is not able to perfectly account for the dropped programs. This really shows that even if using $\mathbf{R} = 32$ responses to characterise programs comes at a cost in terms of simulations (as opposed to extracting features only once per program), this cost is quickly amortised by the fact that the number of **K** representative programs can be reduced to a strict minimum. Overall, this helps to achieve great savings in the number of simulations.

4.8 Conclusions

This chapter has explored a novel approach for design space exploration by building a model that makes use of program similarities. In the first part, a model was presented that is built offline and makes predictions for any new unseen program using only 32 simulations from it.

This model is used as the foundation for the benchmark-suite predictor: a novel model that predicts the average behaviour of an entire benchmark suite. It was shown that this model dramatically reduces the number of simulations required when compared to two state-of-the-art approaches. Using only five representative programs from SPEC CPU 2000, this model accurately predicts the average behaviour of the full suite for cycles, energy, ED and EDD. Furthermore it was shown that it achieves the same error rate with five times fewer training simulations on the SPEC CPU 2000 and MiBench benchmark suites compared with either of the two other approaches.

The benchmark-suite predictor is a practical model that can be used to conduct efficient design space exploration for general purpose microprocessors. Because it characterises programs using responses, it is able to automatically adapt to any design space and always selects the most representative programs, achieving the biggest reduction in the number of simulations with high accuracy.

The next chapter deals with the design of embedded processors, which involves a different methodology and hence requires other modelling techniques. In particular it focuses on the use of compiler optimisation space exploration at processor design time.

Chapter 5

Exploring and Predicting the Co-Design Space

5.1 Introduction

High performance and low energy consumption in embedded systems are typically achieved through efficient processor design and optimising compiler technology. Fast time-to-market is critical for the success of any new product, therefore it is crucial to design new microprocessors quickly and efficiently. However, during the earliest design stages, architectural decisions must be taken with only limited knowledge of other system components, especially the compiler. Ideally both the architecture and the optimising compiler should be considered simultaneously, selecting the best combination to reach higher levels of efficiency.

Unfortunately exploring this combined design or *co-design* space is extremely time consuming. For each architecture considered, an optimising compiler would have to be built, which is clearly impractical. Instead, a typical design methodology consists of first selecting an architecture under the assumption that the optimising compiler can deliver a certain level of performance and energy efficiency. Then, a compiler is built and tuned for that architecture which will hopefully deliver the performance levels assumed.

Clearly this is a sub-optimal way of designing systems. The compiler team may not be able to deliver a compiler that achieves the architect's expectations. More fundamentally, if one could predict the performance that an eventual optimising compiler could achieve on any architecture, then a completely different architecture may be chosen. This inability to directly investigate the combined architecture/optimising compiler interactions means tomorrow's architectures are being designed based on yesterday's compiler technology.

In this chapter a novel approach is proposed to address this co-design space problem. A machine-learning model is built that can automatically *predict* the performance of an optimising compiler across an arbitrary architecture space *without* tuning the compiler first. This allows the designer to accurately determine the performance of any architecture as if an optimising compiler were available. Given a small sample of the architecture and optimisation space, this model can then predict the performance of a yet-to-be-built optimising compiler using information gained from a non-optimising baseline compiler. Ultimately, this has the potential to drive a change in the current methodology of designing embedded processors.

The first section of this chapter, section 5.2, presents the experimental setup. It is followed by an exploration of the microarchitecture design space in section 5.3. This exploration is conducted without considering compiler optimisations, as is typically the case in current methodology. In section 5.4 the optimisation space is then considered in isolation from the architecture space for a fixed baseline architecture. This section shows that substantial gains can be achieved when carefully tuning the compiler for a specific microarchitectural configuration. Section 5.5 then investigates the combined co-design space of both the microarchitecture and the compiler optimisations. It shows that optimisations play a critical role and can greatly influence the designer's decisions. Because exploring the co-design space is not feasible, a practical solution is needed to make it possible for the designer to consider the effects of compiler optimisations at design time. Section 5.6 presents a machine-learning model that can predict ahead of time the performance that a tuned compiler would achieved on any microarchitecture configuration. This model is evaluated later in section 5.7 which shows that it can indeed be used to perform an accurate and efficient co-design space exploration of the microarchitecture and compiler space. Finally section 5.8 concludes this chapter.

5.2 Experimental Setup

This section discusses the architecture and simulator used, the benchmarks selected and the chosen compiler infrastructure. In particular, it shows that the setup is realistic and corresponds to an existing embedded processor with a compiler actively used by industry.

5.2.1 Architecture and Simulator

To evaluate the effectiveness of co-design space exploration and its impact on processor design, the Intel XScale processor [Inte 02] was chosen as the baseline architecture. This processor is typically found in embedded systems and its configuration is shown in table 5.2, column 3. Section 5.5.1 shows that this is in fact a well balanced design for energy and execution time.

The XScale processor is an implementation of the ARM architecture (ARMv5TE). The ARM family of processors are specifically targeted at embedded systems and strongly dominate the market of embedded RISC microprocessor. An interesting fact about ARM is that the company itself does not directly produce processors but instead sells the IP. This enables the customers to make customisations to suit their needs.

For these reasons, choosing an XScale processor to study the typical design of an embedded processor makes sense. Furthermore, a simulator that is validated for cycles and energy exists for this Intel XScale processor; the Xtrem simulator [Cont 04]. This simulator was slightly altered to allow detection of library code. Since it is desirable to run the programs unmodified on the simulator, it was also extended in order to skip all IO related library calls (such as printf). Indeed many of these IO functions are present in the benchmarks for debugging purpose or to read the input from a file to the main memory for instance. These IO calls were ignored to remove their performance impact on the total program execution. In addition to these modifications, the access latencies of each cache configuration were modelled using Cacti [Tarj 06] to ensure the experiments were as realistic as possible.

With this simulation setup, the exploration of the microarchitectural, compiler and combined design spaces can be performed using execution time (cycles), energy, the energy-delay ED and EDD. These are the same target metrics used in the previous chapter.

5.2.2 Benchmark Suite

The design or customisation process of an embedded processor is directly affected by the set of applications that is going to be run on the final product. The designer can target these applications. In fact this is exactly what allows embedded processors to achieve energy efficiency and performance compared to general purpose machines; the choice of benchmarks dramatically influences the design of the processor.

The full MiBench [Guth 01] benchmark suite was chosen to evaluate the performance of the selected embedded architecture. As seen in the previous chapter, MiBench specifically targets embedded systems and is representative of the type of programs that one might want to run on these systems. All 35 programs from the suite were run to completion. For each benchmark the input size was chosen so that a maximum of 100 million instructions were executed whenever possible.

Table 5.1 shows the input chosen for each benchmark, the number of instructions simulated in cycle-accurate mode (everything except IO) and the percentage of instructions spent in library code. Because the compiler cannot change the library code in this setup, it is important to know how much of the program can be tuned. As it can observed, some program such as

Program	Input	Sim. instr.	% lib code
basicmath	small	9m	**77**
bitcnts	small	45m	0
qsort	small	32m	**97**
susan_c	large	22m	3
susan_e	large	52m	0
susan_s	small	18m	4
cjpeg	small	31m	2
djpeg	large	24m	2
lame	small	116m	14
madplay	small	25m	0
tiff2bw	small	34m	18
tiff2rgba	large	31m	36
tiffdither	small	358m	1
tiffmedian	small	148m	8
lout	small	56m	14
dijkstra	small	43m	4
patricia	small	9m	**72**
ghostscript	small	75m	5
ispell	small	8m	27
say	small	59m	8
search	large	1m	14
bf_d	small	26m	0
bf_e	small	26m	0
pgp	*NA*	1m	8
pgp_sa	*NA*	94m	0
rijndael_d	small	23m	0
rijndael_e	small	23m	0
sha	small	14m	3
rawcaudio	small	37m	0
rawdaudio	small	27m	0
crc	small	18m	0
fft	small	5m	**60**
fft_i	small	11m	**58**
toast	small	29m	0
untoast	small	16m	0

Table 5.1: The 35 MiBench benchmarks used with their corresponding input size, the total number of instructions executed compiled with **-O1** and the percentage of library code executed.

basicmath, *qsort*, *patricia*, *fft* and *ffti* rely heavily on library code. Therefore little improvement is expected when the compiler optimisations will be considered for these programs.

5.2.3 Compiler

Gcc version 4.1.0 was chosen as the baseline compiler since it has a backend for the ARM architecture. Moreover this is a well tuned compiler that is widely used within industry. In the experiments, all compiler optimisations were enabled from the command line by using the flags available.

Baseline Optimisation Level A baseline optimisation level needs first to be selected in order to make comparison possible between different microarchitectures and optimisations. For that purpose the three default optimisation levels available in *gcc* were evaluated, namely **-O1**, **-O2** and **-O3**.

Figure 5.1 shows the performance, energy consumption, ED and EDD per benchmark for each of the optimisation levels, normalised to **-O1**. As can be seen, the optimisation levels **-O2**

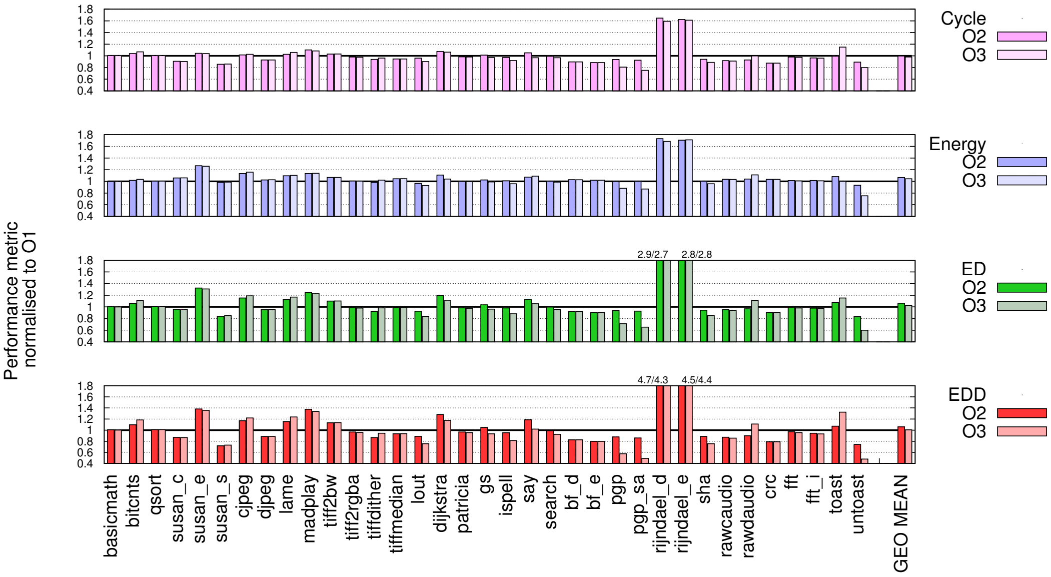


Figure 5.1: Cycles, energy, ED and EDD of each MiBench programs when compiled with **-O2** and **-O3**, normalised by **-O1** (lower than 1 means better than **-O1**).

and **-O3** affect each benchmark in varying degrees. However, surprisingly, on average they both produce the same execution time as **-O1**. There is similar variation for energy, although on average there is higher consumption when using **-O2** or **-O3**. When the tradeoffs between performance and energy, ED and EDD, are considered, it is clear that **-O1** represents the best choice. Hence, **-O1** was chosen as the baseline optimisation level. Note that the conclusions drawn in this chapter are independent of this choice.

Optimising Compiler Since this chapter deals with the notion of "optimising compiler", it is important to define what is meant by an optimising compiler in this context. Given the different optimisations implemented in the compiler and the parameters that control if and how they should be applied, an optimising compiler is defined as a compilation strategy that always select the set of parameters that lead to the best possible performance for a given program on a given microarchitecture. Here performance is used as a generic term that might refer to execution time, energy or any tradeoff of these two metrics.

This definition of an optimising compiler matches the reality of the development of embedded systems. Because it is very expensive to develop a new compiler infrastructure for each new architecture being designed, the typical methodology involves "recycling" existing compilers. Once the architecture is developed, the "existing" compiler is in turn tuned. This involves changing the heuristics and their parameters that control the compilation process.

Because of the large number of possible sets of parameters, it is infeasible practically to exhaustively enumerate all the possible combinations of these parameters. Instead, in this work the choice was made to apply iterative compilation with 1000 randomly-selected flag combinations on each microarchitecture. The best out of these 1000 compilations/runs is expected to be very close to the real best. In fact going beyond 1000 compilations achieves insignificant improvements compared to the best value found with 1000. Therefore, in this thesis an optimising compiler is defined as a compiler that is able to select the best set of flags based on 1000 runs each with a different random combination of flags.

5.2.4 Sample Space

To perform the experiments, 200 microarchitectural configurations and 1000 compiler optimisations were selected from the combined design space using uniform random sampling. In total, for 35 benchmarks, 7 million simulations ($35 \times 200 \times 1000$) were run to create this sample space. The actual microarchitectural space considered is described in the next section while section 5.4 describes the compiler optimisation space.

5.3 Microarchitecture Design Space

Current microprocessor design methodology involves choosing and tuning a microarchitecture whilst developing the compiler independently. This section first presents and analyses the microarchitecture space. Later section 5.4 presents the compiler optimisation space independently of the microarchitectural space. Finally, these two spaces are merged and considered together in section 5.5.

5.3.1 Microarchitectural Parameters

The parameters of the microarchitectural design space are shown in table 5.2. Also shown is the range of values each parameter can take and the baseline microarchitecture which is based on the configuration of the XScale processor [Inte 02]. These parameters were chosen because caches and branch prediction configurations are typical parameters that can be adapted easily without involving a complete redesign of the architecture.

The total design space thus consists of 288,000 different configurations. A sample space of 200 randomly selected configurations was selected to represent the total space. To evaluate the architecture space independently from the compiler space, each benchmark was compiled using the baseline optimisation (**-O1**).

Parameter	Low → High	Baseline
ICache size	4K → 128K	32K
ICache associativity	4 → 64	32
ICache block size	8 → 64	32
DCache size	4K → 128K	32K
DCache associativity	4 → 64	32
DCache block size	8 → 64	32
BTB size	128 entries → 2048 entries	512 entries
BTB associativity	1 → 8	1

Table 5.2: Microarchitectural parameters and the range of values they can take. Each parameter varies as a power of two, with 288,000 total configurations. Also shown are the baseline values.

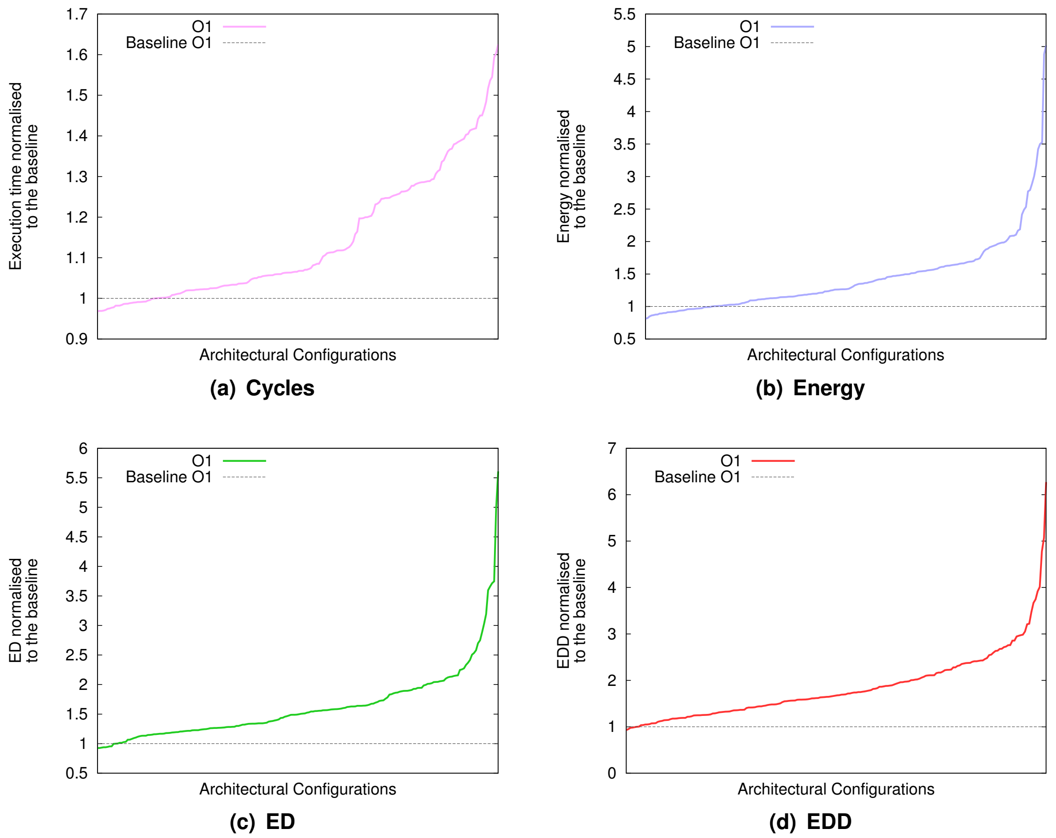
5.3.2 Microarchitecture Exploration

Figure 5.2 shows the microarchitectural design space. Each graph shows the performance achieved by each microarchitectural configuration in terms of execution time, energy, ED and EDD across the MiBench suite, normalised to the baseline architecture. The baseline performance is shown with a horizontal line. Each graph is independently ordered from lowest to highest. These graphs show that the baseline architecture is actually a very good choice. For both execution time and energy consumption it is within the top 15% of all configurations, for ED it is within the top 5% and within the top 2% for EDD. This is not suprising, since this baseline architecture corresponds to the XScale processor and has already been highly tuned. In fact this shows that the experimental setup is realistic.

Conversely, there are many configurations that are worse than the baseline. In terms of execution time, the worst configuration is 60% slower than the baseline and for energy there is one configuration which consumes 70% more than the baseline. The worst ED value achieved is 2.7 and the worst EDD is 6.3. This shows that the space considered varies and is not flat. Whilst these "bad" configurations result in poor performance or high energy consumption, they might none the less be interesting for the designer. Indeed other parameters not modelled in this setup such as cost, area or core frequency to cite a few, might make them good candidates for implementation.

5.3.3 Best Microarchitecture

This section looks at the characteristics of the best configurations in the space for each of the target metrics. The best architectures found are shown in table 5.3 where the configuration parameters and the best values achieved are presented. On average, a modest ED value of 0.93



(a) Cycles **(b) Energy**

(c) ED **(d) EDD**

Figure 5.2: The average execution time, energy, ED and EDD of each microarchitectural configuration across the whole benchmark suite. Each graph is independently ordered from lowest to highest and is normalised by the baseline configuration.

and an EDD value of 0.93 can be achieved over the baseline XScale architecture as shown in table 5.3(b).

For EDD, an improvement of 10% for cycles and 5% for energy is possible. As can be seen from table 5.3(a), on the one hand the configuration leading to the lowest number of cycles has large instruction and data caches compared with the baseline. This allows better performance since the number of misses can be reduced. On the other hand when optimising for energy, the caches are much smaller. The instruction cache is larger than the data cache to avoid too many misses that hurt performance, which in turn implies more energy consumed due to the longer running time of the program. Interestingly, the best configuration when optimising for ED is the same as for energy. This configuration hence has the lowest energy consumption, 19% less than the baseline but sacrifices 14% of performance. When considering the best configuration for EDD, it improves performance by 10% and reduces energy consumption by 5%. This configuration has the same instruction cache size as the baseline but a slightly larger

Configuration	Icache			DCache			BTB	
	size	**assoc.**	**block**	**size**	**assoc.**	**block**	**size**	**assoc.**
Baseline	32K	32	32	32K	32	32	512	1
Best cycles	128K	32	64	128K	4	64	128	1
Best energy	16K	64	16	8K	64	16	128	1
Best ED	16K	64	16	8K	64	16	128	1
Best EDD	32K	64	16	64K	32	32	256	4

(a) Microarchitectural parameters

Configuration	Cycles	Energy	ED	EDD
Baseline	1.00	1.00	1.00	1.00
Best cycles	**0.97**	2.09	2.02	1.96
Best energy	1.14	**0.81**	0.93	1.06
Best ED	1.14	0.81	**0.93**	1.06
Best EDD	0.90	0.95	0.94	**0.93**

(b) Best values achieved

Table 5.3: The microarchitectural parameters of the baseline and the best configurations found from the sampled space with their corresponding values for cycles, energy, ED and EDD.

data cache. The branch target buffer is smaller than the baseline but has increased associativity to prevent conflicts.

5.3.4 Details Per Program

Having identified the best microarchitecture for each target metric, this section now considers its performance broken down per program for the MiBench benchmark suite. This is in fact very important in the embedded world since the architectures are typically targeted to a specific set of applications.

Figure 5.3 shows the microarchitectures achieving the best execution time, energy, ED and EDD value for all benchmarks, normalised to the baseline average architecture. These were picked for each metric over the whole MiBench suite with each benchmark compiled and run with the baseline optimisation **-O1**.

In terms of execution time, all the benchmarks achieve better performance than the baseline. However, the gains are relatively small for most of the programs. Considering energy, the majority of benchmarks achieve a 20% savings over the baseline configuration. For ED, the value for some benchmarks is over 1 because this configuration actually loses performance for these benchmarks. In this setup the best average architecture is not being specialised for each

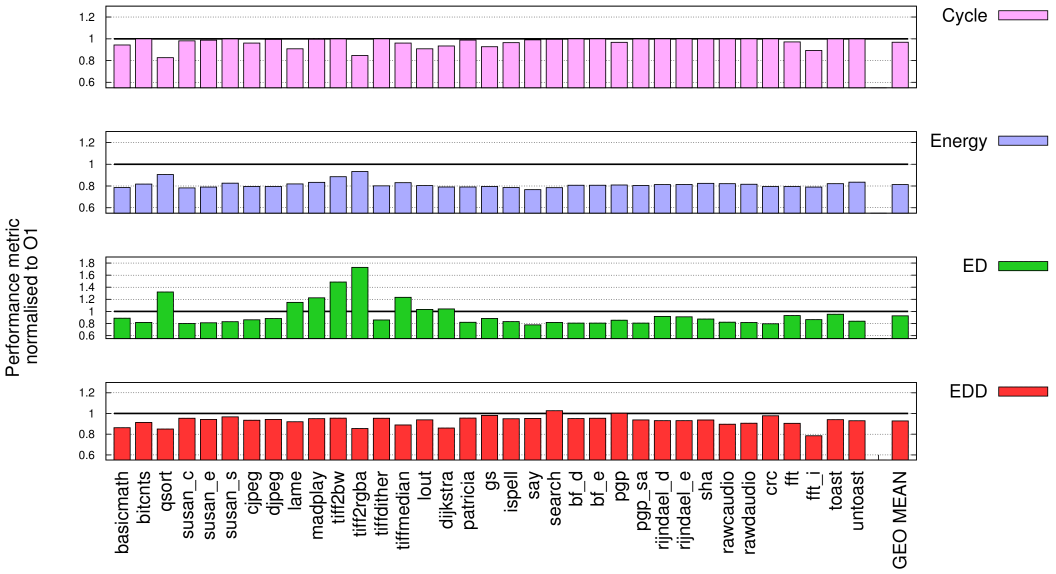


Figure 5.3: Execution time, energy, ED and EDD for each benchmark on the microarchitectural configuration performing best for each metric.

program, so this configuration that is the best for ED overall, is not necessarily the best for each program. When it comes to EDD, the best configuration outperforms the baseline for most of the programs and performs only slightly worse for a couple. This means it is possible to find a better configuration than the baseline that consistently achieves a better tradeoff between performance and energy.

5.3.5 Summary

In this section, a design space exploration of the microarchitecture was conducted. It showed that the baseline configuration which corresponds to the XScale processor is actually a very good design, not surprisingly since it has been highly tuned by architects. However, it is possible to find a better configuration that achieves, for the vast majority of the programs, higher level of performance and lower energy consumption. In the next section, the compiler optimisation space of the baseline architecture is considered independently.

5.4 Compiler Optimisation Space

Having considered the microarchitecture design space in isolation, this section conducts an exploration of the compiler optimisation space only. It shows the characteristics of the compiler space when optimising for the baseline architecture.

N°	Flag
1	-fthread-jumps / ⊘
2	-fcrossjumping / ⊘
3	-foptimize-sibling-calls / ⊘
4	-fcse-follow-jumps / ⊘
5	-fcse-skip-blocks / ⊘
6	-fexpensive-optimizations / ⊘
7	-fstrength-reduce / ⊘
8	-frerun-cse-after-loop / ⊘
9	-frerun-loop-opt / ⊘
10	-fcaller-saves / ⊘
11	-fpeephole2 / ⊘
12	-fregmove / ⊘
13	-freorder-blocks / ⊘
14	-falign-functions / ⊘
15	-falign-jumps / ⊘
16	-falign-loops / ⊘
17	-falign-labels / ⊘
18	-ftree-vrp / ⊘
19	-ftree-pre / ⊘
20	-funswitch-loops / ⊘

N°	Flag	Values
21	-fgcse / ⊘	
22	-fno-gcse-lm / ⊘	
23	-fgcse-sm / ⊘	
24	-fgcse-las / ⊘	
25	-fgcse-after-reload / ⊘	
26	–param max-gcse-passe =	1, 2, 3, 4
27	-fschedule-insns / -fschedule-insns2 / ⊘	
28	-fno-sched-interblock / ⊘	
29	-fno-sched-spec / ⊘	
30	-finline-functions / ⊘	
31	–param max-inline-insns-auto =	10,30,50,...,190
32	–param large-function-insns =	1300,1500,1700,...,3300
33	–param large-function-growth =	20,50,100,200,300,400,500
34	–param large-unit-insns =	4000,6000,8000,...,20000
35	–param inline-unit-growth =	10,20,30,...,100,200,300
36	–param inline-call-cost =	10,12,14,...,30
37	-funroll-loops / -funroll-all-loops / ⊘	
38	–param max-unroll-times =	2,4,6,...,20
39	–param max-unrolled-insns =	50,75,100,...,400

Table 5.4: Compiler optimisations and the values they can take. There are 642 million combinations. The baseline is **-O1** with no further optimisations enabled.

5.4.1 Compiler Flags

The compiler flags used to generate the optimisation space are shown in table 5.4. The space consists of all the possible combinations of these flags, corresponding in fact to the different optimisation passes within the *gcc* compiler. In addition a few parameters control how some passes are performed. For instance, *max-unroll-times* controls what is the maximum number of times a loop might be unrolled if the corresponding flag *funroll-loops* is enabled. This space is in fact similar to the optimisation spaces considered by other researchers [Vasw 07], allowing meaningful comparisons with existing work. This optimisation space has 642 million different combinations of flags, either turned on or off. When combined with the parameters that control the behaviour of the optimisation passes, this gives a total of 1.69×10^{17} unique optimisation settings.

Since exhaustive enumeration of this optimisation space is not feasible, 1000 different optimisations were chosen using uniform random sampling. The benchmarks were then compiled

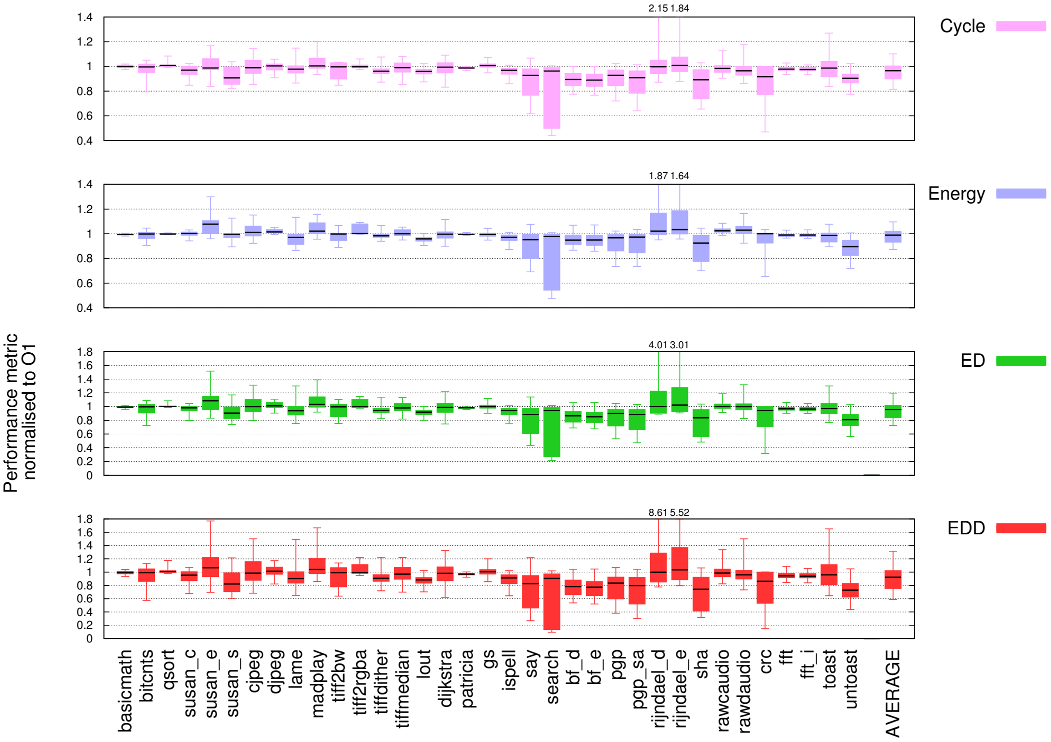


Figure 5.4: Distribution of the compiler optimisation design space on a per-benchmark basis for execution time, energy, ED and EDD (the lower the better). The minimum, maximum, median, 25% and 75% quartiles are shown.

with these flags and run on the baseline architecture. As stated earlier in section 5.2.3, an optimising compiler is defined as a compiler that is able to select the best set of flags based on 1000 runs with different random combinations of flags.

5.4.2 Compiler Optimisation Exploration

To investigate the potential for improvement in the optimisation space, an analysis was conducted on the baseline architecture for each program based on the run from the 1000 random optimisations applied. Figure 5.4 shows for each program different statistics normalised to the baseline optimisation **-O1** for execution time, energy, ED and EDD values. In these graphs the minimum, maximum, median, 25% and 75% quantiles are presented. Also shown in the final column is the average of these statistics across all benchmarks.

What is immediately clear is that for some benchmarks there are significant improvements to be gained in execution time over the baseline optimisation (*e.g. search* at 0.44 and *crc* at 0.47). This also shows that picking the wrong optimisations can significantly degrade performance or increase energy consumption (the worst optimisation on program *rijndael_d* more than doubles its execution time).

However, for some programs on this baseline architecture, the compiler can do little to improve the performance or save energy. Programs *basicmath*, *qsort* or *patricia* for instance have almost no improvement available over the baseline optimisation. Referring back to table 5.1 the reason is obvious since a large percentage of instructions executed belong to library code for these programs; code that is not affected by the optimisations in this experimental setup.

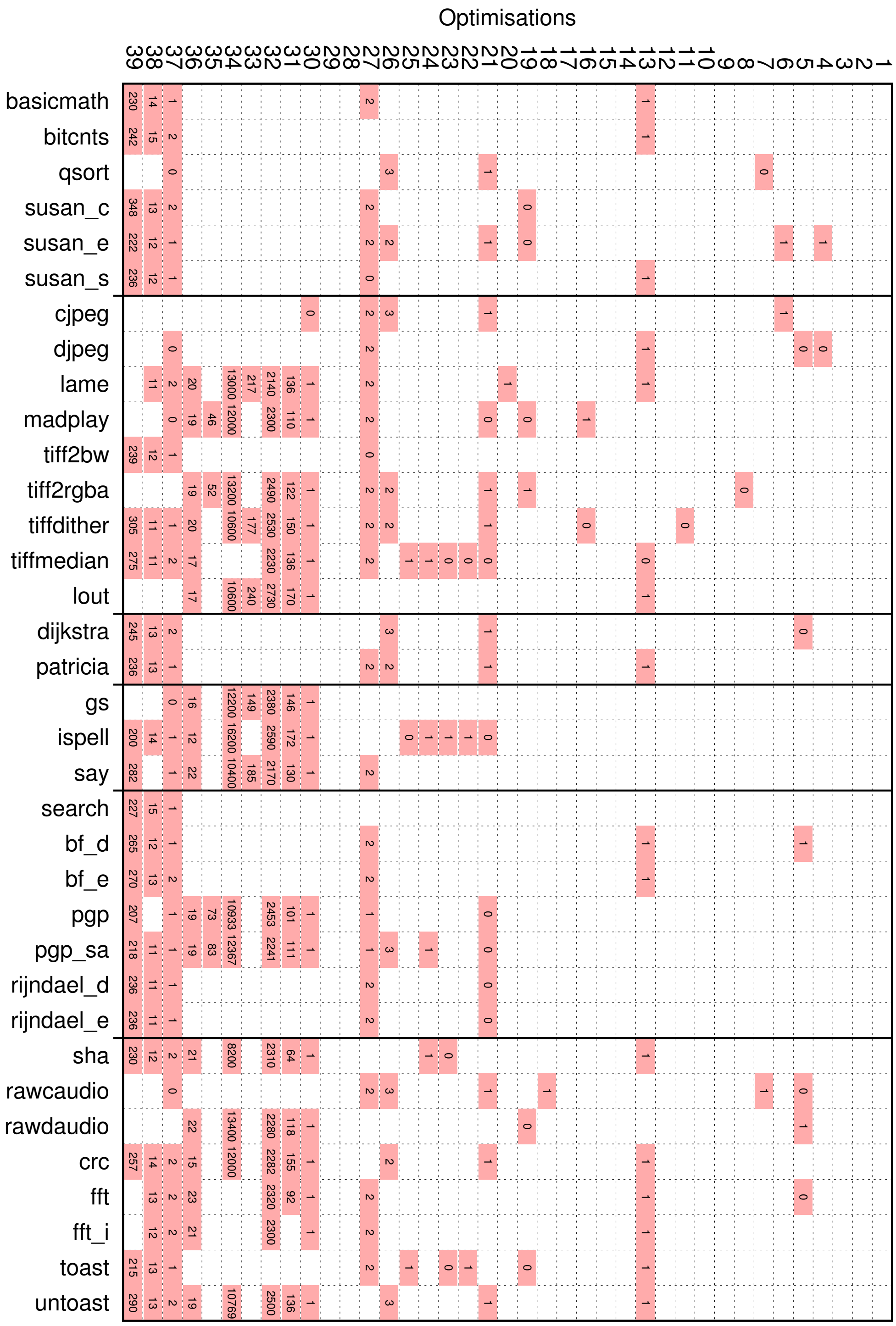
On average, selecting the right optimisations settings for each program means that execution time can be reduced by 19%, energy consumption by 13%, and an ED value of 0.72 or an EDD value of 0.60 can be achieved. This is compared to the best EDD value of 0.93 when varying the microarchitectural space alone. Not surprisingly, there is more room for improvement in the compiler space because the baseline architecture selected has already been significantly tuned.

5.4.3 Different Optimisations For Each Program

Knowing the performance of the best flags for each program is good but it is also important to see whether these flags are actually different from programs to programs. To understand what are the optimisations parameters that have an impact on performance and energy, each program was taken individually and optimised for EDD. Then the optimisation settings were ranked from the best to the worst EDD values and the ones within the top 5% of the best (100% being the performance of **-O1**) were retained. The first ten best were always retained independently of the performance to ensure there were enough flag settings to conduct the analysis.

For all the boolean flags, the flags were marked as important if they happened to be turned on or off at least 90% of the time within the set of the top 5%. For the parameter flags such as *max-unroll-times*, the flags were marked as important if the standard deviation within the top 5% was significantly lower than the standard deviation across all the settings. The value reported in this case was simply the mean of the parameters present in the top 5%.

Figure 5.5 shows the value of the important flags for each benchmark individually since the best combination of optimisation flags varies from program to program. As can be seen the importance of the flags and their corresponding values are dependent on the program for some of the flag settings. For instance consider optimisation number 27 (*f-schedule-insns*): for program *susan_s* it is better to disable this optimisation, whereas for program *pgp* it is better to enable it with instruction scheduling policy 1 and for program *rawcaudio* with policy 2. These two policies influence when the instruction scheduling will be performed (before



	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
basicmath	230	14	1										2														1												
bitcnts	242	15	2																								1												
qsort			0											3					1														0						
susan_c	348	13	2										2								0																		
susan_e	222	12	1										2	2					1		0													1		1			
susan_s	236	12	1										0														1												
cjpeg										0			2	3					1															1					
djpeg			0										2														1								0	0			
lame		11	2	20		13000	217	2140	136	1			2							1							1												
madplay			0	19	46	12000		2300	110	1			2						0		0			1															
tiff2bw	239	12	1										0																										
tiff2rgba				19	52	13200		2490	122	1			2	2					1		1											0							
tiffdither	305	11	1	20		10600	177	2530	150	1			2	2					1					0					0										
tiffmedian	275	11	2	17				2230	136	1			2		1	1	0	0	0								0												
lout				17		10600	240	2730	170	1																	1												
dijkstra	245	13	2											3					1																0				
patricia	236	13	1										2	2					1								1												
gs			0	16		12200	149	2380	146	1																													
ispell	200	14	1	12		16200		2590	172	1					0	1	1	1	0																				
say	282		1	22		10400	185	2170	130	1			2																										
search	227	15	1																																				
bf_d	265	12	1										2														1								1				
bf_e	270	13	2										2														1												
pgp	207		1	19	73	10933		2453	101	1			1						0																				
pgp_sa	218	11	1	19	83	12367		2241	111	1			1	3		1			0																				
rijndael_d	236	11	1										2						0																				
rijndael_e	236	11	1										2						0																				
sha	230	12	2	21		8200		2310	64	1						1	0										1												
rawcaudio			0										2	3					1			1											1		0				
rawdaudio				22		13400		2280	118	1											0														1				
crc	257	14	2	15		12000		2282	155	1				2					1								1												
fft		13	2	23				2320	92	1			2														1								0				
fft_i		12	2	21				2300		1			2														1												
toast	215	13	1										2		1		0	1			0						1												
untoast	290	13	2	19		10769		2500	136	1				3					1								1												

Figure 5.5: Important flags and their corresponding values for the baseline architecture that leads to the best EDD value for all the MiBench programs. As can be seen the optimisations that lead to the best EDD are different from programs to programs.

register allocation in one case and after in the other). This clearly shows that the flag settings that achieve the best EDD are different from program to program.

5.4.4 Summary

This section has considered the compiler optimisations space and the benefits of optimising each program individually on the baseline architecture. In terms of EDD, the best optimisations selected on a per-program basis lead to an EDD value of 0.6 on average for the benchmark suite. This is compared to the best EDD value of 0.93 when varying the microarchitectural space alone. Not surprisingly, there is more room for improvement in the compiler space because the baseline microarchitecture was already significantly tuned. It was also demonstrated that to achieve such a value, the optimisations selected are different from program to program. This clearly shows that tuning a compiler is not an easy task. In fact chapter 6 will later develop a technique that automatically generates an optimising compiler that works across programs and the microarchitecture space.

Up to this point, each space has been considered independently. The next section will combine the microarchitectural and optimisation spaces to consider the co-design of the microarchitecture and compiler optimisations spaces.

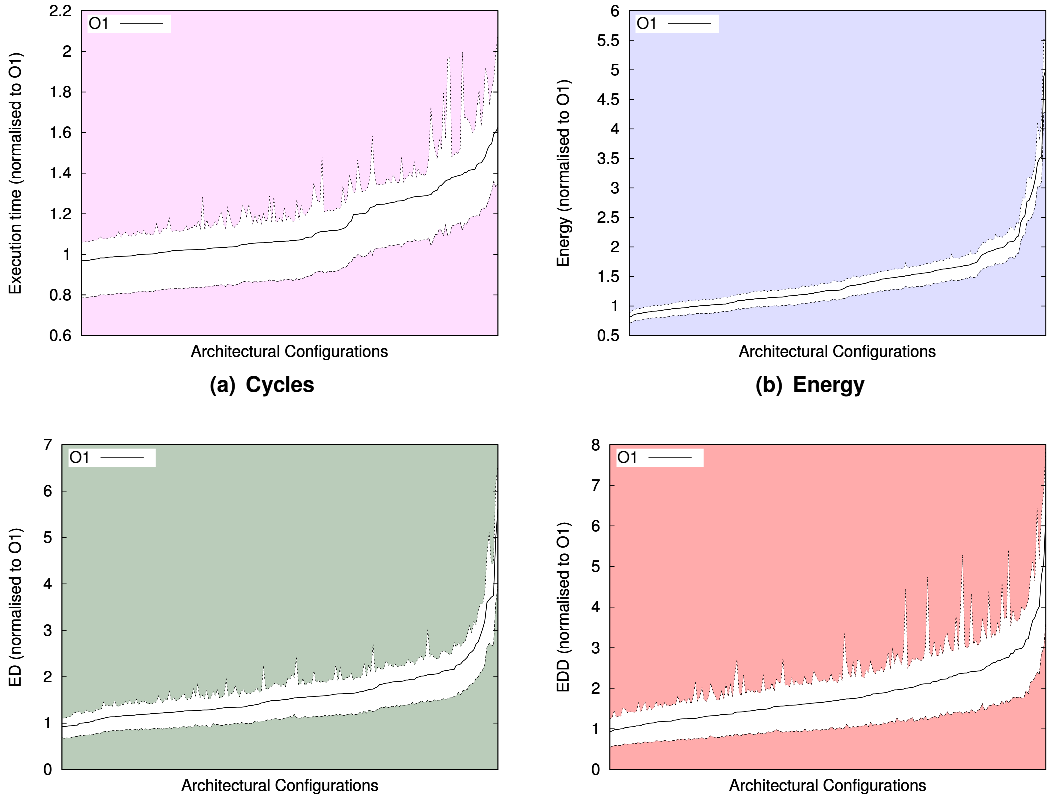
5.5 Co-Design Space Exploration

As seen so far, the design of a new microarchitecture typically involves two separate phases; the microarchitecture design space exploration, followed by the tuning of the compiler. However, this approach can potentially lead to suboptimal designs and, more importantly, to erroneous design decisions. Indeed, since only a basic compiler is available to the architect at design time, the real design space resulting from the potential improvement of compiler optimisations is often hidden from the designer.

This section explores the properties of the combined space. It shows that in fact compiler optimisations can radically change the shape of the design space, especially when the wrong set of optimisations are applied. In addition to the previous section which has shown how the optimal compiler flag settings change from program to program, this section shows that the choice of optimal flag settings is also influenced by the characteristics of the microarchitecture.

5.5.1 Exploration of the Combined Space

Figure 5.6 shows the co-design space across microarchitectural configurations for execution time, energy, ED and EDD. The performance of the baseline compiler on each configuration



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 5.6: The co-design space for execution time, energy, ED and EDD for each microarchitectural configuration across the whole benchmark suite considering the best and worst optimisations for each program. The region in white is the co-design space, with the line showing the performance of **-O1** on each architecture. Each graph is independently ordered from lowest to highest and is normalised by **-O1** on the baseline configuration.

is shown by the solid line (**-O1**). The performance of the optimising compiler on each configuration is also shown. Here the best compiler optimisations on a per-program basis are selected for each microarchitectural configuration from the sample space. This represents the lower bound on the execution time, energy consumption, ED or EDD achievable for each architecture within the sampled space. Also shown is the performance when selecting the worst compiler optimisations which represents the upper bound.

There is large room for improvement over the baseline compiler optimisation across the whole microarchitectural space in terms of execution time, ED and EDD. All four graphs show that picking the wrong optimisations can lead to significant degradation in each metric. Therefore, knowing the performance of the optimising compiler for each architecture is of primary importance for designers. It is interesting to notice that for energy, the compiler optimisations only play a minor role compared to the other metrics. The reason is that compiler optimisations have less of an impact on the energy consumption, which is much more dependent on the microarchitecture characteristics.

According to these results, it is apparent that the compiler optimisations have a critical role to play at processor design time. Indeed, if no exploration of the optimisation space is conducted, the designer risks making a bad decision. Figure 5.6 shows that the design space can be anywhere in the white area, depending on the default optimisations applied. If the architect has some goals, such as a given minimum performance to achieve, it is easy to see how he can be misled when only the microarchitecture space is considered. Based on the performance of the baseline compiler optimisations, **-O1**, some configurations might be discarded where in fact they could perfectly fulfil the requirements, had the compiler optimisation been considered. It might even get worse if other constraints are considered such as area, since these discarded configurations might in fact represent better choices.

Before looking in detail at how this problem can be addressed without having to explore the full co-design space, the next section considers the best points in the sample space. It shows what are the characteristics of these good design points and how sensitive the optimisations are to the microarchitectures.

5.5.2 Best Microarchitecture

Looking at the best configuration in the co-design space for each of the target metrics, it can be seen in table 5.5 that significant improvements can be made over the baseline architecture with the programs compiled with the baseline compiler (**-O1**). This table shows these best configurations and which values they achieved for each target metric. Also shown in parenthesis is the performance achieved for the best configurations for each metric when only the baseline compiler is considered (**-O1**). The performance of the configurations with the optimal compilation settings are always better than when the default compiler optimisations are considered (values in parenthesis). For instance, an EDD value of 0.55 is achieved when both the compiler optimisations and the microarchitectures spaces are explored as opposed to 0.93 when only the architecture space is considered.

Looking at the details for cycle and energy for this same configuration (best EDD), a reduction of 19% for cycle and a savings of 16% in energy is achieved. This is significantly different from the 10% reduction in cycle and 5% in energy savings when only the microarchitecture space was considered. With this example, it is easy to see how the exploration of the co design space can affect the architect's decision compare to an exploration of the microarchitecture only. It is also interesting to notice that when the ED metric is considered, the parameters of

Configuration	**Icache**			**DCache**			**BTB**	
	size	**assoc.**	**block**	**size**	**assoc.**	**block**	**size**	**assoc.**
Baseline	32K	32	32	32K	32	32	512	1
Best cycles	128K	32	64	128K	4	64	128	1
Best energy	16K	64	16	8K	64	16	128	1
Best ED	32K	64	16	16K	32	16	128	4
Best EDD	32K	64	16	64K	32	32	256	4

(a) Microarchitectural parameters

Configuration	**Cycles**	**Energy**	**ED**	**EDD**
Baseline	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)	1.00 (1.00)
Best cycles	**0.79** (0.97)	1.86 (2.09)	1.46 (2.02)	1.14 (1.96)
Best energy	1.01 (1.14)	**0.71** (0.81)	0.72 (0.93)	0.72 (1.06)
Best ED	0.87 (1.14)	0.77 (0.81)	**0.67** (0.93)	0.59 (1.06)
Best EDD	0.81 (0.90)	0.84 (0.95)	0.68 (0.94)	**0.55** (0.93)

(b) Best values achieved

Table 5.5: The microarchitectural parameters of the baseline and the best configurations found from the sampled space when considering the co-design (a). Their corresponding values for cycles, energy, ED and EDD are also shown (b). The values in parenthesis show the corresponding metric for the configurations leading to the best microarchitecture when the compiler optimisations are not considered (copied from table 5.3(b)).

the best configuration are actually different from the ones where the microarchitecture design space was considered in isolation (table 5.3(a)). This illustrates the point raised in the previous paragraph where the architect can potentially make a wrong decision because he does not have access to the real design space.

5.5.3 Details Per Program

In the embedded world, the design process is ultimately driven by the applications. This is in fact key for efficiency since the design can be targeted and tuned specifically for a fixed set of programs. It is therefore important to have a detailed view of the performance for each individual program.

Figure 5.7 shows the execution time, energy, ED and EDD values on a per-benchmark basis for the microarchitectural/optimisation configurations that perform the best for each metric (table 5.5(a)). In terms of execution time, 13 benchmarks achieve at least 20% improvement, the best achieving up to 57% of improvement for *search*. The average improvement is 20% for

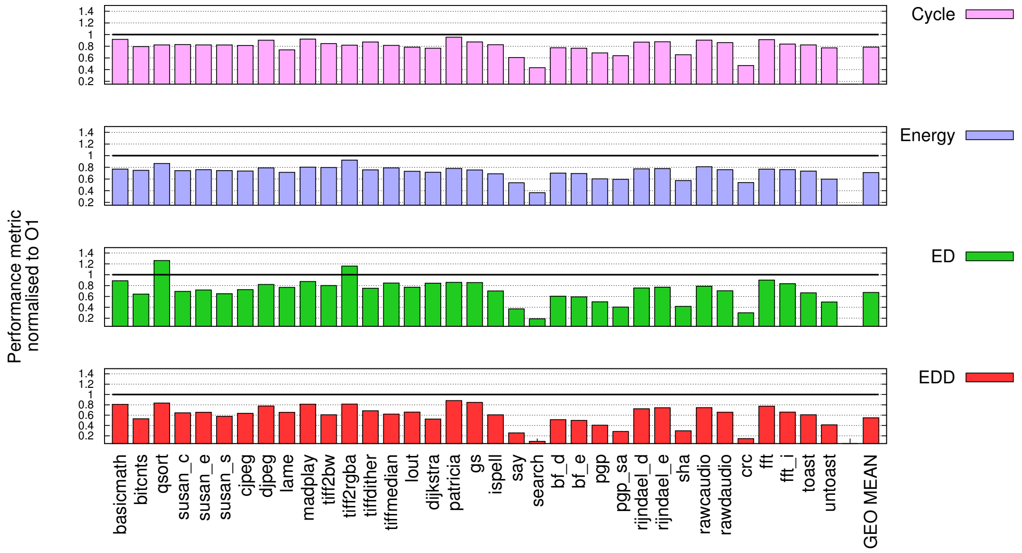


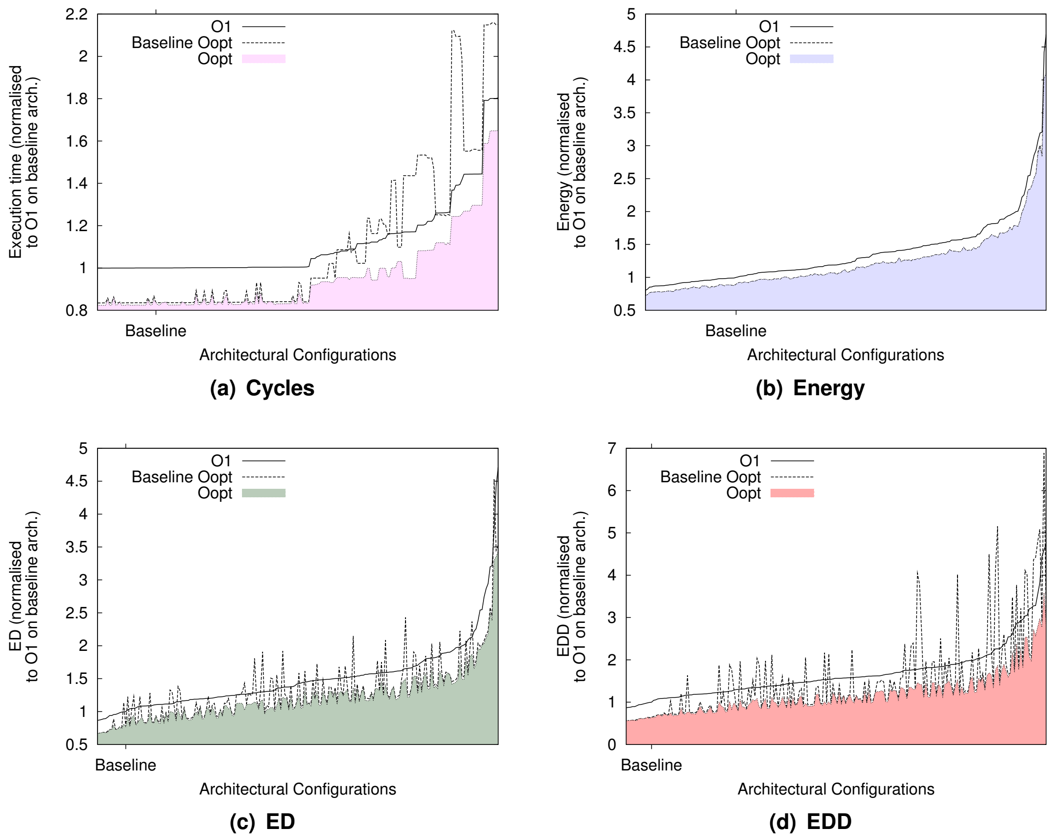
Figure 5.7: Execution time, energy, ED and EDD for each benchmark on the microarchitectural/optimisation configuration performing the best for each metric. The microarchitecture is the same across programs however it varies depending on the target metric.

the whole of MiBench. For energy the majority of the benchmarks achieve a saving of more than 20% with an average of 29% energy saved.

For ED, the majority of benchmarks achieve a reasonable value of 0.8 or under with an average value of 0.67. It is interesting to compare these results with those achieved when performing microarchitecture space exploration alone (figure 5.3). Performing co-design space exploration leads to more balanced results across benchmarks for ED (the maximum value is now 1.3, before it was 1.7). This is possible thanks to the optimising compiler that is able to take full advantage of the microarchitecture, whereas previously all benchmarks were compiled with **-O1**. For EDD, all the benchmarks show an important improvement over the baseline leading to an average value of 0.55, with none being worst than the baseline.

5.5.4 Optimisation Sensitivity to Microarchitecture

The previous sections have shown that co-design space exploration is beneficial over performing microarchitecture and optimisation space exploration in isolation. This section now looks at what happens if one uses the best flag settings found for one program on the baseline architecture for the rest of the microarchitectural space for this same program. In other words this section examines whether a fixed set of optimisation flags exists for a particular program that can lead to the best performance independently of the microarchitecture.



(a) **Cycles**

(b) **Energy**

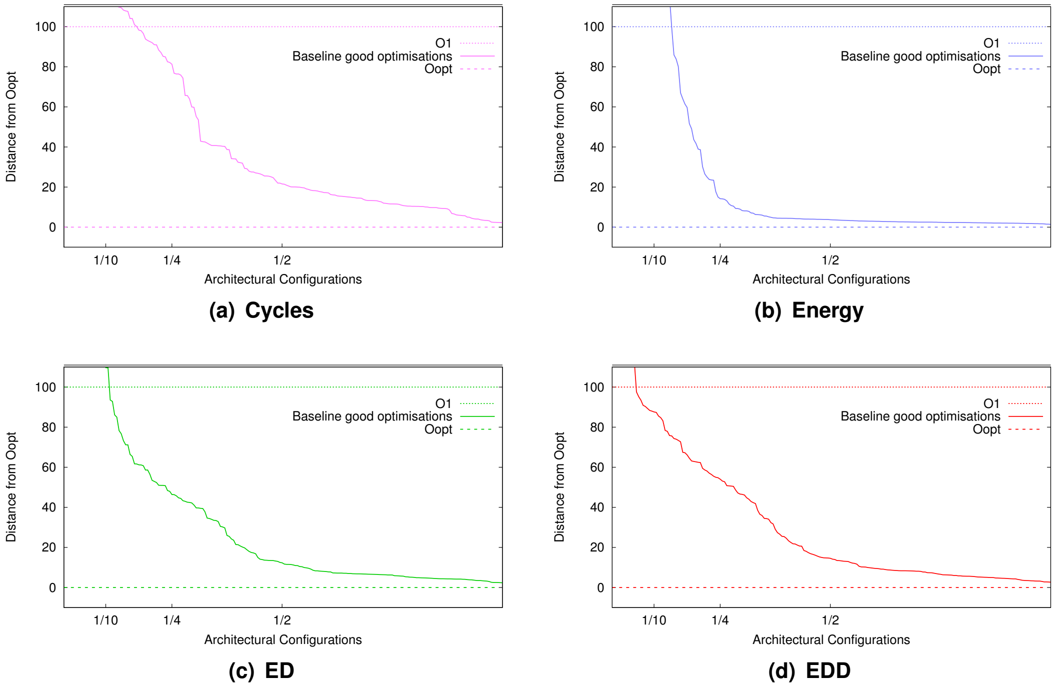
(c) **ED**

(d) **EDD**

Figure 5.8: Optimising program *toast* on the baseline architecture and running it on all other microarchitectural configurations. All the values are normalised by **-O1** on the baseline microarchitecture.

Figure 5.8 shows an example for the *toast* benchmark when optimising the program on the baseline architecture and running it on the other configurations. As can be seen, the optimisations that is the best for the baseline microarchitecture actually performs worse than compiling with **-O1** on other configurations for cycles, ED and EDD. Critically, the best compiler optimisations vary across the microarchitecture space. However, for energy it seems to make only little difference. As seen earlier in this chapter, this is due to the fact that the compiler optimisations have very little impact on the energy consumption.

Figure 5.9 shows this averaged across all benchmarks. Here all programs were run using 1000 optimisations on the baseline architecture and those optimisations that are within 5% of the best found for each benchmark were selected (a different set for each program). They are called the *baseline good* optimisations. Then the benchmarks compiled with these baseline good optimisations were run on the rest of the microarchitectures to determine the average cycle, energy, ED and EDD values that they achieve. For each configuration, the performance of



(a) Cycles

(b) Energy

(c) ED

(d) EDD

Figure 5.9: Optimising on the baseline architecture and running on all other microarchitectural configurations. Optimisations that are good on the baseline microarchitecture can perform worse than **-O1** on other configurations. All the results are averaged across all benchmarks.

the *baseline good* optimisations were evaluated using the distance from the best value achievable on that configuration. The distances were normalised by the performance of the best optimisation (distance=0%) and the performance of **-O1** (distance=100%).

For ED and EDD it can be seen that on half (1/2) the architectures the good optimisations for the baseline are at least 15% away from the best. For a quarter (1/4) of the architectures, these good optimisations are at least around 50% away from the best. Crucially, the good optimisations on the baseline architecture are actually worse than **-O1** for one tenth (1/10) of the microarchitectures. This shows that good optimisations for one architecture are not necessary suitable for others. In essence, the optimal compiler optimisations to apply for one architecture are not the best for all. Therefore the compiler has to be tuned on each configuration and cannot be developed independently of the microarchitecture.

5.5.5 Summary

This section has shown the importance of performing co-design space exploration. When the optimisation space is explored at the same time as the microarchitecture space, significant improvements can be gained over the baseline. However, designing the architecture without considering the optimisation space can result in sub-optimal performance on the final system. This is due to the fact that the real design space, that integrates compiler optimisations exploration, is hidden from the designer.

However, one of the problems with performing the exploration of both design spaces is the large amount of simulations needed. It is, therefore, not practical to systematically explore the optimisation space of every single design point. In order to address this issue, the next section presents a model that predicts, with high accuracy, the best performance achievable by an optimising compiler for any microarchitecture on a given program.

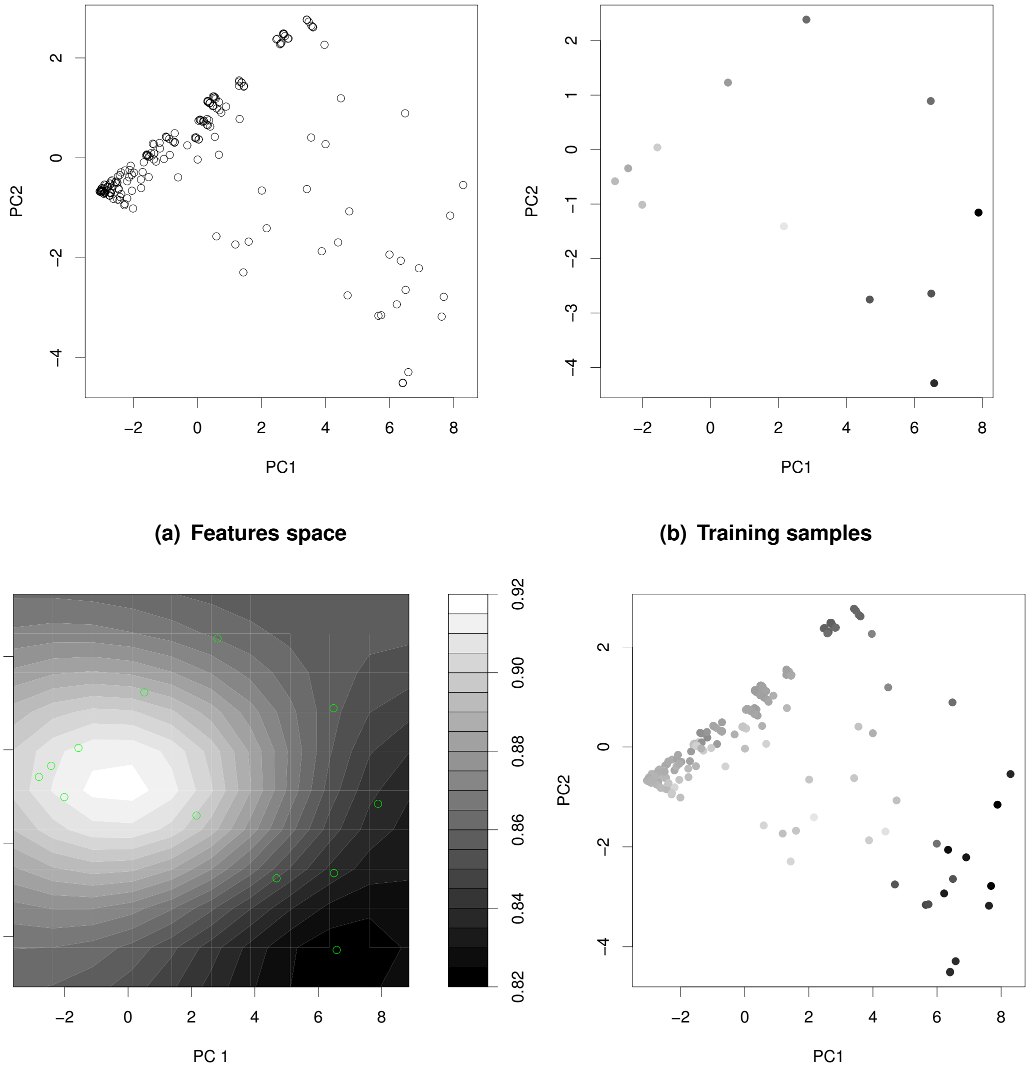
5.6 Predicting the Performance of an Optimising Compiler

Previous sections presented the characteristics of the design spaces. It was shown that the optimal compiler settings for one architecture are not necessarily the best for all and that they are also program-dependent. For this purpose, a sample of the total design space was explored, consisting of 200 microarchitectural configurations and 1000 compiler optimisations over 35 benchmarks. In practise, however, it is not desirable to conduct such a costly co-design space exploration.

To address this issue, this section develops a machine-learning model which predicts the performance of the optimal compiler settings on any microarchitectural configuration for a given program. Based on this prediction, the designer can know what the performance is of the most optimal flag settings for any architecture, without having to search the optimisation space for every single microarchitecture. This information can then be used to conduct an efficient exploration of the architectural design space, taking into account the effects of compiler optimisations.

5.6.1 Overview

The model is built in three steps, as shown on the example of the benchmark *fft* in figure 5.10. A new model is created for each individual benchmark the designer wants to predict for. First the program, compiled with **-O1**, is run on a number of randomly-selected microarchitectures forming the sample design space (200 in this case). This is what is typically done when exploring the microarchitectural design space of a processor. Then for each of these runs, the values of the performance counters are extracted to allow the characterisation of the behaviour of each architecture (figure 5.10(a)).



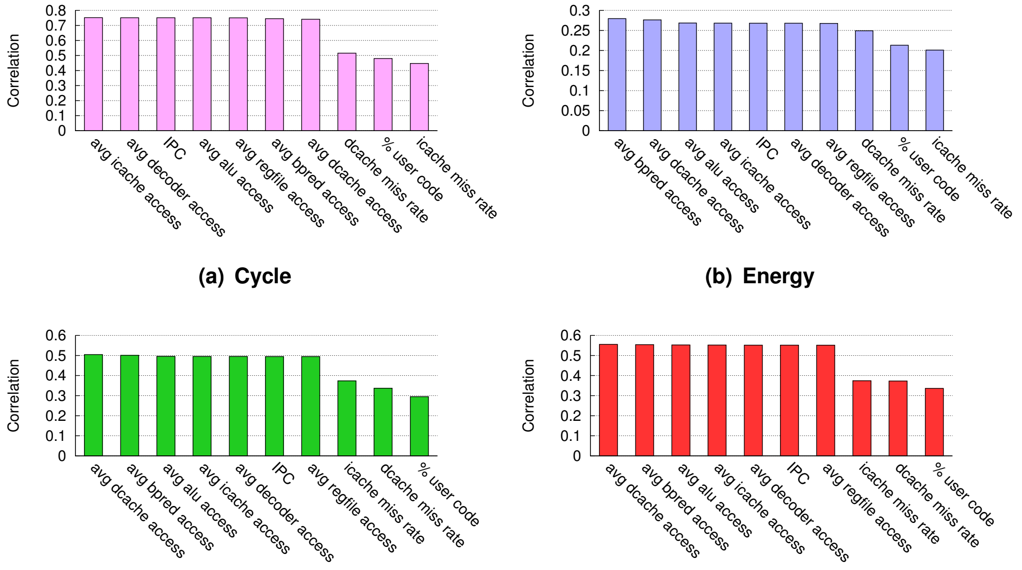
(a) **Features space**

(b) **Training samples**

(c) **Trained model (SVM)**

(d) **Real space**

Figure 5.10: Example use of the model for the program *fft* when considering ED (the darker a dot, the better the ED value is over the baseline **-O1**). First performance counters are collected, then PCA is used to select two components (a). Some training configurations are then selected and a search of their optimisation spaces for the best performance is conducted (b). Finally, a SVM model is trained to determine the contour map around configurations (c). This map provides a prediction of the real performance of the optimising compiler (d).



(a) Cycle

(b) Energy

(c) ED

(d) EDD

Figure 5.11: The correlation (absolute value) between the selected performance counters and the performance of the optimising compiler (normalised by the performance of the baseline optimisation (**-O1**)). For each target metric, the correlation is averaged across all programs.

From these runs, a small number of architectural configurations are selected for training (figure 5.10(b)). The optimisation space is then explored only for these selected microarchitectures in order to find the best performance achievable. Different compiler settings are tested (using 1000 random flag settings) and only the performance of the best is kept.

Finally, the model is trained with the results of this exploration using Support Vector Machines for regression SVM (figure 5.10(c)). The predictions resulting from this training can be compared with the real space (figure 5.10(d)).

The next sections describe these three steps in more detail by first showing how the performance counters correlate with the performance of the best flag settings for each architecture. Then the selection procedure used to gather the training data is explained and finally the machine-learning model is described.

5.6.2 Characterisation of Microarchitectures with Performance Counters

As seen in the overview section, the model first characterises each microarchitecture and then performs an exploration of the optimisation space for a few selected ones. The rationale being that the results obtained from these explorations can be reused for other microarchitectures. This is done by identifying similar microarchitectures using features. These microarchitectures that have similar features are expected to have similar performance improvement over the default optimisation level **-O1**.

To characterise each microarchitecture in the co-design space, the model uses performance counters extracted from a single run of the program with the default optimisation level (**-O1**) on each architecture. Nine performance counters were chosen to characterise the behaviour of each microarchitecture. They are the IPC; resource utilisation of the caches, branch predictor, ALUs and register file; cache miss rates and the branch miss-prediction rate. These performance counters are typically found in microprocessors' analytic models [Eyer 06b, Kark 04]. The correlation between each of these counters and the performance of the optimal flag settings for each microarchitecture is shown in figure 5.11 for each of the target metrics considered. Interestingly for each of these metrics, the highest correlation is obtained for the counters associated with resources utilisation and the IPC. Then information about cache miss rate comes as the second most important kind of features followed by the percentage of user code (code that the compiler can transform). The other performance counters available were found to lead to a much lower correlation. Therefore they were ignored and only this set of nine was retained.

In order to reduce the number of inputs to the model, PCA was used. This technique was used to summarise these nine features into a couple of values or principal components. In this case the number of selected principal components was fixed to two. This offers the advantages of easy visualisation and adding more components did not improve the overall accuracy. Figure 5.10(d) shows the projections of the nine performance counters onto the two main components (PC1 & PC2) for the real space over 200 microarchitectures for the benchmark *fft*. As can be seen the features are able to separate the microarchitectures in the space depending on the improvement available when exploring the optimisation space compared to the baseline compiler optimisation. For instance the microarchitectures that offer significant improvement when exploring the optimisation space tend to be situated in the lower right corner of the space (dark points), with ED values around 0.82.

5.6.3 Gathering Training Data

Before being able to build a model, a few microarchitectures need to be selected to train the model. For each of these, an exploration of the optimisation space is required to obtain the performance of the best optimisation settings. For this reason it is important to select carefully the training configurations so as to avoid unnecessary explorations.

To achieve high efficiency, the idea consists of selecting training samples that best cover the projected space. To do so, K-Means was used to find clusters of microarchitectures based on the performance counters. Then one representative microarchitecture was selected for each cluster followed by an exploration of the optimisation space for that particular microarchitecture. Figure 5.10(b) shows the configurations being selected for training for the benchmark *fft* with a number of clusters set to 12. The optimal number of clusters selected is discussed later in section 5.7.1 where the technique is evaluated. As can be seen the selected microarchitectures successfully cover the space.

Once this selection has taken place, a search of the compiler optimisation space on each of the selected microarchitectures is performed using iterative compilation with 1000 randomly-selected optimisations. Note that this search could be made more efficient by using more advanced search strategies [Alma 04, Coop 05, Hane 05, Tria 03]. However, this is orthogonal to the focus of this work. The result of this search corresponds to an estimation of the maximum performance achievable on each of the selected training microarchitectures. This is shown in figure 5.10(b) where darker points lead to better performance. With this training data gathered, the model is ready to be trained.

5.6.4 Training the SVM Model

Having collected the training data, the model is now ready to be trained. The model chosen is based on Support Vector Machines (SVM), adapted for the regression problem [Smol 03]. This model is able to distinguish between data points that behave differently, *i.e.* microarchitectures with different potential performance improvement when exploring the optimisation space over the baseline compiler.

Following the example of the benchmark *fft*, the results from training can be seen visually in figure 5.10(c). Here the selected training configurations have been circled. Intuitively the model learns the areas of similar performance (similar colours) based on the best performance seen on the selected microarchitectures from the previous step. Architectural configurations that lie in the same coloured area are predicted to have similar performance improvement available. In other words, the model predicts that the optimising compiler has little effect in the light areas and can achieve high performance gains in the dark areas.

For *fft*, this can be compared to the real space of 200 microarchitectures in figure 5.10(d). As can be seen in this figure, the real space is correctly predicted. The points in the centre have a lower potential for improvement (light area) whereas the ones in the lower right corner tend to have larger improvement available (dark area).

Having trained the model, new predictions can now be made for any microarchitecture. All that is needed is a single run of the application with **-O1** on the new microarchitecture. This run is used to gather the performance counters characterising the new microarchitecture. The model then predicts the performance achievable if one were to conduct an exploration of the optimisation space. This is achieved by first projecting the nine performance counters using PCA and then by making a prediction based on the corresponding value on the contour map.

5.6.5 Summary

This section has described a model that predicts the performance of the best compiler flag settings on any microarchitectural configuration. First, a run of the program compiled with **-O1** on 200 architectures is performed and nine performance counters are gathered. PCA is used to reduce these to just two values and then the training samples are selected to train the model. For each of these training configurations a random search of the optimisation space is performed to estimate the best performance achievable by a yet-to-be-built optimising compiler. Then an SVM is trained to model the entire co-design space. Once the model is trained, it can be used to predict the performance of the best compiler flag settings for a given program on any new microarchitecture. To do so, a single run of the program compiled with **-O1** is required on the configuration of interest. After extraction of the performance counters, the model is then able to make an accurate prediction.

In the next section, a complete evaluation of the model is performed and its accuracy assessed. It will show that the model achieves high accuracy using only a tiny fraction of the design space for training, making it suitable for efficient co-design space exploration.

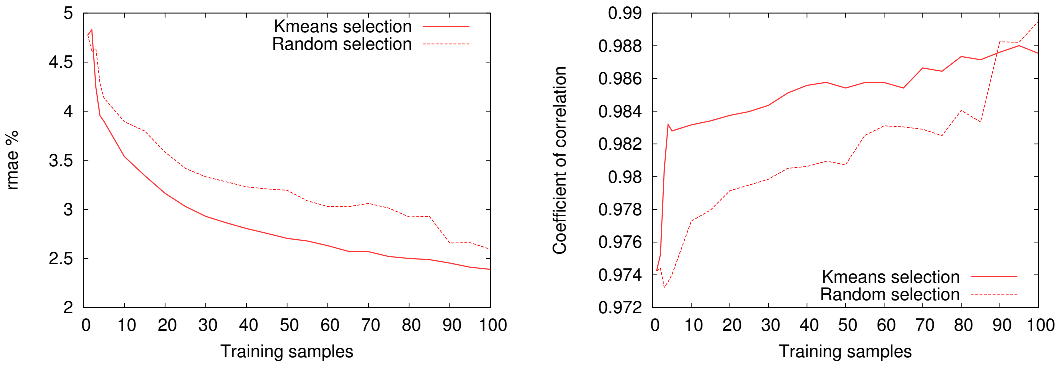
5.7 Model Evaluation and Comparison

This section evaluates the prediction accuracy of the machine-learning model developed in the previous section. It first looks at the accuracy of the model when the number of training samples is varied. The accuracy of the model is then evaluated on a per program basis for the whole of the MiBench benchmark suite. Finally, a comparison with the state-of-the-art is performed and a search of the design space is conducted using the model.

5.7.1 Training Samples Selection: K-Means vs Random

As seen in the previous section which described the model, a few microarchitectures are selected in order to train the predictor. This selection process is performed by using the K-Means clustering technique to pick a representative microarchitecture for each cluster found. This procedure is now evaluated for different training sizes and compared with a purely random selection process.

Figure 5.12 shows the mean error and the coefficient of correlation of the model for various training sizes. These results are averaged across all the benchmarks and obtained using cross-



(a) Error **(b) Correlation**

Figure 5.12: Mean error and coefficient of correlation for both the K-Means and a random selection process averaged across all programs for various training samples size.

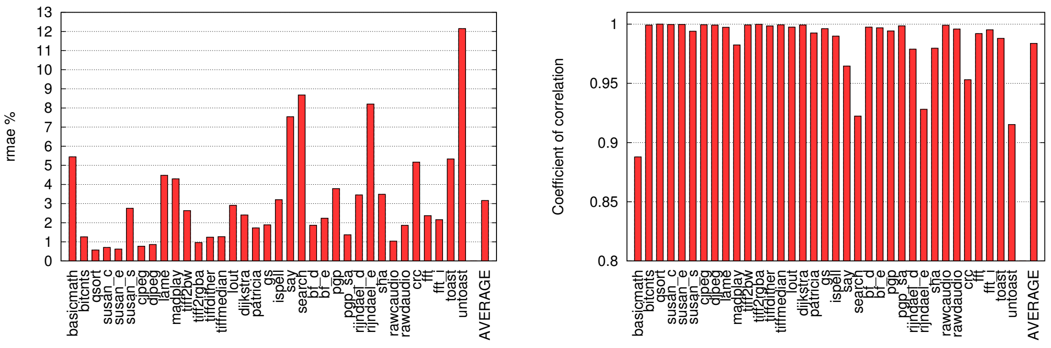
validation where the microarchitectures used for training are left out of the testing set. This figure clearly shows that selecting the training points with the K-Means algorithm is better than using a purely random selection. For instance with 20 training samples, the K-Means approach achieves an error rate of just 3.2% and a correlation of 0.984 whereas the random selection achieves only 3.7% and 0.979 respectively. This also shows that a low error can be achieved using only a fraction of the design space.

It might at first seem surprising to see that the coefficient of correlation is high even when using very few training samples. Looking back at figure 5.6 it can be seen that the performance of the best flag settings is strongly correlated with the performance of the baseline optimisation **-O1**. This is due to the fact that the microarchitectural space has a much higher variance than the compiler optimisation space. Furthermore, it is important to keep in mind that these numbers are averaged across all programs. Hence, programs that show very little variation in their optimisation space will tend to be easier to predict, independently of the number of training points.

For the following sections, the training budget was fixed arbitrarily to 20 training samples since it is a good tradeoff between accuracy and the number of training samples. As the next section shows, this leads to a very good correlation and relatively low error for most of the programs.

5.7.2 Prediction Accuracy Per Program

The prediction error and coefficient of correlation for each benchmark of MiBench is shown in figure 5.13 for 20 training samples. As stated earlier, these 20 samples correspond to 20 mi-



(a) Error **(b) Correlation**

Figure 5.13: The prediction errors of the model broken down by program when predicting the EDD value achievable by the optimising compiler using only 20 training microarchitectures. The average error is just 3.2% and the average correlation 0.98.

croarchitectures on which the best performance achievable was estimated using 1000 random optimisations. The average error and correlation of the whole suite is also shown.

As can be seen, the model achieves a very low error rate of 5% or under for the majority of the benchmarks. In fact, for some benchmarks (such as *susan_e*), the error is as low as 0.6%. The coefficient of correlation is also very good for all the benchmarks, the lowest achieving a correlation of 0.88.

This shows that the model is accurate and can correctly predict the performance of the optimising compiler. In the next section a comparison will be conducted with a state-of-the-art technique for co-design space exploration.

5.7.3 Comparison with State-of-the-Art

This section now compares the accuracy of the SVM model with the only other technique that considered the joint exploration of the microarchitecture and compiler space. This other technique proposed by Vaswani *et al.* [Vasw 07] makes use of an Artificial Neural Network (ANN) to predict any point in the co-design space. However, their model does not directly predict the performance of an optimising compiler but instead predicts the performance of a set of compiler flags for a given microarchitecture described by its configuration. It uses as an input the microarchitectural parameters and the compiler flag settings to make a prediction.

Since the *Vaswani* approach is slightly different than the scheme developed in this chapter, it was used in the following way: when a new microarchitecture is encountered, their model is used to search the optimisation space and the flag settings corresponding to the best prediction

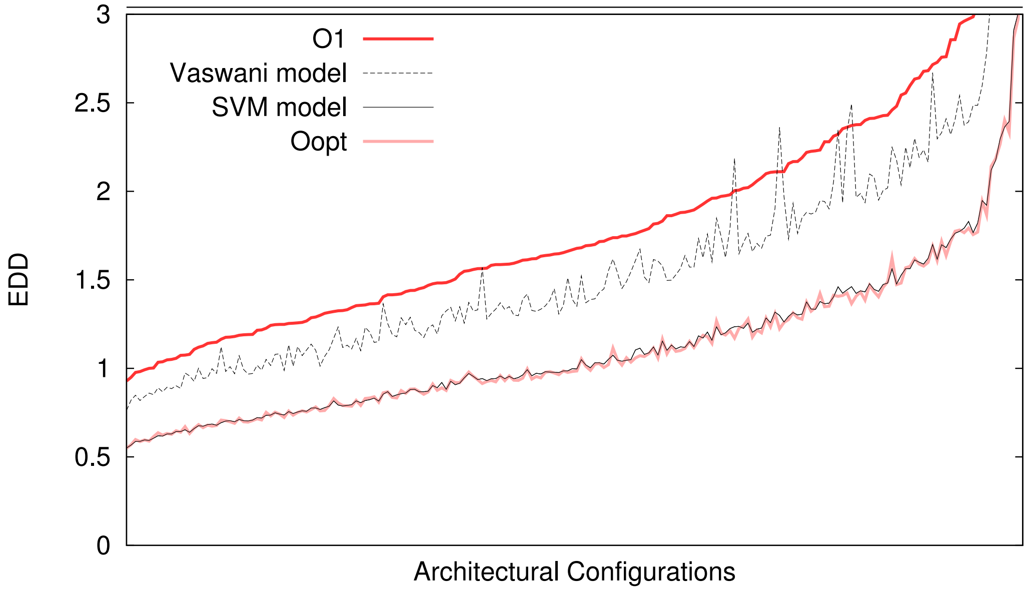


Figure 5.14: Predicting the EDD metric achieved by the optimising compiler across the microarchitectural space for the whole of MiBench. Also shown are the predictions made by *Vaswani*. Note that the SVM model developed in this chapter is highly accurate and overlaps significantly with **Oopt**.

are recorded. Then a run with the application compiled with these settings is performed to find the real performance measure. To allow a fair comparison, their model was trained with exactly the same data as for the SVM model; 20 architectures and 1000 flag settings each.

Figure 5.14 shows the EDD values achieved by the baseline compiler on each microarchitectural configuration averaged over the whole of MiBench benchmark suite (labelled O1). It also shows the EDD values achieved by the optimising compiler on each configuration (Oopt). A third line shows the predictions made by the model proposed by Vaswani *et al.* (*Vaswani* model) and a final line shows the predictions of the SVM model.

As can be seen, the SVM predictions follow the curve of the optimising compiler with great accuracy. More specifically, the SVM model accurately predicts the peaks and troughs in EDD as well as the stable areas. This shows the ability of this model to predict the design points that behave significantly differently from the baseline. The *Vaswani* model, however, fails to accurately predict the performance of the optimising compiler. In particular, it predicts peaks in EDD where there are none and follows the **-O1** line closely. Furthermore it even predicts values larger that **-O1** in some cases. This predictor, therefore, cannot be used to find the performance of an optimising compiler.

The *Vaswani* model is not adequate for the task of exploring the co-design space. Indeed

Configuration	**Icache**			**DCache**			**BTB**	
	size	**assoc.**	**block**	**size**	**assoc.**	**block**	**size**	**assoc.**
Best EDD	32K	64	16	64K	32	32	256	4

(a) Microarchitectural parameters

Configuration	**Predicted EDD**		**Real EDD**
	SVM model	***Vaswani* model**	
Best EDD	0.550	0.762	**0.549**

(b) Best value predicted and achieved

Table 5.6: Parameters and EDD value of the best configuration found using the SVM model.

this model tries to generalise the performance of the optimisation flags across all the architecture space. However, due to the lower variation available in the compiler space in comparison to that of the microarchitectural space, the model fails to correctly learn since the microarchitectural parameters dominate. Therefore this model cannot distinguish between different flag settings. In this experimental setup it was noticed that the *Vaswani* model often makes the exact same prediction for a given architecture, independently of the input flags. In a nutshell it is hard to build a unique model that both models the architecture space and the compiler space for all the possible flag settings. This contrasts with the approach that uses the SVM model where the model only uses information about the microarchitecture to infer how much improvement is possible, were a search of the optimisation conducted.

5.7.4 Predicting the Best Architectural/Optimising Compiler Configuration

Having built and evaluated the SVM model, this section considers its use for searching the co-design space. This allows designers to determine the optimising compiler/architectural configuration that achieves the best EDD value in the space. To this end, the SVM model was used to predict the EDD value that an optimising compiler would achieve on 200 microarchitectures, chosen by uniform random sampling. The best predicted value was 0.550 for the configuration shown in table 5.6.

To verify the prediction accuracy, iterative compilation was used on this architecture with 1000 randomly-selected optimisation settings. It was found that the best EDD value achievable is 0.549. This is just 0.2% away from the prediction, showing that this SVM model is very accurate. Had the *Vaswani* model been used, it would have predicted an EDD value of 0.762 for this same configuration, which is an error of 39%.

It is interesting to notice that this configuration corresponds to the best one found when performing an exhaustive search of the sample space as seen earlier in table 5.5(a). This proves that this SVM model can be used in practice. It can successfully find the best configuration without performing the full co-design space exploration for every microarchitecture. Instead, it simply explores the co-design of 20 configurations for training purposes. This represents an order of magnitude fewer simulations than what would be required in order to explore this sample space exhaustively.

5.7.5 Summary

This section has evaluated a SVM model and shown that it is highly accurate. It was compared with the only other technique that predicts the joint compiler/architecture space. As seen, the SVM model is more accurate at predicting the performance of the optimising compiler on any microarchitectural configuration. Furthermore, it was shown that this model can predict the best optimising compiler/architectural configuration for EDD within the sample space with just a 0.2% error. Using this model, designers can accurately and efficiently predict the impact of the optimising compiler across the entire co-design space.

5.8 Conclusion

This chapter shows how compiler optimisations can influence the design of embedded processors. Using a typical embedded processor, the XScale, an exploration of its microarchitecture and compiler optimisation spaces has been conducted in isolation. It shows that significant improvements exist in the compiler space and that it cannot be ignored at design time. For this reason the co-design space was then explored and presented using a sample space composed of 200 microarchitectures and 1000 optimisation settings. Failing to take into account compiler optimisations at the design stage can mislead the architect.

Because it is not practical to conduct such a co-design space exploration, a machine-learning model has been proposed. This model, trained on a fraction of the co-design space, makes accurate predictions of the best performance achievable for any microarchitecture, had the compiler been tuned for it. This clearly gives an advantage to the designer who can use this model to make better design decisions. The model has been demonstrated to work accurately. Furthermore, it was compared against an existing approach which failed to provide insightful information to the designer.

Chapter 6

Towards a Portable Optimising Compiler

6.1 Introduction

The previous chapter presents a model to predict the performance of the architecture/optimising compiler co-design space. This chapter builds a new machine-learning compiler that can achieve a significant portion of the best performance available in the compiler space, for any microarchitecture. This compiler is named TALC: the Trans-Architecture Learning Compiler. Given a new microarchitecture, it automatically determines the right optimisation settings to apply for any new program with just one profile run. This approach is based on machine-learning, where a model is learnt off-line "at the factory". This model maps an architecture description plus the hardware counters from a single run of a program to the compiler flag settings leading to the best execution time.

The learning process is a one-off activity whose cost is amortised across all future uses of the compiler on any variation of the processor's base architecture. This approach achieves a 1.14x speedup on average over the highest default compiler optimisation, validated across 200 microarchitectural configurations. This represents 61% of the performance improvement gained by standard iterative compilation using 1000 evaluations.

Given this new approach, a new compiler does not need to be developed when the processor microarchitecture changes. This allows compilers to become fully integrated in the design space exploration of a new processor, helping designers to fully evaluate the potential of any new architecture. In addition, this enables the design of a parametrised embedded processor that can be shipped with this portable compiler. Customers that acquire such design do not need to retune the compiler for their specific implementation since the compiler is generic and

Parameter	**Low → High**	**Baseline**
ICache size	4K → 128K	32K
ICache associativity	4 → 64	32
ICache block size	8 → 64	32
DCache size	4K → 128K	32K
DCache associativity	4 → 64	32
DCache block size	8 → 64	32
BTB size	128 entries → 2048 entries	512 entries
BTB associativity	1 → 8	1

Table 6.1: Microarchitectural parameters and the range of values they can take. Each parameter varies as a power of two, with 288,000 total configurations. Also shown are the baseline values.

will know, given the specific microarchitectural parameters chosen by the customer, how to compile optimally for it.

This chapter is organised as follow. The experimental setup used within this chapter is almost identical to that described in the previous chapter and is briefly discussed in the next section. A characterisation of the co-design space is shown in section 6.3, which focuses on compiler optimisations. Section 6.4 describes the design of TALC, while section 6.5 evaluates the model parameter and the features used as an input. An experimental validation of the model is then performed in section 6.6 where the actual predictions made by the model are checked by simulation. Section 6.7 analyses in more detail the results obtained and gives insights into why the model is actually working. Finally section 6.8 concludes this chapter.

6.2 Experimental Setup

The experimental setup considered in this chapter is absolutely identical to that described in the previous chapter. For the sake of completeness, the different tables describing the experimental setup are reproduced here.

The Xtrem simulator [Cont 04] which was used to run the benchmarks, models the Intel XScale processor. Table 6.1 shows the 14 different microarchitectural parameters varied, leading to a total design space of 288,000 different configurations. A sample space of 200 configurations was selected with uniform random sampling to conduct the experimentation.

All the 35 programs from MiBench [Guth 01], shown in table 6.2 were used and consistently run until completion in all experiments. They were compiled with *gcc* using a 1000 different flag settings randomly selected from a total space of 1.69^{17} points. This space was obtained by varying the values of 39 different flags shown in table 6.3 from the command line.

Program	Input	Sim. instr.	% lib code
basicmath	small	9m	**77**
bitcnts	small	45m	0
qsort	small	32m	**97**
susan_c	large	22m	3
susan_e	large	52m	0
susan_s	small	18m	4
cjpeg	small	31m	2
djpeg	large	24m	2
lame	small	116m	14
madplay	small	25m	0
tiff2bw	small	34m	18
tiff2rgba	large	31m	36
tiffdither	small	358m	1
tiffmedian	small	148m	8
lout	small	56m	14
dijkstra	small	43m	4
patricia	small	9m	**72**
ghostscript	small	75m	5
ispell	small	8m	27
say	small	59m	8
search	large	1m	14
bf_d	small	26m	0
bf_e	small	26m	0
pgp	*NA*	1m	8
pgp_sa	*NA*	94m	0
rijndael_d	small	23m	0
rijndael_e	small	23m	0
sha	small	14m	3
rawcaudio	small	37m	0
rawdaudio	small	27m	0
crc	small	18m	0
fft	small	5m	**60**
fft_i	small	11m	**58**
toast	small	29m	0
untoast	small	16m	0

Table 6.2: The 35 MiBench benchmarks used with their corresponding input size, the total number of instructions executed and the percentage of library code executed.

In contrast with the previous chapter that looked at the co-design space for four target metrics, namely cycles, energy, ED and EDD, this present chapter only considers one target metric: cycles. This is because the effects of the optimisations influence mainly the cycle metric. As a result, the choice of the baseline optimisation level has been reconsidered. Since **-O3** leads to the best average execution time across all programs (figure 5.1), it was chosen as the baseline optimisation level throughout this chapter.

6.3 Characterising the Compiler Space

Before building a compiler that can optimise across architectures, it is important to examine whether there is any performance to be gained within the compiler optimisation space. In comparison with the previous chapter, this section focuses more on showing how difficult the problem of finding the best optimisations is. In particular it shows the empirical distribution of the speedups corresponding to the different optimisation settings applied. Additionally, the performance of iterative compilation is examined.

N°	Flag
1	-fthread-jumps / ⊘
2	-fcrossjumping / ⊘
3	-foptimize-sibling-calls / ⊘
4	-fcse-follow-jumps / ⊘
5	-fcse-skip-blocks / ⊘
6	-fexpensive-optimizations / ⊘
7	-fstrength-reduce / ⊘
8	-frerun-cse-after-loop / ⊘
9	-frerun-loop-opt / ⊘
10	-fcaller-saves / ⊘
11	-fpeephole2 / ⊘
12	-fregmove / ⊘
13	-freorder-blocks / ⊘
14	-falign-functions / ⊘
15	-falign-jumps / ⊘
16	-falign-loops / ⊘
17	-falign-labels / ⊘
18	-ftree-vrp / ⊘
19	-ftree-pre / ⊘
20	-funswitch-loops / ⊘

N°	Flag	Values
21	-fgcse / ⊘	
22	-fno-gcse-lm / ⊘	
23	-fgcse-sm / ⊘	
24	-fgcse-las / ⊘	
25	-fgcse-after-reload / ⊘	
26	–param max-gcse-passe =	1, 2, 3, 4
27	-fschedule-insns / -fschedule-insns2 / ⊘	
28	-fno-sched-interblock / ⊘	
29	-fno-sched-spec / ⊘	
30	-finline-functions / ⊘	
31	–param max-inline-insns-auto =	10,30,50,...,190
32	–param large-function-insns =	1300,1500,1700,...,3300
33	–param large-function-growth =	20,50,100,200,300,400,500
34	–param large-unit-insns =	4000,6000,8000,...,20000
35	–param inline-unit-growth =	10,20,30,...,100,200,300
36	–param inline-call-cost =	10,12,14,...,30
37	-funroll-loops / -funroll-all-loops / ⊘	
38	–param max-unroll-times =	2,4,6,...,20
39	–param max-unrolled-insns =	50,75,100,...,400

Table 6.3: Compiler optimisations and the values they can take. There are 642 million combinations.

For this purpose, the impact of the compiler optimisations was evaluated on the 35 MiBench programs compiled with the 1000 random flag settings, each of them being executed on the 200 different architectural configurations. This corresponds to a sample space of approximately seven million simulations and should provide some evidence of the potential benefits of tuning optimisation flags across microarchitectures and programs.

6.3.1 Sample Space Distribution

This section demonstrates that flag selection has a significant effect on program performance. Figure 6.1 shows the sample space's distribution of speedups or slowdowns for each program across the microarchitectural configurations. In other words, this distribution reflects what happens if one chooses a random architectural configuration and a random flag setting. The x-axis represents the program and the y-axis the speedup relative to **-O3**. Each "box and whisker" entry summarises the distribution of speedups for that program. The central line denotes the

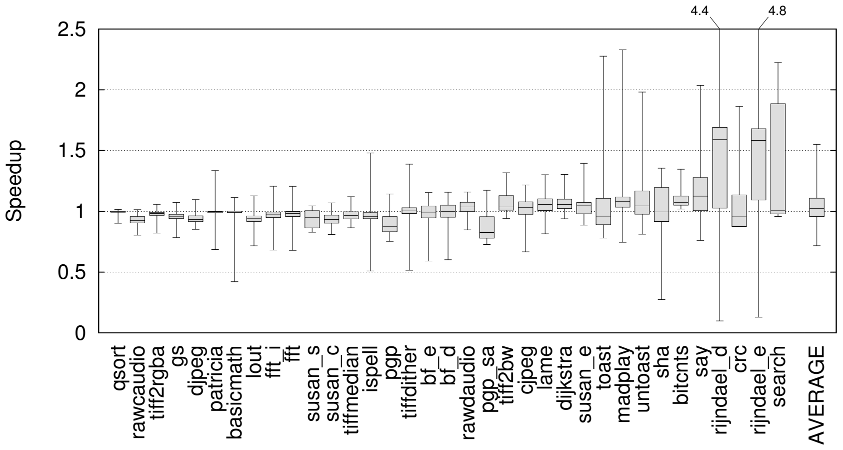


Figure 6.1: Distributions of speedups for the 1000 optimisation settings within the sample space across all architectures on a per-program basis. The x-axis represents the program and the y-axis the speedup relative to compilation with **-O3**. The central line denotes the median speedup. The box represents the 25 and 75 percentile area whilst the outer whiskers denote the extreme points of the distribution.

median speedup. The box represents the 25 and 75 percentile area whilst the outer whiskers denote the extreme points of the distribution.

It is apparent that, for some benchmarks, the optimisation flags can have a significant effect across the microarchitectural space. The most extreme example is *rijndael_e*. Here, a random optimisation setting leads to a speedup of 1.6x on average (the central line within the box) across the architecture space, showing how **-O3** performs relatively poorly. For a particular architecture, it is possible to find a flag setting that improves performance by a factor of 4.8x. Similarly, it is also possible to find an architecture/flag pair leading to a speedup of 0.12x, or put another way, the program executes eight times slower.

In the case of *search*, nearly all compiler settings will do better than **-O3** on any architecture. Programs dominated by library calls (see table 6.2) such as *qsort* are almost immune to compiler optimisations as these do not affect pre-compiled library code. The same is largely true for *basicmath*, except that the number of instruction cache misses is occasionally increased through excessive loop unrolling, which halves performance for certain architectures with small cache size. Hence, for these cases it is possible to find optimisation settings that prevent too much unrolling and allow better performance than **-O3**.

Overall, a random optimisation on a random architecture will give just a 1.02x speedup over **-O3**. Although no-one would suggest optimising a program by randomly applying optimisations, it is important to show that a purely random technique would not perform much

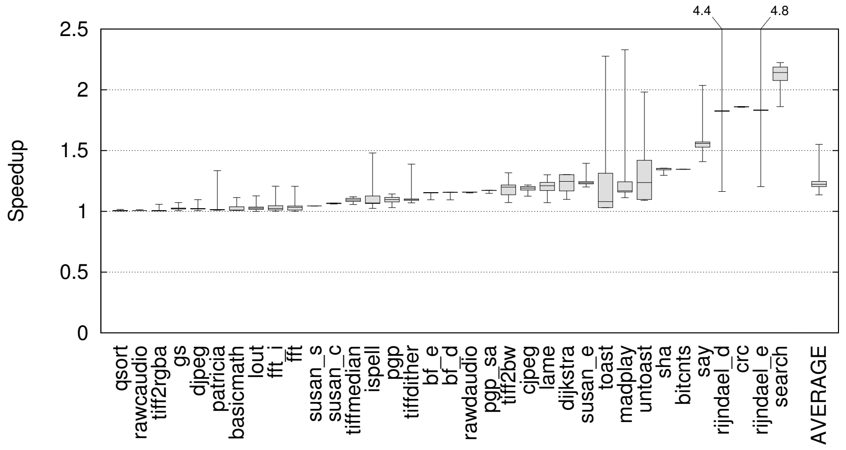


Figure 6.2: Distribution of the *maximum* speedup available across all architectures on a per-program basis. The x-axis represents the program and the y-axis the speedup relative to **-O3**. The central line denotes the median speedup. The box represents the 25 and 75 percentile area while the outer whiskers denote the extreme points of the distribution.

better than **-O3**. This shows that **-O3** is actually a good baseline optimisation level since any other optimisation settings on any architecture for any program would not perform much better on average. Finally, the distribution shows that there is considerable room for improvement over compiling with **-O3**, justifying further investigation.

6.3.2 Best Optimisation Distribution

Having seen what are the effects of applying a random set of optimisations, this section focuses on what is the *best* performance achievable for each program on each architecture in the sample optimisation space of 1000 optimisations. Figure 6.2 shows again a distribution of speedups, this time corresponding to selecting the best set of optimisations per program per architecture. Therefore the middle bar in each box corresponds to the average performance found when applying the *best* flag settings for each architecture.

As before, there is significant variation across the programs. For many the performance improvement is modest; selecting the best optimisations does not help for the library-bound benchmarks *qsort* and *basicmath*. Once again *rijnadael_e* has significant performance outliers ranging from a 1.2x speedup to 4.8x in the best case, 1.8x being the average. In the case of *search* the extremes are much less but on average selecting the best optimisation gives a 2.2x speedup across all configurations. In programs such as *toast*, *madplay* and *untoast*, there are modest speedups to be gained on an average architecture (as the middle bar shows) but

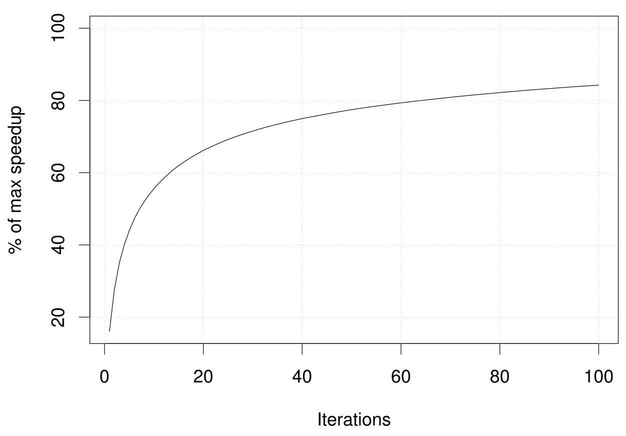


Figure 6.3: Performance of iterative compilation as a percentage of the maximum speedup available (from the sample space) over **-O3** for the first 100 iterations. The values shown are average across all programs and architectures.

significant improvements available on certain architectures (up to 2.4x for *madplay* as the top whisker shows).

The most important entry is on the right-hand side, giving the average performance. It shows that there is an average speedup of 1.23x available across the programs and architectures if one was able to select the best optimisations. The challenge consists of automatically developing a compiler that can do better than random (1.02x average speedup) and approach the speedup found by selecting the best of 1000 executions random compiler flags (1.23x average speedup). Furthermore, it should be able to capture the high performance available on certain architectures and avoid the large slowdowns found by picking the wrong optimisations.

Before looking at how to build a compiler that can achieve such performance, the next section reviews the performance that iterative compilation would achieve.

6.3.3 Performance of Iterative Compilation

An obvious and simple way of trying to achieve good performance consists of applying iterative compilation. This technique randomly chooses some compiler flags, compiles the program with them, runs it and keeps repeating this until no further improvement can be achieved. Figure 6.3 shows the average performance achieved by such a technique for a maximum of 100 iterations. The results are averaged across the 35 programs and 200 architectures from the sample space. As can be seen after 20 iterations, it achieves 66% of the total performance available and with 60 iterations, it gets about 80%. Obviously this method requires many simulations for every new program one wants to compile on a particular architecture.

What is needed is a technique that can come close to this performance for any microarchitecture, without the overhead of having to run each program many times. The next section describes a new machine-learning compiler that achieves precisely this. It also includes a detailed description of the model used internally by this compiler.

6.4 Designing a Trans-Architecture Learning Compiler

This section presents a novel optimising compiler: the trans-architecture learning compiler (TALC). This compiler is designed to produce *optimised* code for any new unseen program on any variation of a given microarchitecture. Such a compiler effectively adapts its compilation strategy based on the program behaviour and architecture features.

6.4.1 Overview of TALC

Figure 6.4 gives an overview of the structure of TALC. The compiler works like any other, taking as an input the source code of a program and producing an optimised binary as its output. However, in addition to the source code, this compiler has two other inputs which it uses internally to optimise the program specifically for the machine it will run on.

Firstly, the compiler takes in a description of the architecture to target. This is similar to standard compilers where the architectural description is hard-coded in a machine description file; here it is just an input. Secondly, it takes in performance counters derived from a previous run of the program. Again, this is similar to feedback-directed compilers that typically use profiling information from a previous run to generate an optimised version of the program. However, unlike any existing technique, TALC generates an optimised binary specifically for the target microarchitecture, even when it has never seen the program or the microarchitecture before. Therefore, the compiler does not have to be modified or regenerated whenever a new program or architecture is encountered. This is in stark contrast to all the previous compilers that need to be retuned for each new microarchitecture.

At the heart of this compiler is a model that correlates the behaviour of the new input program and architecture with programs and architectures that it has previously seen. That model is built using machine learning and can be considered as a three stage process: generating training data, building the model and deploying it. The next section looks at the input used by the model, after which these three steps will be described in detail in the subsequent sections.

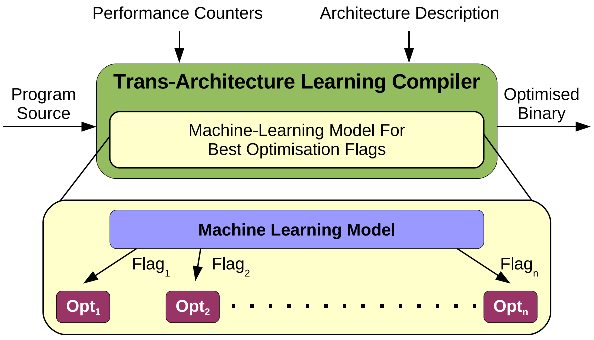


Figure 6.4: Overview of the trans-architecture learning compiler (TALC). The compiler takes in a program source, some performance counters extracted from the target architecture and an architecture description. It outputs an optimised program binary for that architecture. At the heart of the compiler lies a machine-learning model that predicts the best compiler flags to use, controlling the optimisations applied.

6.4.2 Input to the Model

In addition to a conventional compiler that takes as an input the program source, TALC captures information regarding the program and the target architecture. The idea being that when a new program and architecture is encountered, the compiler examines their features and compares it against prior knowledge in order to determine good optimisations based upon similar programs and architectures observed during training.

6.4.2.1 Architecture Description

To capture the features of an architecture k, its static description, *i.e.* the parameters shown in table 6.1, are simply captured as a vector $\mathbf{D}_k$. Therefore, $\mathbf{D}_k$ is an eight elements vector each of whose entries corresponds to one of the parameters shown in table 6.1. For example $\mathbf{D}_k = [32,32,32,32,32,32,512,1]$ corresponds to the description of the XScale architecture.

6.4.2.2 Performance Counters

Program interaction with the processor is characterised by 11 performance counters, $\mathbf{C}$. The performance counters are shown in table 6.4 and are similar to those typically found in processor analytic models [Eyer 06b, Kark 04]. These counters contain information about the resource utilisation, the level of parallelism, the memory accesses and miss rates.

Each vector of counters $\mathbf{C}_{j,k}$ is extracted from one run of each benchmark j compiled with **-O3** on each architecture k, where each entry corresponds to one of the values described in table 6.4.

Performance Counter
Instructions per cycle
ALU usage
Shifter usage
Mac usage
Decoder access rate
Register file access rate
Branch predictor access rate
Instruction cache access rate
Data cache access rate
Instruction cache miss rate
Data cache miss rate

Table 6.4: Performance counters used as a representation of program-architecture pairs.

6.4.3 Generating the Training Data

In order to build a model that predicts good optimisation flags, examples of good and bad optimisations on different programs and microarchitectures are needed. This training data is generated by applying f distinct compiler flag settings $\mathbf{F}_{i=1,\ldots,f}$ to b benchmarks $B_{j=1,\ldots,b}$ and running them on a architectures $A_{k=1,\ldots a}$ to obtain an execution time $T_{i,j,k}$ specific to each flag, benchmark and architecture. Now $\mathbf{F}_i$ is a vector where each entry describes the value for a specific optimisation flag. In the experimental setup there are 39 entries in $\mathbf{F}_i$, corresponding to the flags described in table 6.3. For example, $\mathbf{F}_i = [1,0,0,\ldots,0]$ corresponds to enabling the *thread-jumps* flag and disabling all others.

The relationship between the features, optimisations and execution time can be summarised as follows:

$$time(execute(compile(B_j,\mathbf{F}_i),A_k)) = T_{i,j,k} \tag{6.1}$$

$$counters(execute(compile(B_j,\textbf{-O3}),A_k)) = \mathbf{C}_{j,k} \tag{6.2}$$

$$description(A_k) = \mathbf{D}_k \tag{6.3}$$

In other words, compile benchmark B_j with compiler flags $\mathbf{F}_i$, execute it on architecture A_k and record its execution time $T_{i,j,k}$ (equation 6.1). In addition, record the performance counters $\mathbf{C}_{j,k}$ (equation 6.2) when compiling the same program with the default optimisation level **-O3** on the same architecture and record the description for this architecture $\mathbf{D}_k$ (equation 6.3). The result of executing all these programs and recording the generated information is a training set consisting of a architecture descriptions $\mathbf{D}$, f compiler flag settings $\mathbf{F}$, $f \times b \times a$ execution times T and $b \times a$ performance counters $\mathbf{C}$. Although this is a large training set, it is a one-off cost incurred “at the factory”. Techniques such as clustering [Phan 05] can be used to reduce the cost of gathering the training data. However this is orthogonal to the work presented in this chapter.

6.4.4 Building a Model

Once the training data has been gathered, a model is built which, given a new benchmark B^* and a new architecture A^*, predicts the best compiler flag setting $\mathbf{F}^*$. It uses the performance counters $\mathbf{C}^*$ extracted from a single run of program B^* compiled with **-O3** on architecture A^* plus its description $\mathbf{D}^*$. In other words the model to build, *predict*, is defined as:

$$predict(\mathbf{C}^*, \mathbf{D}^*) = \mathbf{F}^* \tag{6.4}$$

This problem is approached by learning the mapping from the features $\mathbf{C}, \mathbf{D}$ to a *probability distribution over good solutions*, $q(\mathbf{F}|\mathbf{C}, \mathbf{D})$. In other words, given performance counters $\mathbf{C}$ and architecture description $\mathbf{D}$ what is the probability that the flag setting $\mathbf{F}$ is a good solution? Once this distribution has been learnt (see next section), prediction of a new program on a new architecture is straightforward and is achieved by sampling at the mode of the distribution (*i.e.* taking the value of the flag that appears the most frequently in the distribution of good solutions). Therefore the predicted set of optimisation flags is obtained by computing:

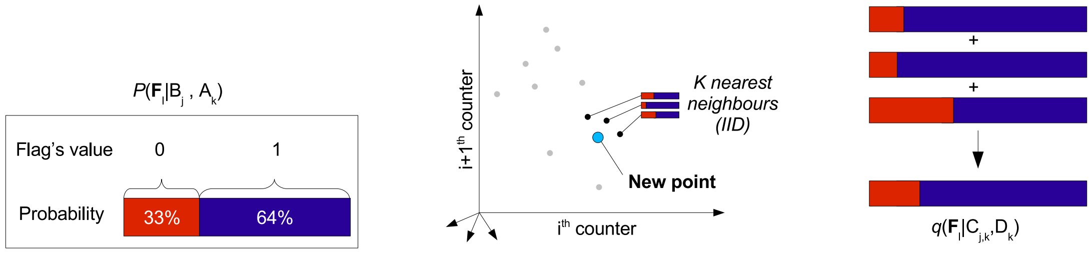
$$\mathbf{F}^* = \operatorname*{argmax}_{\mathbf{F}} q(\mathbf{F}|\mathbf{C}^*, \mathbf{D}^*). \tag{6.5}$$

This corresponds to finding the value of $\mathbf{F}$ that gives the greatest probability of being a good optimisation.

6.4.4.1 Fitting Individual Distributions

In order to learn the model a probability distribution need to be fitted over good solutions for each training program/architecture pair. The set of “good” solutions is chosen to be the optimisation settings that are within a threshold of 5% of all training optimisations for the specific program/architecture pair. This threshold ensures that enough data points are available for training the model.

The distribution fitted to the good solutions on each training program/architecture pair is denoted by $P(\mathbf{F}|B, A)$. In principle, many different distributions can be fitted to this data. However, the simplest of these was used: the IID (independent and identically distributed) model. In other words, the probability of a good set of optimisation flags is simply the product of each of the individual probabilities corresponding to how likely each flag is to belong to a good



(a) Individual distribution (IID)

(b) K-Nearest Neighbours

(c) Predictive distribution

Figure 6.5: Building the machine learning model. First the IID are learnt for each flag for each pair of program/architecture from the training data (a). Then the performance counter of a new point in the space (a new program/architecture pair) are extracted allowing to find the K nearest neighbours (b). Once the closest neighbours have been identified, their corresponding IID can be combined in order to get the predictive distribution of the new program/architecture pair (c).

solution:

$$P(\mathbf{F}|B,A) = \prod_{\ell=1}^{L} P(F_\ell|B,A), \tag{6.6}$$

where L=39 is the number of available flags.

Hence, for a given benchmark j and an architecture k, a distribution is learnt for each flag l, as illustrated in figure 6.5(a) which shows such a distribution for a binary flag. For each possible value of the flag, the distribution return the probability that this value will lead to good performance. In fact in this case, this probably is simply equal to the percentage of time the value of the flag appears in the set of good optimisation settings (top 5%).

This model assumed that the effect of each flag (F_ℓ) is considered to be independent of all others. While this seems like a rather strong assumption, the model achieves reasonable performance as it will be shown later. Nonetheless if complex interactions do exist between compiler flags, the model can be adapted and is expected to achieve similar performance.

Note that in fact some flags are directly dependent on others. For instance the flags that control the unrolling heuristics (for example *max-unroll-times*) only make sense when the flag that controls whether unrolling is applied or not is turned on. In this case, if the model predicts that unrolling should not be applied, then the values of the flags that control the heuristic are simply dismissed, since the optimisation is not applied. With this approach the IID model can still be used even in the presence of dependencies between flags.

6.4.4.2 Learning a Predictive Distribution Across Programs and Architectures

Once the individual training distributions $P(\mathbf{F}|B,A)$ are obtained, a predictive distribution $q(\mathbf{F}|\mathbf{C},\mathbf{D})$ can be learnt, conditioned on the performance counters $\mathbf{C}$ and architecture descriptors $\mathbf{D}$. This enables generalisation across programs and architectures. One possible way of learning this distribution is to use a memory-based methods such as K-Nearest Neighbours.

With the K-Nearest Neighbours approach, the predictive distribution $q(\mathbf{F}|\mathbf{C},\mathbf{D})$ can be set to be a convex combination of the K distributions corresponding to the training programs and architectures that are closest in the feature space to the new (test) program and architecture. This is illustrated in figure 6.5(b) where, given a new point in the design space (a new program and a new architecture), the K nearest neighbours are found. Given the corresponding probability distribution (IID), they can be combined in order to obtained the predictive distribution $q(\mathbf{F}|\mathbf{C},\mathbf{D})$ as seen in figure 6.5(c). This predictive distribution is in fact a weighted sum of the K nearest neighbours distribution.

6.4.5 Deployment

Once the model is built, it can be used to predict the best optimisation flags for any new program on any new architecture. It does this using just one run of the new program compiled with **-O3** on the new architecture. Therefore, given a new benchmark B^* and a new architecture A^* with its architecture description $\mathbf{D}^*$, equations 6.2, 6.3 and 6.4 above can be used to derive the predicted-best optimisation flags $\mathbf{F}^*$. The program is then compiled with this new, predicted optimisation setting. Before experimentally evaluating this approach, the next section will first consider the optimal parameters of the model, *i.e.* the number of nearest neighbours K, and the performance of the selected features.

6.5 Evaluation of the Model Parameter and Features

Having described the machine learning model on top of which the TALC compiler is built, this section evaluates the optimal choice of K (the number of nearest neighbours) and the quality of the features used. In particular, a comparison is conducted with the architecture-independent features already mentioned in chapter 4. First, the evaluation methodology is explained in the following section.

6.5.1 Evaluation Methodology

Cross-validation To evaluate the impact of the model parameter K and the features, leave-one-out cross-validation is used. This means that each program and architecture are removed temporarily from the training set, one by one. A model is built using the remaining training set, which includes all the architecture/program pairs from the sample space, except the program and architecture removed. Then the model is used to predict the best optimisations for the removed program for each architecture. Since the model is making a prediction for each flag separately, it follows that it is very unlikely that the predicted flag settings lie within the sample space. This means that for every new prediction one wishes to evaluate, a compilation followed by a run of the program is needed.

Ranking Since this section evaluates the model for different choices of K, the number of nearest neighbours, it is impractical to compile/run each prediction for every architecture, every program and every choice of K. Instead of actually running the program with the predicted flags, a ranking approach can be used to evaluate the quality of a prediction. Using the 1000 flag settings already collected in the sample space for each program/architecture pair, it is possible to estimate what would be the performance of a predicted flag setting without having to run any more simulations. This ranking consists of first selecting only the flags that are most likely to affect performance (section 6.7.1 will give more detail about this). Once this set of flags has been established for each program/architecture pair, the ranking simply consists of ordering all the 1000 flags by attributing one vote to each flag settings that contain a flag with the corresponding value equals to the predicted value. The flag setting with the most votes will be ranked first, while the one with the least will be ranked last.

Given the ranking of the flag settings for each program/architecture pair, the performance value corresponding to the predicted flag setting is simply estimated by using the highest ranked flag setting. With this approach, no extra compilation/simulation is required. Therefore, it is a fast and convenient way of evaluating the performance of the model. However, for the sake of completeness, a proper evaluation of the model is conducted in section 6.6 where each prediction results in a real compilation/simulation for each program/architecture pair.

6.5.2 Optimal Number of Neighbours

With the K-Nearest Neighbours approach, it is important to fix the parameter K properly in order to get good performance in terms of prediction accuracy. Figure 6.6 shows the performance of the model as a function of K. The case where $K = 0$ corresponds to a random prediction, since the model does not use any information from the training data. The performance is ex-

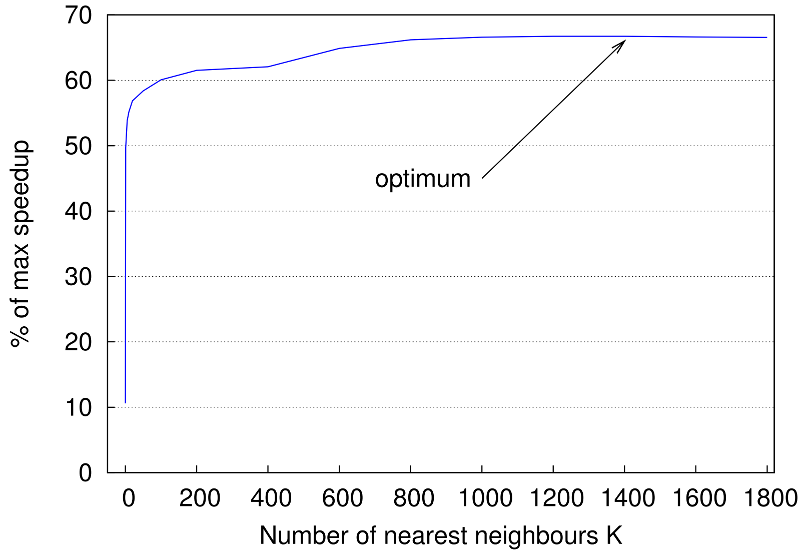


Figure 6.6: Performance achieved by TALC for different values of K; the number of nearest neighbours used by the model. The performance is expressed as the percentage achieved of the maximum speedup available in the sample space. The optimum value of $K = 1400$ lead to 67% of the speedup available.

pressed as the percentage of the maximum speedup available reached by the predicted flag settings, averaged across all program/architecture pairs from the sample space.

As can be seen, the performance tends to be fairly stable for $K \geq 800$. The actual optimum is reached for $K = 1400$ which achieves 67% of the maximum speedup available in the sample space. When K is increased to higher values, the performance decreases. This happens because the higher the value of K, the more generic the model becomes. On the one hand, if K was to be set to its maximum, the nearest neighbours would include all the points from the training set, thus averaging everything and simply predicting a flag settings that achieved the best speedup on average. On the other hand, for small values of K, the model would become too specific and would overfit. For this reason a choice of $K = 1400$ represents a good trade-off between both extremes and offers the best in terms of performance, as demonstrated by figure 6.6.

6.5.3 Efficiency of Features

As seen in the previous chapters, the choice of input features for a machine learning model often has a great influence on performance. For this reason, the features used as an input for TALC, described in section 6.4.2, are evaluated against two alternative feature sets.

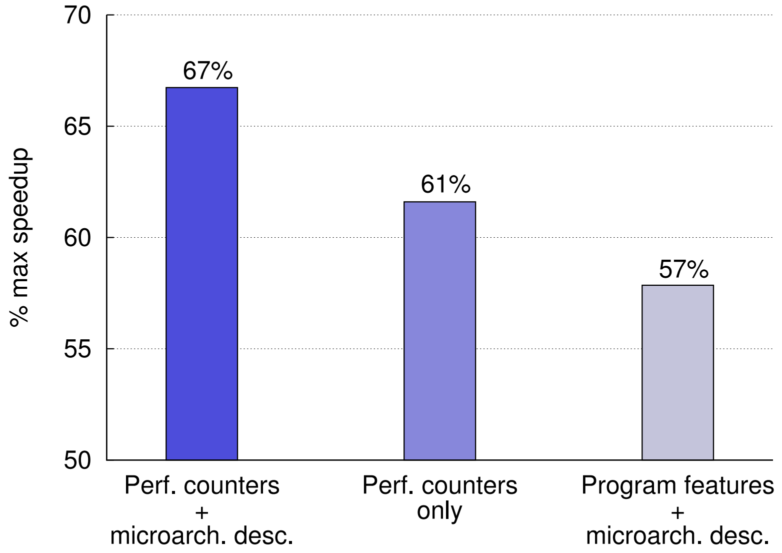


Figure 6.7: Comparison of three different feature sets. The first one is the one used by TALC, composed of the performance counters and the architecture description. The second set is a reduced set where the architecture description was removed from the features. The last set consists of microarchitectural-independent features.

Performance Counters Only The first of these sets consists of the performance counter information only, where the architecture description is dropped. So for any new program and architecture, the program is run on the target system with **-O3** and performance counters are extracted. These performance counters serve, as previously seen, as an input to TALC, however the architecture description is not used as an input to the compiler. This is done in order to determine whether the performance counters alone are sufficient to capture information about the architecture and to estimate how much information is actually contained in the architecture description.

Microarchitectural-Independent Features The second approach uses microarchitectural-independent features extracted only once for a new program. This extraction is comparable to a profile run and is done on an host machine. These features were developed by Eechkout *et al.* [Eeck 02] and have already been described in chapter 4, section 4.7. In addition to these features, the architecture description was added to the feature vector. This is necessary since the features are completely independent from the architecture and, therefore, the model would always yield the same prediction independently of the microarchitecture.

The comparison between these two feature sets and the one presented in the previous section was conducted using the ranking approach. The result obtained with cross-validation using

$K = 1400$ neighbours, was once again averaged across all the program/architecture pairs. Figure 6.7 shows the average performance achieved by each of the three different feature sets. As reported before, the performance of the original feature set, which includes the performance counters and the architecture description is 67%. Interestingly, when the archictecture description is removed, the performance dropped to 61%. This means that the architecture description contains additional information that cannot be extracted from the performance counters alone.

It is easy to imagine how this information about the architecture description improves the performance of the model. Consider, for instance, the cache miss rate which is included in the performance counters. This information is somehow incomplete when microarchitectures are compared to each other, since the size of the instruction cache itself is unknown. Conversely if you know that a given program has a given cache miss rate when the cache size has a particular value, this information becomes then more meaningful.

Looking back at figure 6.7, it can be seen that the last technique, which makes use of the microarchitectural-independent program features performs worse than the two others. It achieves on average 57% of the speedup available. While this performance is reasonable, it clearly shows that having features directly extracted from the target system is certainly an advantage. This is in fact a recurrent problem with architecture-independent features as seen already in chapter 4.

In the rest of this chapter, all the experiments are conducted using the features that include the performance counters and the architecture description, since they are superior to the other two approaches. The next section experimentally validates the model. In contrast with this section, which uses a ranking approach to evaluate the performance of the model, the next section actually compiles and runs the programs with the predictions made by the model for each pair of program/architecture.

6.6 Experimental Evaluation of the Model

In this section, TALC is evaluated experimentally by compiling and running the predictions made for each program/architecture pair resulting from the cross-validation methodology. The results of these experiments are directly compared with the *gcc*'s highest default optimisation level, **-O3**, and to the best performance available in the sample space (using iterative compilation with 1000 random optimisation settings).

Because the performance of the model might depend on the program or the architecture under consideration, this section evaluates the model in three phases. First the performance of the model is evaluated across the program space where the results are averaged across the

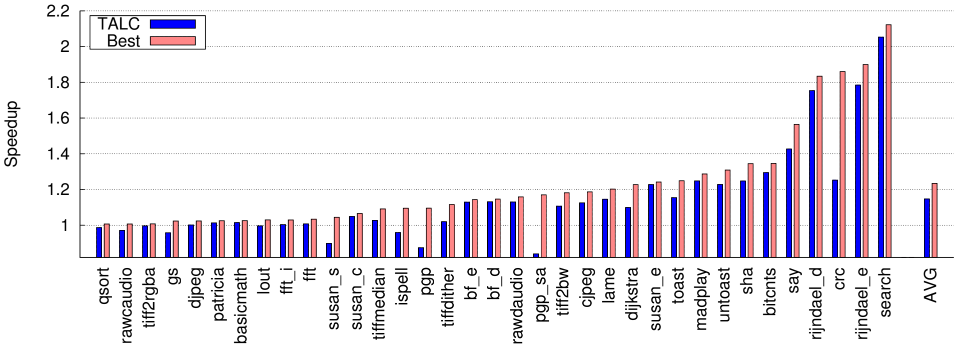


Figure 6.8: The performance of TALC and the best optimisations achieved by iterative compilation using 1000 random samples for each program, normalised to **-O3** and averaged over all microarchitectures.

architectures. In a second step, the evaluation is performed across the architecture space, when averaged across programs. Finally, the last stage consists of a 3D surface plot of the sample space, when both program and architecture are shown without any averaging.

6.6.1 Evaluation Across Programs

As explained, the model is first evaluated across the program space only. Figure 6.8 shows the performance of each program when optimised with TALC, relative to compiling with **-O3**, averaged across all architecture configurations. The second bar, labelled *Best*, is the maximum speedup achievable for each program. On average, TALC is able to achieve a 1.14x performance improvement across all programs and architectures with just one profile run, achieving up to 2.05x speedup for *search*.

For three benchmarks in particular (*search*, *rijndael_e* and *rijndael_d*), TALC achieves significant speedups, approaching the best performance available. Figure 6.8 shows that the model is able to correctly identify good optimisations, allowing these programs to exploit the large performance gains when available.

However, figure 6.8 also shows that several programs experience slowdowns compared with **-O3**. Considering *pgp_sa* for example, TALC achieves a 0.84x speedup: a slowdown of 16%. This can be explained by looking back at figures 6.1 and 6.2. These figures show that the majority of the optimisations are detrimental to this program and that there is little room for improvement over **-O3**. Due to this, it is very difficult for TALC to beat **-O3**. The same is true for programs *pgp* and *susan_s* for which TALC also leads to a slowdown.

Considering TALC compared to the maximum speedup achievable, it approaches *Best* in

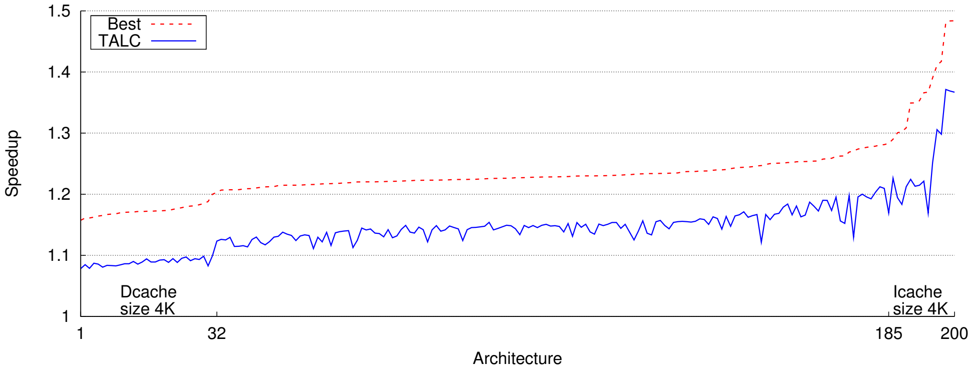


Figure 6.9: The performance of TALC and the best optimisations achieved by iterative compilation for each microarchitecture, normalised to **-O3** and averaged across programs.

most cases. For some programs, such as *susan_e*, it achieves over 94% of the maximum performance. However, for *crc* it achieves only 29%. The reason for this shortfall is caused by a subtlety in the source code of *crc*. The main loop within this benchmark updates a pointer on every iteration, resulting in a large number of loads and stores. By performing function inlining and allowing a large growth factor (flag *max-inline-insns-auto*), this pointer increment is reduced to a simple register addition which in turn reduces the number of data cache accesses. For this benchmark, it is clear that the performance counters are not sufficiently informative to enable TALC to capture this behavior. This prevents the model from selecting the best flags. However, the addition of extra program features would enable TALC to pick this up and is future work.

6.6.2 Evaluation Across Architectures

Now that the evaluation has been conducted across the program space, this section focuses on the architecture space, shown in figure 6.9. This time the best performance available is shown for each architecture, averaged across programs, with the line labelled *Best*. The microarchitectural configurations are ordered in terms of increasing speedup available over **-O3**. Those on the left have little speedup available whereas those on the right can gain significantly.

For TALC the amount of improvement over **-O3** varies from 1.08x to 1.37x. This gives an average speedup of 1.14x across all programs and architectures. It is important to see that TALC closely follows the trend of the *Best* optimisations; the difference between the two lines (*Best* and *TALC*) is more or less constant. This shows that this model successfully captures the variation between configurations, exploiting architectural features when performance improvements can be achieved.

Looking at figure 6.9 in more detail, it can be seen that it is divided into roughly three sections. On the left, up to configuration 32, there is little performance improvement available. All microarchitectural configurations in this area have a small data cache of just 4K. *Gcc* has very few data access optimisations, therefore the available speedups are relatively small. After this is the second section where the *Best* optimisations gain 1.23x speedup and TALC manages to capture a respectable 1.14x.

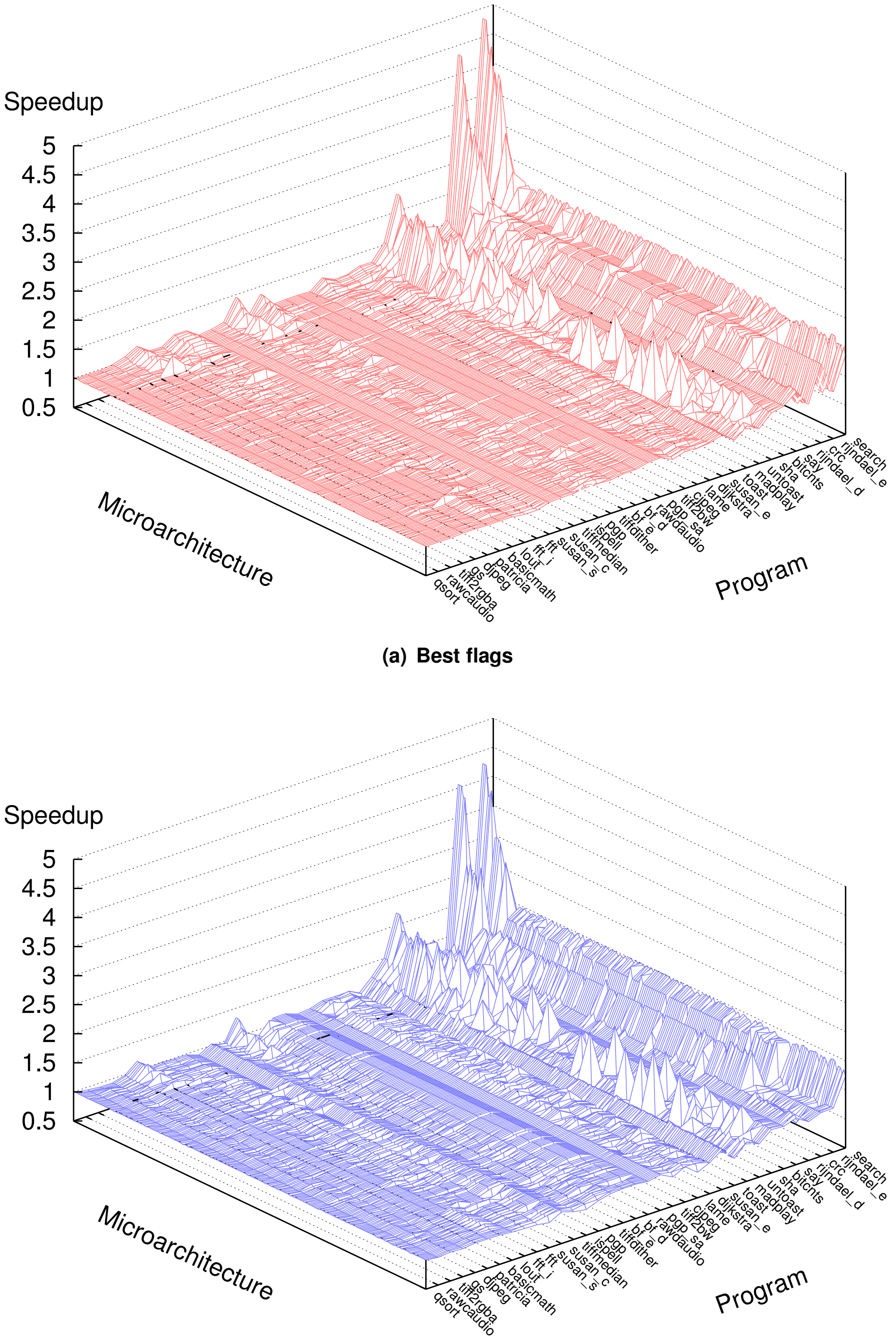
Finally in the third section, after configuration 185, the available performance improvement increases dramatically. These architectures on the right have a small instruction cache of just 4K, meaning that it is important to remove code duplication wherever possible. The performance counter specifying the instruction cache miss rate enables the model to learn this from the training programs. In particular, TALC learns that instruction scheduling (*schedule-insns*) and function inlining (*inline-functions*) must be disabled to prevent code size increases. In the case of instruction scheduling, this increase is due to a subsequent register allocation pass which emits more spill code for certain schedules. This is a typical example of the effect of the complex relationships between passes within the compiler. Nonetheless, the model is able to cope with these interactions and achieve the majority of the speedups available in this area.

Overall the maximum speedup available is on average 1.23x and TALC achieves a 1.14x speedup with just one profile run across all microarchitecture configurations and programs. This is equivalent to 61% of the speedup achieved by the *Best* flags and is roughly consistent across the architecture configuration space.

6.6.3 Program/Architecture Optimisation Space

The quality of the predictions made by TALC has been considered separately across the program and architecture space, showing that TALC can, in most cases, fully exploit the amount of available speedups across programs and architectures. Figures 6.8 and 6.9 showed that on average, this approach performs well and achieves impressive performance improvements. In this section, the performance of TALC is shown for each combination of program/architecture.

Figure 6.10 shows the performance achieved by the best flags and by using TALC across both the programs and the microarchitectural configurations. These graphs show in detail the information that is summarised in figures 6.8 and 6.9. The benchmarks are ordered as in the former figure, so that those with large performance increases (such as *search*) are on the right. The microarchitectural configurations are ordered as the latter, so that those with large speedups available over **-O3** are on the left. As figure 6.10 shows, the surface generated when using TALC is almost identical to that generated when using the best compiler flags. The model is highly accurate at predicting very good compiler settings across the programs or architecture.



(a) Best flags

(b) TALC predictions

Figure 6.10: Speedup over **-O3** for each program/architecture pair. The top diagram shows the best improvement possible over the programs and architectures. The bottom figure shows the performance of the optimisation flags predicted by TALC.

In the back corner, the maximum speedup achievable with the best compiler flags is obtained by *rijndael_e*. This benchmark achieves a factor 4.85x speedup on an architecture with a 4K instruction cache size. The optimisation flags leading to this result prevent the compiler from performing any loop optimisations (apart from moving loop-invariant code out of the loops). In particular, the flags prevent any loop unrolling from being performed because there is already extensive, optimised software loop unrolling programmed into the source code. As can be seen, TALC is able to achieve an impressive speedup of 4.84x for the same architecture and program.

For all program/architecture pairs with large performance improvements available, TALC is able to achieve significant speedups, as shown by the peaks for programs *ispell*, *madplay*, *rijndael_d* and *rijndael_e*. These graphs clearly demonstrate that this model not only achieves good average performance but is also able to capture the variation in speedups available across the program and architecture spaces.

6.6.4 Summary

This section has shown that TALC achieves an average 1.14x speedup over the highest default optimisation level, **-O3**, across the entire architecture space for the MiBench benchmark suite. It is able to reach this performance with just one profile run, achieving 61% of the maximum speedup available if one were to use iterative compilation with 1000 evaluations. In addition, this approach is able to achieve higher levels of performance whenever they are available, accurately exploiting the compiler optimisation space. This is all achieved with a one-off training cost incurred “at the factory” which can be used within the trans-architecture learning compiler for any variation of the architecture within the design space. The next section analyses the results, describing the flags that are important in the sample space and how the model selects good optimisiation flags for new programs and architectures.

6.7 Analysis of Results

This section shows how the model developed in this chapter manages to achieve almost the best performance available in most cases. It does so by analysing, in more depth, the interaction between the compiler optimisations and the microarchitecture design space. First, an analysis of the importance of each flag is conducted, which reveals which flags have a real effect on performance. In a second step, it shows that the values for these flags leading to the best performance are different for each program. Then the relation between the features and the flags leading to good performance is examined. Finally, a detailed analysis of the interaction

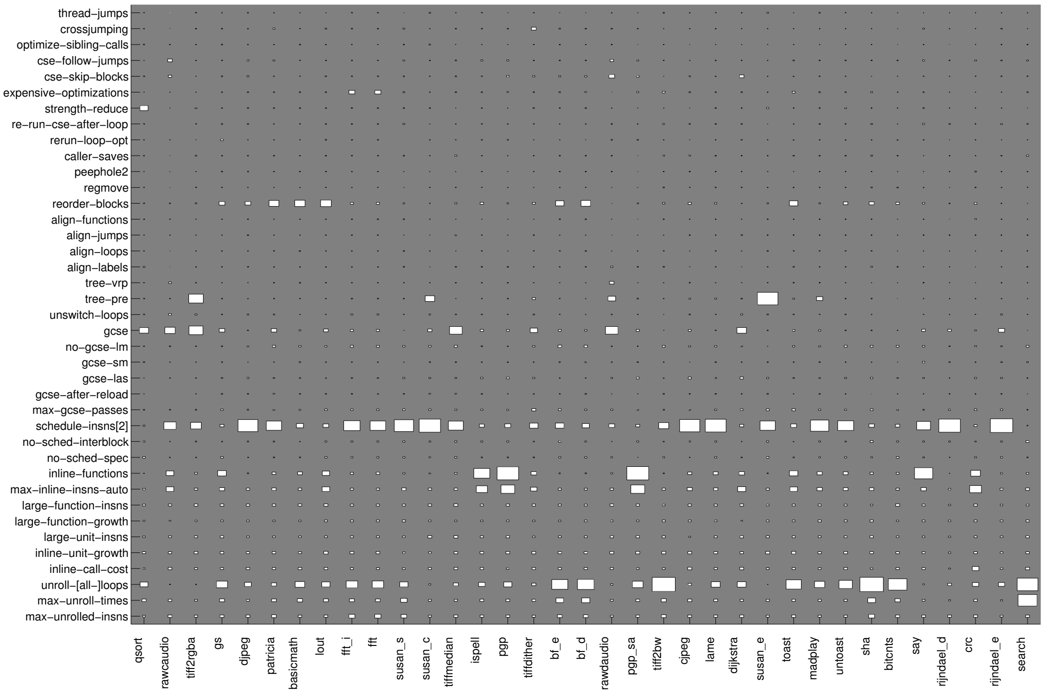


Figure 6.11: A Hinton diagram showing which flags are most likely to affect performance for each benchmark. The larger the box, the larger the mutual information is. *I.e.* the more likely a flag is to affect the performance of the relevant program.

between the compiler flags and the architectures is presented for one benchmark, concluding this section.

6.7.1 Flags Likely to Affect Performance

The previous section showed that TALC's performance closely follows the speedups achieved by the best flags found by iterative compilation for each program/architecture pair. This section considers how it achieves this by looking at the flags that are most likely to affect performance for each benchmark, explaining why programs require different flags to be enabled in order to get good performance. Figure 6.11 displays a Hinton diagram of the normalised mutual information between each flag and the speedups obtained by each program, calculated by the model based on the training data. This shows the flags for each program that are most likely to affect performance. The larger the box, the greater the impact of the flag. Conversely a small box indicates the flag setting does not matter.

From this graph, it is clear that some flags are important across all programs, whereas others are not important to any, or are only important to a few benchmarks. For example, instruction scheduling (*schedule-insns*) is important for almost all benchmarks. As discussed in

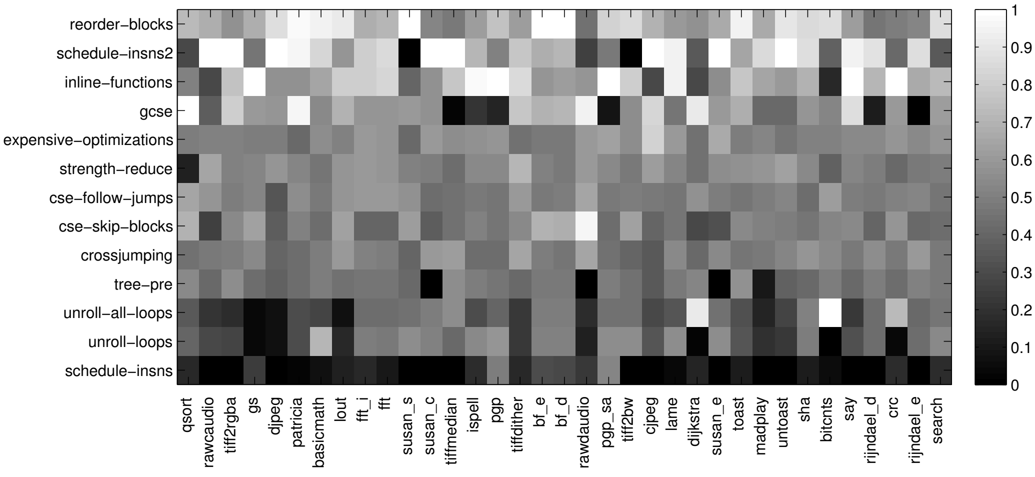


Figure 6.12: Distribution of important flags for each program (averaged across architectures). For each flag/program pair the distribution represents the probability that the given flag is enabled within the top 5% of the best performing optimisation settings from the sample space independently of the architecture.

section 6.6.2, in some cases this optimisation has a negative effect on architectures with a small instruction cache. Loop unrolling (*unroll-loops*) is also an important flag for many programs. For programs such as *search*, it is important to consider this flag to reach good performance because it contains loops with a known number of iterations that can be optimised. However, for others, such as *rijndael_e*, this optimisation flag does not play a critical role in achieving good performance because extensive unrolling is already implemented in the source code.

Looking at the mutual information shown in figure 6.11, it can be seen how the model focuses on those flags that are most likely to affect performance. The next section analyses why specific flag values are important and how these values change with the program under consideration.

6.7.2 Important Flags' Probability

This section provides insight into why some flags are more important than others and how likely they are to be enabled. This is indeed important to understand how TALC manages to learn to make accurate predictions.

Figure 6.12 shows the probability of each flag selected by the model being enabled for each program to achieve good performance, averaged across all microarchitectural configurations. The lighter the rectangle is, the greater the number of architectures that will benefit from having the specific flag enabled. The flags are ordered such that, on average, it is more important to enable the flags at the top and more important to disable those at the bottom.

In general, figure 6.12 shows the value that these flags take varies significantly between programs. As previously discussed in section 6.6.1, for *crc* it is important to enable function inlining (*inline-functions*) because this allows the compiler to convert an increment to pointer data into a register increment. Also notice that of the two variations of instruction scheduling (*schedule-insns* and *schedule-insns2*), the first should generally be disabled and the second enabled. This is because the first variant is performed before register allocation whereas the second is executed afterwards. Therefore, *schedule-insns* can result in extra spill code being generated by the register allocator, as previously mentioned in section 6.6.2.

Looking at the program *bitcnts* in figure 6.11, loop unrolling is identified as an important flag. As it can be seen in figure 6.12 it is important to disable the first variant (*unroll-loops*) and enable the second (*unroll-all-loops*) which unrolls all loops, whether their number of iterations is known or unknown (as opposed to the first variant which only unrolls loops for which the number of iterations is known). Unrolling loops is important for *bitcnts* because it reduces the number of executed branches. This offsets the small number of extra instruction cache misses that occur from the additional unrolled instructions. Enabling *unroll-loops* does not allow the compiler to unroll at all because the loops in *bitcnts* are predominantly *while* loops and the compiler does not know their iteration count for any of them. Therefore this variant should be disabled. However, *unroll-all-loops* causes *while* loops to be unrolled too, therefore this variant should be enabled to obtain the best performance. The model captures this through the performance counter specifying the branch predictor access rate and identifies *dijkstra* as a benchmark with similar characteristics.

6.7.3 Important Flags and Features Relation

Before looking at how the values of some flags varies across the architecture space for one particular benchmark, this section evaluates the relationship between the features and the important flags. Figure 6.13 shows the mutual information shared between the flag values that lead to good performance and the features.

As can be seen the most important features, on the left half of the figure are all dynamic information extracted from the performance counters. It is not surprising to see that many of them seem equally important since most of these counters are in fact correlated with each other. Moreover, the performance counters are different from program to program, allowing one to distinguish between programs. This contrasts with the architecture description on the right half of the figure since they are obviously only dependent on the architecture and not the programs. However, as was shown in section 6.5.3, the architecture description is nonetheless important

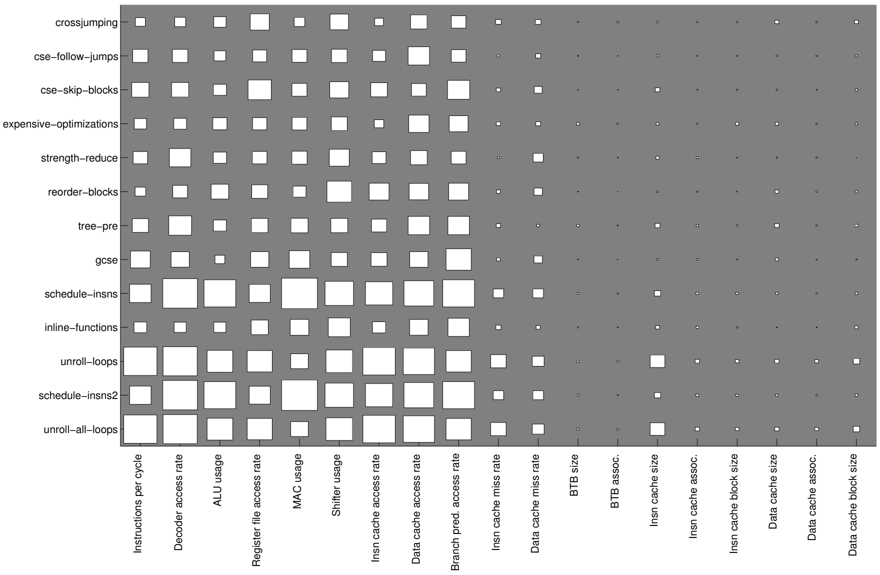


Figure 6.13: Hinton diagram representing the mutual information between the flags values leading to the best performance and the features. Each square represent how informative a specific features is for one specific flag. In other words it shows whether a given features is useful at predicting a given flag.

and contains valuable information that the model can use.

It is interesting to notice that among the architecture description features, the instruction cache size seems to contain important information about some optimisation flags. In particular *unroll-loops*, *schedule-insns2* and *unroll-all-loops*. This is in fact not surprising since, as it was shown earlier, these optimisations tend to increase the number of instruction cache misses. Hence, knowing the value of the instruction cache size helps determining whether these optimisations should be enabled or not.

6.7.4 Detailed Analysis of One Benchmark

Having seen how the flag values leading to good performance differ from one program to another, this section considers their variation across the architecture space. Unfortunately, it is difficult to really understand what is going on when results are summarised or averaged across different programs or architectures. Therefore, this section considers only one benchmark, *rijndael_d*, and analyses how the flag values vary across architectures.

Figure 6.14 shows the performance achieved by TALC for each architecture for the program *rijndael_d*. On average this is 90% of the best performance available. In addition to this, it shows how the values of the important flags vary according to the microarchitectural config-

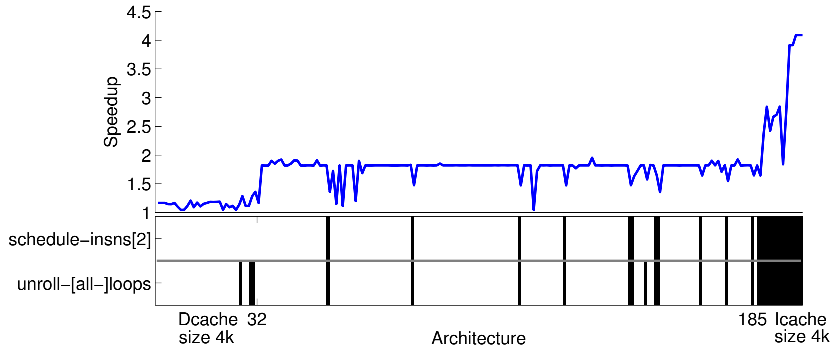


Figure 6.14: Performance of TALC for *rijndael_d* across the architecture space normalised by **-O3**. Also shown are the important flags whose values vary significantly depending on the architecture.

urations, which are ordered as in figure 6.9. Therefore the configurations on the far left have a 4K data cache size whereas those on the right have a 4K instruction cache size. The white areas for the flags show that these optimisations should be enabled whereas black indicates that the flags should be disabled.

It is clear from this graph that the large speedups available to this program are achieved when there is a small instruction cache due to the large number of misses that occur (architecture ≤ 32). For this program, loop unrolling (*unroll-loops* and *unroll-all-loops*) and instruction scheduling (*schedule-insns*) must be enabled on these configurations and disabled on all others. This is because these optimisations reduce code size and, therefore, instruction cache misses, as discussed in section 6.6.2. In fact, all the configurations for which unrolling is not performed have an instruction cache size of 4K for this benchmark (the black bars at the performance drops in figure 6.14). However, it is interesting to note that instruction scheduling should not be performed on this instruction cache size except when the data cache size is also 4K (around configuration 32). In this situation for *rijndael_d*, instruction scheduling reorders memory operations within the basic blocks which map to the same data cache location, reducing data cache misses. This again shows the complex interactions that exist between compiler and architecture, which TALC correctly identifies.

This section has shown that the values of the important flags change across microarchitectural configurations, adapting to characteristics such as cache size. The model developed in this chapter can correctly identify and learn these relationships in order to develop a compiler that automatically optimises any new program on any new microarchitecture.

6.7.5 Summary

This section has analysed the results of the experiments using TALC and shows why this approach works. It has identified the flags that are important in achieving good performance and examined their values on a per-program basis, identifying the program's features and performance counters that are used by TALC to capture the program behaviour. Finally, the flags affecting the performance of one program have been examined, showing how the values of these flags also vary across architecture configurations. Overall, TALC is able to accurately capture the variation in potential performance available across the program and architecture spaces.

6.8 Conclusions

This chapter has demonstrated that it is possible to build a compiler that can optimise new programs for any microarchitecture, given a parametrised design. This was achieved by developing TALC, a trans-architecture learning compiler. This compiler learns the optimisation settings leading to the best performance across microarchitectures and programs. A sample space composed of the Cartesian product of 200 distinct microarchitectures and 1000 optimisation settings was used to demonstrate the effectiveness of this approach.

Using a machine-learning model, an average speedup of 1.14x was achieved over the default best *gcc* optimisation level. As seen this corresponds to 61% of the maximum speedup available in the sample space. This approach shortens time-to-market in the design process of embedded processors. This represents an important step towards complete integration of processor design and compiler tuning. The fact that a parametrised processor design can be developed and a compiler automatically generated, that can adapt to any choice of microarchitectural parameters, is the main contribution of this thesis.

Chapter 7

Conclusions

This thesis has investigated how the design process of new processors and the generation of their associated optimising compiler can be made more efficient through the use of machine-learning. In particular, chapter 4 has presented predictive models that can speedup the design space exploration of new microarchitectures. The effects of compiler optimisations on the design process have been considered in chapter 5 where machine-learning has been used to automatically predict the joint microarchitectural and compiler optimisation co-design space. Finally, a new optimising compiler has been developed in chapter 6 that can find the right set of optimisations to apply to new programs for any microarchitectural point in the design space.

This chapter summarises the main contributions of this thesis in section 7.1, presents a critical analysis of this work in section 7.2 and discusses future work in section 7.3.

7.1 Contributions

This section summarises the main contributions of this thesis for processor design and co-design space exploration, and automatic optimising compiler generation.

7.1.1 Processor Design

Chapter 4 has explored a novel approach to design space exploration by building a model that uses program similarities. A first model was presented that is built offline and makes predictions for any new unseen program using only 32 simulations from it. Compared to previously proposed schemes [Ipek 06, Jose 06a, Lee 06], this model is the first to transfer knowledge about the design space between programs.

This model has then been used as the foundation of the benchmark-suite predictor: a novel model that predicts the average behaviour of an entire benchmark suite. This model dramatically reduces the number of simulations required, compared with two state-of-the-art approaches [Host 06, Ipek 06]. It has been shown that previous techniques, making use of microarchitectural-independent features, fail to capture a program's behaviour correctly. Conversely, the model developed in this thesis makes use of program responses extracted from a few select points in the output space.

Using only five representative programs from SPEC CPU 2000, this model accurately predicts the average behaviour of the full benchmark suite for cycles, energy, ED and EDD. It was also shown that it achieves the same error rate with five times fewer training simulations on the whole of the SPEC CPU 2000 and MiBench benchmark suites than any of the two other approaches [Host 06, Ipek 06]. Furthermore, it uses, asymptotically, an order of magnitude fewer simulations as the size of the benchmark suite increases.

Hence, the benchmark-suite predictor is a practical model that can be used to conduct efficient design space exploration of new microprocessors. Because it characterises programs using responses, it is able to automatically adapt to any design space and always selects the best representative programs, achieving a large reduction in the number of simulations whilst maintaining high accuracy.

7.1.2 Co-Design

Chapter 5 has first shown how compiler optimisations influence the design of embedded processors. Using a typical embedded processor, the XScale, an exploration of its microarchitecture and compiler optimisation spaces has been conducted separately. It was observed that significant improvements exist in the optimisation space and that this cannot be ignored at design time. For this reason, the co-design space has been explored and presented using a sample space composed of 200 microarchitectures and 1000 optimisation settings. It has been shown that failing to take into account compiler optimisations at the design stage can mislead the architect.

Because exploring the co-design space is not practical, a machine-learning model has been proposed. This model, trained on a fraction of the co-design space, makes accurate predictions of what the best performance achievable for any microarchitecture is, had the compiler been tuned for it. This gives a substantial advantage to the designer, who can use this model to make better design decisions. The model has been demonstrated to be accurate and has been compared against the state-of-the-art [Vasw 07]. This approach tries to address this problem by predicting the performance that any compiler optimisation will have on any microarchitecture. However, it has been shown that it fails to provide insightful information to the designer.

As seen, the current design methodology of embedded processors is suboptimal. When system performance is critical, compiler optimisations have an important role to play in helping meet the tight constraints of the embedded world. The work presented in this thesis has looked at integrating compiler optimisations directly into the design process of new microarchitectures.

7.1.3 Optimising Compilation

Finally, chapter 6 has demonstrated that it is possible to build a compiler that can optimise new programs for any microarchitecture, given a parametrised design. This was achieved by developing TALC, a trans-architecture learning compiler. This compiler learns the best optimisation settings to apply to achieve the best performance across microarchitectures and programs. A sample space composed of the Cartesian product of 200 distinct microarchitectures and 1000 optimisation settings was used to demonstrate that this approach actually works.

Using a machine-learning model, an average speedup of 1.14x was achieved over the default best *gcc* optimisation level. As seen, this corresponds to 61% of the maximum speedup available in the sample space. This performance was achieved by using a single profile run of the new program on the archictecture to be compiled for. Performance counters were used from this run, allowing the model to characterise the program and predict good optimisation flags leading to the best performance for the particular microarchitecture under consideration.

This novel approach has the potential to shorten time-to-market in the design process of embedded processors. More importantly, this represents an important step towards complete integration of processor design and optimising compiler generation. The fact that a parametrised processor can be developed and an optimising compiler automatically generated for it, that can adapt to any choice of microarchitectural parameters, is one of the main contributions of this thesis.

7.2 Critical Analysis

This thesis has investigated the design process of microprocessors and its interactions with compiler optimisations. This section now conducts a critical analysis of this work.

7.2.1 Simulation Methodology

All the experimental results presented in this thesis have been obtained through the use of simulators. The reason for this is that it is difficult to use a real processor for design space exploration. Indeed, real processors are fixed and, therefore, cannot be changed. The major problem with simulators is that they are only an approximation of reality. Even though their timing and energy models are validated against a real design, when one starts exploring the design space and gets away from the validated baseline design, inaccuracies will occur. As a result the design space obtained from simulation might not be representative of the real design space. The hope is that it is sufficiently similar to the real one and that the relative difference between the design points holds.

Ideally, each design point should be simulated at the Register Transfer Level (RTL). This is one of the most reliable ways to obtain accurate energy and cycle values. However, simulating entire programs at RTL takes a tremendous amount of time and resources. Therefore this methodology was not used in this thesis, since numerous simulations were needed to validate the approaches proposed. However, should this methodology be used, the savings obtained through the use of the newly developed techniques would be substantial.

7.2.2 Compiler Optimisations

The compiler optimisations used in chapters 5 and 6 were those available in *gcc*. This compiler was chosen since it is widely used and has a port for the ARM architecture simulated. However, amongst all the different optimisations considered, it was shown that only a handful of them interact with the microarchitecture. While this was sufficient to illustrate the point that interactions do exist between microarchitecture parameters and optimisations, it would have been desirable to have a larger base of optimisations. In particular, having better fine-grained control over the different optimisations and having more data transformations.

Furthermore, the optimisations were applied globally to the whole program. In practice, it is expected that better results could be achieved if the optimisations were applied at a finer granularity, for instance at the function level. Applying optimisations at a finer level could in fact be easier for a model to predict. However, this was not performed in this thesis since it would have been difficult to extract performance counters at such a granularity and therefore was not practical.

7.2.3 Use of Performance Counters vs Static Features

Chapter 6 has shown how information extracted from performance counters can be used in order to optimise programs. However, the use of performance counters requires a run of the application. While it might be perfectly feasible in some cases, it would be better to extract information that does not require this run of the program. For this reason, the use of static code features is an interesting direction. Moreover, this would allow the application of optimisations at a finer granularity and improve the whole process.

7.3 Future Work

This thesis has investigated how machine-learning techniques can be used to improve the design process of new microprocessors and automate the generation of optimising compilers. One important area of future research is that of multicore systems. These systems are more difficult to design than their unicore counterparts. Firstly, the communication channels between these cores are complex and can be implemented in many ways, including shared-memory or message-passing paradigms. Secondly, their number and organisations are critical in order to achieve high-performance and energy-efficiency. Finally, simulating such systems takes much longer than standard processors: diverse synchronisation mechanisms and coherence protocols are needed within the simulator which slows down the whole process.

The use of machine-learning would, therefore, be beneficial to these systems. It has the potential to realise high gains in terms of simulation time. One could predict, for instance, the behaviour of the whole system for a large number of cores, based on the performance of a lower number.

Another direction for future research is the extension of the techniques developed in this thesis for dynamic runtime adaption of software and hardware. The techniques considered in this thesis were only applied to find a fixed architecture or optimise the program once. However, since the behaviour of the program is likely to change over time, it would be beneficial to continuously adapt the hardware as well as the software.

Machine-learning has the potential to be beneficial to runtime adaptation. By studying the program's past behaviour, one can predict its future behaviour. For instance, repetitive program phases can be detected and optimised adequately. At the same time the architecture could also be modified on the fly to reflect these software changes and be more efficient.

Finally, the optimisation space can go beyond optimisation selection and integrate phase reordering. Techniques such as reinforcement learning could then be applied to develop an optimising compiler. These techniques would then drive the optimisation process and exploit the performance available.

Bibliography

[Abra 00] S. G. Abraham and B. R. Rau. "Efficient design space exploration in PICO". In: *Proceedings of the 2000 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2000.

[Agak 06] F. V. Agakov, E. V. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. "Using Machine Learning to Focus Iterative Optimization". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[Alma 04] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. "Finding effective compilation sequences". In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2004.

[Alpe 99] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Shepherd, and M. Mergen. "Implementing jalapeño in Java". In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-14)*, 1999.

[Beni 88] M. E. Benitez and J. W. Davidson. "A portable global optimizer and linker". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[Bish 06] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[Bodi 98] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. "Iterative Compilation in a Non-Linear Optimisation Space". In: *In Proceedings of the Workshop on Profile and Feedback-Directed Compilation*, 1998.

[Broo 00] D. Brooks, V. Tiwari, and M. Martonosi. “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations”. In: *Proceedings of the 27st Annual International Symposium on Computer Architecture (ISCA-27)*, 2000.

[Burg 97] D. Burger and T. M. Austin. “The SimpleScalar tool set, version 2.0”. *ACM SIGARCH Computer Architecture News*, Vol. 25, No. 3, 1997.

[Cava 04] J. Cavazos and J. E. B. Moss. “Inducing heuristics to decide whether to schedule”. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[Cava 06a] J. Cavazos, C. Dubach, F. V. Agakov, E. V. Bonilla, M. F. P. O’Boyle, G. Fursin, and O. Temam. “Automatic performance model construction for the fast software exploration of new hardware designs”. In: *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006.

[Cava 06b] J. Cavazos and M. F. P. O’Boyle. “Method-specific dynamic compilation using logistic regression”. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA-21)*, 2006.

[Cava 07] J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O’Boyle, and O. Temam. “Rapidly Selecting Good Compiler Optimizations using Performance Counters”. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2007.

[Cont 04] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G.-Y. Lueh. “XTREM: a power simulator for the Intel XScale® core.”. In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2004.

[Cont 96] T. M. Conte, M. A. Hirsch, and K. N. Menezes. “Reducing State Loss For Effective Trace Sampling of Superscalar Processors”. In: *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors (ICCD)*, 1996.

[Coop 05] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. “ACME: adaptive compilation made efficient”. In:

Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2005.

[Coop 99] K. D. Cooper, P. J. Schielke, and D. Subramanian. “Optimizing for reduced code space using genetic algorithms”. In: *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 1999.

[Duba 07a] C. Dubach, T. M. Jones, and M. F. P. O’Boyle. “Microarchitectural Design Space Exploration Using An Architecture-Centric Approach”. In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, 2007.

[Duba 07b] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O’Boyle, and O. Temam. “Fast compiler optimisation evaluation using code-feature based performance prediction”. In: *Proceedings of the 4th International Conference on Computing Frontiers (CF-4)*, 2007.

[Duba 08] C. Dubach, T. M. Jones, and M. F. O’Boyle. “Exploring and predicting the architecture/optimising compiler co-design space”. In: *Proceedings of the 2008 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2008.

[Eeck 02] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. “Workload Design: Selecting Representative Program-Input Pairs”. In: *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002.

[Eeck 04] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. D. Bosschere, and L. K. John. “Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies”. In: *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA-31)*, 2004.

[Emma 87] P. Emma and E. Davidson. “Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance”. *IEEE Transactions on Computers*, Vol. 36, No. 7, 1987.

[Eyer 06a] S. Eyerman, L. Eeckhout, and K. D. Bosschere. “Efficient design space exploration of high performance embedded out-of-order processors”. In: *Pro-*

ceedings of the conference on Design, Automation and Test in Europe (DATE), 2006.

[Eyer 06b] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. "A Performance Counter Architecture for Computing Accurate CPI Components". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-12)*, 2006.

[Fisc 01] D. Fischer, J. Teich, R. Weper, U. Kastens, and M. Thies. "Design space characterization for architecture/compiler co-exploration". In: *Proceedings of the 2001 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2001.

[Guth 01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the IEEE 4th International Workshop on Workload Characterization (WWC-4)*, 2001.

[Hane 05] M. Haneda, P. Knijnenburg, and H. Wijshoff. "Automatic selection of compiler options using non-parametric inferential statistics". *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT-14)*, 2005.

[Hart 02] A. Hartstein and T. R. Puzak. "The optimum pipeline depth for a microprocessor". In: *Proceedings of the 29st Annual International Symposium on Computer Architecture (ISCA-29)*, 2002.

[Henn 00] J. L. Henning. "SPEC CPU2000: Measuring CPU Performance in the New Millennium". *Computer*, Vol. 33, No. 7, 2000.

[Host 06] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere. "Performance prediction based on inherent program similarity". In: *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT-15)*, 2006.

[Inte 02] Intel Corporation. "Intel XScale Microarchitecture". http://www.intel.com/design/intelxscale/, 2002.

[Inte 07] Intel Corporation. "Intel Core Microarchitecture". http://www.intel.com/technology/architecture-silicon/core/index.htm, 2007.

[Ipek 05] E. İpek, B. R. de Supinski, M. Schulz, and S. A. McKee. “An Approach to Performance Prediction for Parallel Applications”. In: *Proceedings of the 11th International Euro-Par Conference (EuroPar-11)*, 2005.

[Ipek 06] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. “Efficiently exploring architectural design spaces via predictive modeling”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-12)*, 2006.

[Jose 06a] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. “Construction and use of linear regression models for processor performance analysis”. In: *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA-12)*, 2006.

[Jose 06b] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. “A Predictve Performance Model for Superscalar Processors”. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, 2006.

[Josh 06] A. Joshi, A. Phansalkar, L. Eeckhout, and L. John. “Measuring benchmark similarity using inherent program characteristics”. *IEEE Transactions on Computers*, Vol. 55, No. 6, 2006.

[Kark 04] T. S. Karkhanis and J. E. Smith. “A First-Order Superscalar Processor Model”. In: *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA-31)*, 2004.

[Khan 07] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. “Using Predictive Modeling for Cross-Program Design Space Exploration in Multicore Systems”. In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT-16)*, 2007.

[Kisu 00] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O’Boyle. “Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation”. In: *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2000.

[Kulk 04] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. “Fast searches for effective optimization phase sequences”. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[Lanc 67] G. N. Lance and W. T. Williams. “A general theory of classificatory sorting strategies 1. Hierarchical systems”. *The Computer Journal*, Vol. 9, No. 4, 1967.

[Latt 04] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization (CGO)*, 2004.

[Lee 06] B. C. Lee and D. M. Brooks. “Accurate and efficient regression modeling for microarchitectural performance and power prediction”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-12)*, 2006.

[Lee 07a] B. C. Lee and D. M. Brooks. “Illustrative Design Space Studies with Microarchitectural Regression Models”. In: *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA-13)*, 2007.

[Lee 07b] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. “Methods of inference and learning for performance modeling of parallel applications”. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP-12)*, 2007.

[Lloy 82] S. Lloyd. “Least squares quantization in PCM”. *IEEE Transactions on Information Theory*, Vol. 28, No. 2, 1982.

[Long 04] S. Long and M. O’Boyle. “Adaptive java optimisation using instance-based learning”. In: *Proceedings of the 18th Annual International Conference on Supercomputing (ICS-18)*, 2004.

[Luk 05] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[Mich 99] P. Michaud, A. Seznec, and S. Jourdan. “Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors”. In: *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1999.

[Mons 02] A. Monsifrot, F. Bodin, and R. Quiniou. "A Machine Learning Approach to Automatic Production of Compiler Heuristics". In: *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA-10)*, 2002.

[Moss 98] E. Moss, P. Utgoff, J. Cavazos, C. Brodley, D. Scheeff, D. Precup, and D. Stefanović. "Learning to schedule straight-line code". In: *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10 (NIPS-10)*, 1998.

[Noon 94] D. B. Noonburg and J. P. Shen. "Theoretical modeling of superscalar processor performance". In: *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, 1994.

[Nuss 01] S. Nussbaum and J. E. Smith. "Modeling Superscalar Processors via Statistical Simulation". In: *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[Oski 00] M. Oskin, F. T. Chong, and M. Farrens. "HLS: combining statistical and symbolic simulation to guide microprocessor designs". In: *Proceedings of the 27st Annual International Symposium on Computer Architecture (ISCA-27)*, 2000.

[Pan 06] Z. Pan and R. Eigenmann. "Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[Pear 01] K. Pearson. "On lines and planes of closest fit to systems of points in space". *Philosophical Magazine*, Vol. 2, No. 6, 1901.

[Phan 05] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites". In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005.

[Pusc 05] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. "SPIRAL: Code Generation for DSP Transforms". *Proceedings of the IEEE*, Vol. 93, No. 2, 2005.

[Rao 02] R. Rao, M. Oskin, and F. T. Chong. "HLSpower: Hybrid Statistical Modeling of the Superscalar Power-Performance Design Space". In: *Proceedings of the 9th International Conference on High Performance Computing (HiPC-9)*, 2002.

[Saav 96] R. H. Saavedra and A. J. Smith. "Analysis of benchmark characteristics and benchmark performance prediction". *ACM Transactions on Computer Systems*, Vol. 14, No. 4, 1996.

[Sand 07] Sandpile. "IA-32 implementation. Intel Core". http://www.sandpile.org/impl/core.htm, 2007.

[Sher 02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically Characterizing Large Scale Program Behavior". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-10)*, 2002.

[Smol 03] A. J. Smola and B. Schölkopf. "A Tutorial on Support Vector Regression". *Statistics and Computing*, Vol. 14, No. 3, 2003.

[Step 03] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. "Meta optimization: improving compiler heuristics with machine learning". *ACM SIGPLAN Notices*, Vol. 38, No. 5, 2003.

[Step 05] M. Stephenson and S. Amarasinghe. "Predicting Unroll Factors Using Supervised Classification". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.

[Tarj 06] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. "Cacti 4.0". Tech. Rep. HPL-2006-86, HP Laboratories Palo Alto, 2006.

[Tria 03] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. "Compiler optimization-space exploration". In: *Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization (CGO)*, 2003.

[Trimaran 00] "Trimaran: An infrastructure for research in instruction-level parallelism". http://www.trimaran.org/, 2000.

[Vasw 07] K. Vaswani, M. J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph. "Microarchitecture Sensitive Empirical Models for Compiler Optimizations". In: *Pro-*

ceedings of the International Symposium on Code Generation and Optimization (CGO), 2007.

[Vudu 04] R. Vuduc, J. W. Demmel, and J. A. Bilmes. "Statistical Models for Empirical Search-Based Performance Tuning". *International Journal of High Performance Computing Applications*, Vol. 18, No. 1, 2004.

[Whal 97] R. C. Whaley and J. Dongarra. "Automatically Tuned Linear Algebra Software". Tech. Rep., University of Tennessee, 1997.

[Wund 03] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling". In: *Proceedings of the 30st Annual International Symposium on Computer Architecture (ISCA-30)*, 2003.

[Yi 05] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. "Characterizing and Comparing Prevailing Simulation Techniques". In: *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[Yoto 03] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. "A comparison of empirical and model-driven optimization". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[Zhao 02] W. Zhao, B. Cai, D. Whalley, M. W. Bailey, R. van Engelen, X. Yuan, J. D. Hiser, J. W. Davidson, K. Gallivan, and D. L. Jones. "VISTA: a system for interactive code improvement". In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES/SCOPES)*, 2002.

[Zhao 03] M. Zhao, B. Childers, and M. L. Soffa. "Predicting the impact of optimizations for embedded systems". *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Vol. 38, No. 7, 2003.

[Zhao 05] M. Zhao, B. R. Childers, and M. L. Soffa. "A Model-Based Framework: An Approach for Profit-Driven Optimization". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.