Microarchitecture Sensitive Empirical Models for Compiler Optimizations

Kapil Vaswani, Matthew J. Thazhuthaveetil, Y. N. Srikant
Indian Institute of Science, Bangalore
{kapil,mjt,srikant}@csa.iisc.ernet.in

P. J. Joseph
Freescale, India
P.J.Joseph@freescale.com

Abstract

This paper proposes the use of empirical modeling techniques for building microarchitecture sensitive models for compiler optimizations. The models we build relate program performance to settings of compiler optimization flags, associated heuristics and key microarchitectural parameters. Unlike traditional analytical modeling methods, this relationship is learned entirely from data obtained by measuring performance at a small number of carefully selected compiler/microarchitecture configurations. We evaluate three different learning techniques in this context viz. linear regression, adaptive regression splines and radial basis function networks. We use the generated models to a) predict program performance at arbitrary compiler/microarchitecture configurations, b) quantify the significance of complex interactions between optimizations and the microarchitecture, and c) efficiently search for 'optimal' settings of optimization flags and heuristics for any given microarchitectural configuration.

Our evaluation using benchmarks from the SPEC CPU2000 suits suggests that accurate models ($< 5\%$ average error in prediction) can be generated using a reasonable number of simulations. We also find that using compiler settings prescribed by a model-based search can improve program performance by as much as 19% (with an average of 9.5%) over highly optimized binaries.

1. Introduction

High performance processors and optimizing compilers are arguably the two most critical building blocks of high performance computing systems. With access to large parts of program code and abundant computational resources, modern optimizing compilers employ sophisticated program analyzes to identify optimization opportunities and drive several code and data transformations that can improve program performance. On their part, processors rely on a number of aggressive microarchitectural optimizations that exploit dynamic properties of code and data and speedup programs by facilitating the execution of instructions in parallel. It is therefore not surprising that most compiler and micro-architectural optimizations *interact* with each other i.e. the effect of an optimization is determined by the presence of other optimizations and cannot be measured in isolation. Understanding the nature of these optimizations and their interactions is critical for problems such as phase ordering[2, 10] and important for the design of efficient compilers in general.

Despite several research efforts, an accurate and scalable method of characterizing compiler optimizations and their interactions remains elusive. The difficulty in developing such methods stems from the sheer complexity of modern compilers and hardware. A typical compiler consists of a large number of potentially interacting optimizations, each parametrized by a number of independent heuristics, thresholds and flags. Moreover, optimizations such as compiler-directed prefetching, procedure inlining and loop unrolling involve complex cost-benefit trade-offs that cannot be accurately captured by simple models. Modern superscalar processors add to the complexity of analysis by employing several hard-to-analyze dynamic optimizations of their own - out-of-order execution, branch prediction and multi-level caching are prime examples of such optimizations.

Although the dynamics of interactions between optimizations are not well understood, existing compilers are not entirely oblivious to the presence of these interactions. In fact, most compiler optimizations rely on carefully designed heuristics based on some abstraction or model of the underlying hardware, to maximize the benefits of the optimization and guard against *negative* interactions that may hurt performance. Although not without its advantages, the use of such hardware models and manually designed compiler heuristics has the following key drawbacks.

- **Unknown interactions.** In the absence of accurate methods for quantifying compiler and hardware interactions, the compiler writer must speculate on the magnitude and nature of interactions and identify interactions that are most likely to effect performance. Any oversight or error on the compiler writer's part may result in heuristics that focus on the wrong inter-

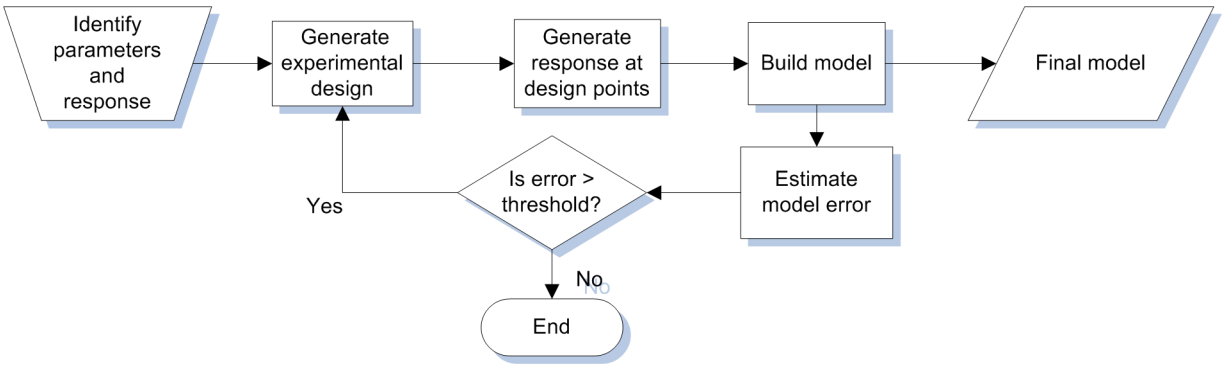

Figure 1. Overview of empirical model building process

actions, leading to sub-optimal performance. For instance, while designing heuristics that determine the unroll factor, a compiler writer may have modeled the increase in code size but ignored the positive effects of unrolling on the optimizations that follow. As a result, loops may be conservatively unrolled and the full benefits of unrolling may not materialize.

- **Imprecise hardware models.** Programs may also perform sub-optimally if the heuristics use an imprecise model of the hardware. For instance, if the compiler inserts prefetches ignoring secondary effects such as cache pollution or an increase in bus contention, the eventual performance of the optimized program may be lower than expected.

- **Microarchitecture dependence.** It is often the case that models and related heuristics are manually tuned to work well over a small range of microarchitectural configurations and may not be optimal for a different microarchitecture.

We believe that the key to solving these problems lies in the design of precise, scalable and micro-architecture sensitive models for compiler optimizations. While traditional analytical models may eventually evolve to meet these requirements, we propose an alternative approach based on a combination of *experimental design* and *empirical modeling* techniques. Unlike analytical models, empirical modeling techniques treat the underlying system as a *black box* and assume that the system response y is described by a function f of one or more *predictor variables* $\{x_i\}_1^n$ defined over a domain of interest D (Equation 1).

$$y = f(x_1, x_2, ..., x_n) + \epsilon \quad (1)$$

Here, the component ϵ reflects the dependence of the response y on quantities other than $\{x_i\}_1^n$ that are not considered for modeling. The goal of empirical modeling is to derive a function $\hat{f}$ which satisfactorily *approximates* f over the domain D. Constructing empirical models is an iterative process (Figure 1) that consists of the following basic steps:

1. Identify the predictor variables $x_1, x_2, ..., x_n$ and the domain $(x_1, x_2, ...x_n) \in D \subset R^n$ of interest.

2. Determine the mathematical form of the function $\hat{f}$.

3. Determine the response y at carefully selected *design points*. Here, each design point is a vector representing an assignment of values to the predictor variables.

4. Estimate (the unknown parameters of) the function $\hat{f}$ using data collected in Step 2. Also estimate the model error.

5. Repeat steps 3 and 4 until a model with desired accuracy is obtained.

This simple methodology offers several advantages over analytical modeling techniques. Building empirical models is an automatic process that requires minimal user intervention and does not rely on any prior knowledge about the relationship between the predictor variables and the response. Due to the iterative nature of the process, empirical models with a desired level of accuracy can be built simply by collecting more data as shown in Figure 1. Empirical models can also discover arbitrarily complex interactions between predictor variables. As a result, empirical models have a fair amount of interpretive value and can reveal interesting characteristics of the underlying system. Furthermore, these models can be used to predict system response at arbitrary points in the design space, enabling efficient exploration of the design space.

In this paper, we illustrate the use of this methodology by building application-specific empirical models for compiler optimizations. Figure 2 depicts the components and parameters of the system we model. The empirical models we build relate a program's execution time to a set of predictor variables that include compiler optimizations flags,

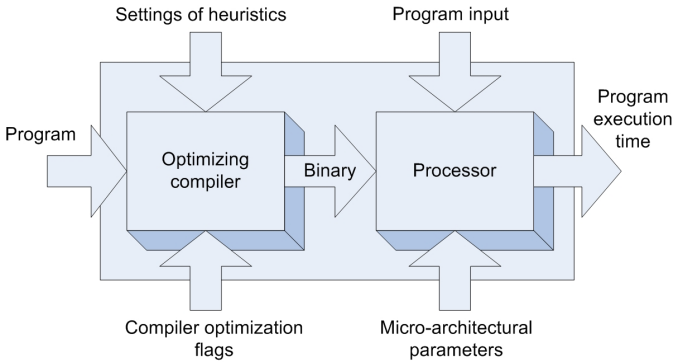

Figure 2. Components and parameters of the system we model.

numerically encoded heuristics and parameters that represent the microarchitectural configuration. The design points at which we measure system response are determined using a variant of a commonly used experimental design technique called *D-optimal designs*. We evaluate three different regression modeling techniques, *linear regression models, Multivariate Adaptive Regression Splines (MARS)* [5] and *Radial Basis Function (RBF) networks* [3] as approximations for the functional nature of this relationship.

To evaluate the modeling process, we built empirical models for 14 optimizations and related heuristics implemented in the *gcc* compiler, and 11 key microarchitectural parameters. We find that reasonably accurate models ($< 5\%$ error in prediction on average) can be automatically generated using data from a small number of design points. Using these models, we were able to identify compiler heuristics that had the largest impact on performance. Furthermore, we find that these models can be used to *search* for 'optimal' settings of the compiler optimizations flags and associated heuristics for any given micro-architectural configuration, absolving developers from the tedious task of tuning these flags and heuristics for different platforms. Our evaluation shows that using these settings improves program performance by 10% on average and upto 19% for a typical micro-architectural configuration.

The paper is organized as follows. We first discuss various issues involved in the process of identifying predictor variables and their domain in Section 2. In Section 3, we briefly describe the method we use to select design points and explain the rationale behind our choice. We describe the empirical modeling techniques we use in Section 4 and discuss their relative strengths and weaknesses. The framework we used to evaluate model building process is described in Section 5. The results of our evaluation, which involved testing the models for their predictive accuracy and assessing the suitability of the models in searching for optimal compiler settings, are presented in Section 6. We survey some of the related work in Section 7 and conclude with Section 8.

2. Preliminaries

2.1. Definitions

We first define a few terms that we will use through out the text. Most predictor variables can be classified as discrete, continuous or categorical. A categorical variable takes on discrete values with no natural order. For instance, a *binary* categorical variable takes on two values (0/1, true/false etc). Given a set of k predictor variables and their ranges, a *design point* is a k-dimensional vector that represents an assignment of values to each of the predictor variables from within their ranges. The term *design space* or *domain* refers to the set of all possible design points. A *design matrix* or a *sample* is a set of n specific design points chosen for an experiment. The *training data set* refers to the set of design points at which the system has been sampled. Empirical modeling procedures use this data set to estimate the unknown function $f(\mathbf{x})$. It is also common to use an independently generated *test data set* to assess the quality of a model.

2.2. Identifying response and predictor variables

A compiler writer/developer initiates the model building process by identifying the response and predictor variables. Since the focus of this paper is building performance models, we use the absolute execution time (measured in cycles using a cycle-accurate simulator) as the response. However, models can also be built for other metrics such as power consumption or code size. In fact, multivariate modeling techniques such as MARS (Section 4.2) allow several response variables to be modeled together.

The selection of predictor variables is more involved and depends on the eventual use of the model. If the goal is to identify significant parameters and interactions, then the designer should select all variables that can potentially influence the response. However, if the goal is to optimize the response over a constrained space, a smaller set of variables may be selected for modeling. Empirical modeling does restrict the selection to variables that can be numerically expressed or encoded and are bounded. The following non-exhaustive list enumerates different kinds of variables that a compiler writer may be interested in modeling.

Compiler optimization flags. Most compilers support command line flags that can be used to enable/disable individual optimizations. Each flag can be encoded as a binary categorical variable that takes two values,

0/1, which indicate whether the optimization is disabled/enabled.

Microarchitectural parameters. Parameters such as buffer sizes, cache sizes, number of functional units, number of registers, latencies etc. are naturally expressed as ordinary discrete variables whereas others including inorder/out-of-order issue can be represented as binary categorical variables.

Compiler heuristics. We find that several compiler optimizations rely on one or more numeric parameters that are used to determine where and how to apply the optimization. Consider the following optimizations in *gcc*:

- *Function inlining and loop unrolling:* Inlining is driven by a number of threshold based heuristics such as the maximum number of instructions in the callee and the maximum permissible increase in code size of the caller; these heuristics help protect against thrashing in the instruction cache. The inliner also uses an estimate of the relative cost of a call to exclude call sites that are unlikely to yield significant benefits. A similar set of thresholds governs loop unrolling and peeling.
- *Trace scheduling:* This optimization relies heavily on heuristics to determine how traces are formed. For instance, the optimizer decides to extend an existing trace only if the following branch has a bias greater than a threshold. The optimizer can also be tuned to limit the increase in code size due to tail duplication.

Apart from these numeric parameters, other non-numeric variables that influence the behavior of an optimization may also be considered for modeling. For instance, a set of *priority functions* [16] can be represented by a single categorical variable. A complete list of variables we chose to model is presented in Section 5.

2.3. Defining predictor variable ranges, levels and transformations

The compiler writer/developer is also required to specify the operating range of values for each non-categorical predictor variable. Since empirical models are known to be inaccurate around the edges of the chosen design space, it is advisable to select a range of values that is slightly wider than the operating ranges. Furthermore, our experiments show that varying each predictor variables at as many levels as possible tends to increase the accuracy of the resulting models. However, certain predictor variables such as cache sizes and buffer sizes that can only vary in powers of 2. Such variables can be transformed into linear predictor variables using functions such as log or $inverse$.

3. Design of Experiments

Having identified the response and the predictor variables, the next step in empirical model building is to select a set of design points in the domain of interest at which the response will be measured. Such a selection is necessitated because the total number of points in the domain is usually very large (exponential in the number of predictor variables) and measuring the response at all points is infeasible. It is important to note that the choice of design points is closely related to the accuracy and cost of building empirical models. For instance, models generated using a set of design points that are clustered in one region are unlikely to be accurate over the entire domain. *Experimental design* techniques address this problem by selecting design points based on certain criteria; the resulting designs are more amenable for analysis and likely to generate models with higher accuracy. For reasons discussed below, we use an experimental design technique known as *D-optimal design* [11] for selecting design points.

A D-optimal design of a specified size can be generated by first generating a set of *candidate* design points (either randomly or through methods such as latin hypercube sampling) and then solving the following optimization problem.

D-optimal design problem. Given a set of m *candidate* design points Z (an $m \times k$ matrix, where k is the number of predictor variables), choose a set of n design points (an $n \times k$ matrix X) from Z such that the determinant of the information matrix $det(X'X)$ is maximized.

It can be shown that maximizing the criteria $det(X'X)$ is roughly equivalent to increasing the confidence in the empirical models generated using the design. Algorithms for generating D-optimal designs are available in most statistical packages. D-optimal designs are extensible i.e. an existing D-optimal design can be easily augmented with additional design points. This is useful in scenarios where an initial design proves to be insufficient and additional data is required. Furthermore, D-optimal designs are compatible with almost all empirical modeling techniques.

4. Empirical Modeling Techniques

The goal of empirical modeling is to *learn* the relationship between the response and the predictor variables using samples from the system. We first describe three techniques that are commonly used to accomplish this task. We then address the problem of *overfitting*, which commonly occurs in over-trained empirical models.

4.1. Linear regression models

Linear regression belongs to a class of *global, parametric* regression techniques where the nature of the functional relationship between the response and the predictor variables is assumed to be known *a priori* and the specific parameters of the relationship are determined from the data. In the specific instance of linear models, we assume that the response y is *linearly* related to predictor variables $\{x_i\}_1^n$. In its simplest form, a linear regression model is represented as follows:

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \epsilon$$

Here, the coefficients $\{\beta_i\}_0^n$, also known as *partial regression coefficients*, reflect the effect or *significance* of the corresponding predictor variable on the response. Linear models can easily be extended to model interactions between predictor variables. For instance, the following model includes terms that represent two-factor interactions.

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \sum_{i=1}^{n} \sum_{j=i}^{n} \beta_{ij} x_i x_j + \epsilon \quad (2)$$

Building linear models. Linear regression models are relatively easy and quick to compute. Given a linear model with k terms and a training data set consisting of a design matrix $\mathbf{X}$ and the response vector $\mathbf{y}$, the *least squares* estimates of the partial regression coefficients $\hat{\beta} = (\beta_0, \beta_1, \ldots, \beta_{k-1})$ are computed as follows:

$$\hat{\beta} = (\mathbf{X'X})^{-1}\mathbf{X'y} \quad (3)$$

This estimate is known as the least squares estimate because it minimizes the sum of squares training error SSE.

$$SSE = \sum_{i=1}^{p} (\hat{f}(\mathbf{x}_i) - y_i)^2 \quad (4)$$

where p is the number of samples in the training data set.

Adequacy of linear models. Although global parametric models are easy to build and interpret, they are accurate only if the specified parametric model is a reasonable approximation for the underlying function. Consider the response in Figure 3, which plots the variation in execution time of the benchmark *art* (*train* input) from the SPEC CPU2000 suite for different maximum unroll factors (a heuristic in *gcc*, see Table 1 for description) and instruction cache sizes. As expected, the execution time first decreases with increasing unroll factors. However, no further improvements in execution times are observed if the unroll factor is increased beyond a threshold. Any further increase

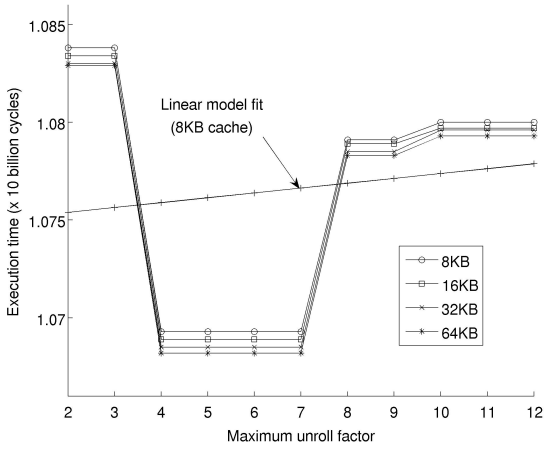

Figure 3. Variation in execution time of the *art* benchmark for different maximum unroll factor and instruction cache sizes. Also shown is a linear model approximation of performance for the 8KB icache.

in the unroll factor causes the program to slowdown, possibly due to increased register pressure. The figure also shows the execution times predicted by a simple linear model for a configuration with an 8KB instruction cache. It is clear that the simple model does not approximate the response accurately. A positive slope of the predicted response even suggests that increasing the unroll factor has an overall negative influence on performance, which is clearly not the case.

In scenarios where simple parametric models are inadequate, non-parametric procedures may be used. Unlike their parametric counterparts, non-parametric methods make no prior assumptions about the nature of the relationship between the response and the predictor variables. Instead, both the nature and the parameters of the function $\hat{f}$ are determined from experimental data. We describe two such methods in the following sections.

4.2. Multivariate Adaptive Regression Splines (MARS)

MARS [5] belongs to a class of *recursive partitioning* based regression techniques that use a divide and conquer approach to derive functions that approximate arbitrarily complex relationships. Instead of attempting to find one global parametric function that explains the response, these techniques recursively partition the domain into disjoint *regions* until the response in each region can be accurately described using simple functions of predictor variables. The final model is simply a linear combination of these *basis* functions.

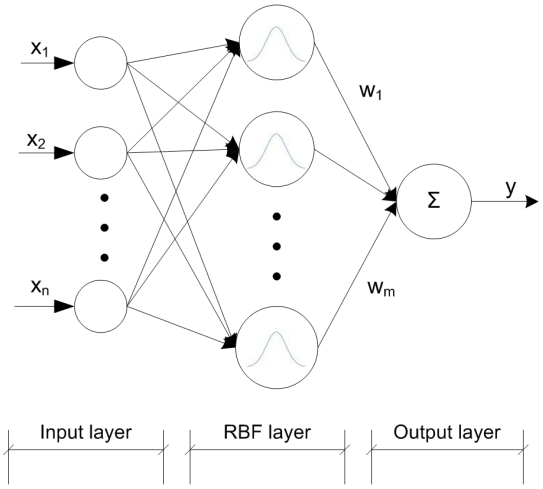

Figure 4. A typical three-layered RBF network

Formally, assume that the domain D of predictor variables is divided into M disjoint regions $\{R_m\}_1^M$. Then the response $\hat{f}(\mathbf{x})$ at a design point **x** is obtained as follows

$$\text{if } \mathbf{x} \in R_m, \text{ then } \hat{f}(\mathbf{x}) \quad = \quad w_m B_m(\mathbf{x}) \tag{5}$$

Here, the basis functions B_m are simple parametric functions that describe the response in the region R_m, and w_m are the regression coefficients. The following equation represents the resulting model.

$$\hat{f}(\mathbf{x}) \quad = \quad w_0 + \sum_{m=1}^{M} w_m B_m(\mathbf{x}) \tag{6}$$

Unlike traditional recursive partitioning procedures, MARS uses *q-order splines* [5] as basis functions. The key aspect of the MARS algorithm is that all parameters of the model, i.e. the region boundaries and the regression coefficients w_m, are determined using experimental data. Furthermore, a MARS model (Equation 6) can be re-written into a form in which the terms corresponding to predictor variables and their interactions are associated with their own coefficients (much like Equation 2). These coefficients are estimates of the influence of the variables/interactions on the response. Due to these features, models produced by MARS are not only more accurate but also interpretable.

4.3. Radial Basis Function Networks

An RBF network, shown in Figure 4, is a three-layered neural network widely used for regression and interpolation. Like MARS, RBF networks are non-parametric; they model the response as a weighted sum of basis functions:

$$\hat{f}(\mathbf{x}) \quad = \quad w_0 + \sum_{i=1}^{N} w_i h_i(\mathbf{x}) \tag{7}$$

The basis functions are transfer functions of the hidden RBF units in the network. However, unlike MARS, which uses splines as basis functions, RBF networks use localized *radial basis functions* $h_i(\mathbf{x})$ of the form

$$h_i(\mathbf{x}) \quad = \quad K(\parallel \mathbf{x} - \mathbf{x}_c^i \parallel)$$

where the function K is known as a *kernel*. Each RBF is characterized by two parameters, a *center* $\mathbf{x}_c^i$, and a *radius* r_i. The key property of RBFs is that the response of an RBF to an input $\mathbf{x}$ varies smoothly and monotonically with the distance between the input and the RBF center. The radius r_i determines the rate with which the response of the RBF increases or decreases with the distance. Equation 8 shows two of the most commonly used kernel functions, the Gaussian and the inverse multi-quadratic function.

$$\begin{aligned} [Gaussian]K_i(\mathbf{x}) &= e^{(-\|\mathbf{x}-\mathbf{x}_c^i\|^2/2r_i^2)} \\ [multiquad]K_i(\mathbf{x}) &= ((-\parallel \mathbf{x} - \mathbf{x}_c^i \parallel^2/2r_i^2) + 1)^{1/2} \end{aligned} \tag{8}$$

Training RBF networks. Given a training data set consisting of a design matrix $\mathbf{X}$ and a response vector $\mathbf{y}$, the goal of the training process is to determine the number of RBF neurons, and the center points, radii and weights of the neurons such that the resulting RBF model (Equation 7) is a good approximation of the real response function.

The choice of the number and centers of RBF neurons can be made in several ways. An obvious method is to use the design points in the training data set as RBF centers. However, as discussed in Section 4.4, the resulting RBF network, with as many neurons as the size of the training data set, is likely to *overfit*, specially for small sample sizes. Therefore, most RBF network implementations support other data centric approaches such as clustering or regression trees for choosing the number and centers of RBF neurons. For instance, the regression tree based approach uses training data to recursively partition the design space into regions with uniform response. The design point closest to the center of each such region is selected as an RBF center. Once the number of RBF neurons and their centers have been determined, the least squares approximation of the weights $\mathbf{w}$ can easily be derived using Equation 3.

4.4. Overfitting

Overfitting is a common problem that plagues many empirical modeling techniques. Overfitting occurs when a relatively small training data set is used to learn a model with high complexity (measured in terms of the number of unknown parameters). An overfit model fits the training data almost perfectly but fails to generalize and predict the response at other design points. Usually, overfitting can be avoided if a large enough sample is used for training. However, since we rely on simulations to generate training data, large sample sizes are prohibitively expensive.

#	Parameter	Description	Low Value	High Value	# levels
1	-finline-functions	Inline simple functions into their callers	0	1	2
2	-funroll-loops	Unroll loops whose number of iterations can be determined statically or at loop entry	0	1	2
3	-fschedule-insns2	Reorder instructions to eliminate execution stalls. Perform before and after register allocation	0	1	2
4	-floop-optimize	Perform simple loop optimizations such as moving constant expressions, simplify test conditions etc.	0	1	2
5	-fgcse	Perform GCSE pass, also perform constant and copy propagation	0	1	2
6	-fstrength-reduce	Perform loop strength reduction and elimination of induction variables	0	1	2
7	-fomit-frame-pointer	Do not keep the frame pointer in a register if not required	0	1	2
8	-freorder-blocks	Reorder block to reduce number of taken branches and improve code locality	0	1	2
9	-fprefetch-loop-arrays	Generate prefetch instructions in loops that access large arrays	0	1	2
10	max-inline-insns-auto	Maximum number of instructions (in GCC IR) in a single function for it to be considered for inlining	50	150	11
11	inline-unit-growth	Maximum overall growth of a compilation unit due to inlining	25	75	11
12	inline-call-cost	Cost of a call relative to a simple computation; used to identify beneficial call sites	12	20	9
13	max-unroll-times	Maximum number of times a single loop can be unrolled	4	12	9
14	max-unrolled-insns	Maximum number of instructions a loop can have for it to be considered for unrolling	100	300	21

Table 1. Compiler flags and heuristics considered for empirical modeling. Also listed are the parameter ranges and the number of levels for each parameter. All compiler parameters are linearly transformed to a scale -1 to 1 for modeling.

Overfitting can also be avoided by constraining model complexity using metrics that estimate the ability of a model to generalize. Over the years, many such metrics have been developed. One such metric is the Bayesian Information Criteria (BIC) defined by the following relation.

$$BIC = \frac{p + (ln(p) - 1)\gamma}{p(p - \gamma)} SSE \quad (9)$$

Here, SSE is the sum of squares error on the training data (Equation 4), p is the number of samples in the training data set and γ is the number of parameters in the model. In essence, the BIC is a version of the SSE that also accounts for the model's complexity. Other metrics such as the GCV (Generalized Cross Validation) are also commonly used.

5. Experimental Framework

We now describe the framework we used to test the model building procedure and evaluate the accuracy, scalability and utility of the performance models.

Parameter selection. We model a set of optimizations in the *gcc* compiler infrastructure (version 4.0.1). The choice of the compiler was governed by two factors - its widespread use, and support for a backend for which a validated simulator was available. Table 1 enumerates the set of 14 optimizations and associated heuristics we chose to model, their ranges (defined by the low and high values) and the number of levels. All optimization flags are binary encoded categorical variables, whereas the 5 heuristics are numeric variables that control inlining and unrolling.

The microarchitectural parameters we considered for modeling are listed in Table 2 along with the respective ranges and levels. The list primarily comprises of parameters related to the processor core and the memory subsystem. Previous studies [8] have shown that many of these parameters have a high influence on performance. Furthermore, the microarchitectural design space defined by these parameter ranges covers the configurations of most modern superscalar processors. Two of these parameters deserve special mention. Since the number of functional units is usually dependent on the issue width, we use the issue width parameter to determine the functional unit configuration. Also, the branch predictor size parameter refers to the size of the predictor tables in a combined branch predictor consisting of a bimodal predictor and a 2-level predictor of equal sizes. We note that this selection of compiler optimizations and microarchitectural parameters is by no means exhaustive. However, this selection is large enough to demonstrate the efficacy of our approach. Modeling this set of parameters is already beyond the capabilities of traditional analytical methods [21].

Generating experimental designs. We used the *AlgDesign* package in the R statistical tool to generate D-optimal designs for the space defined by 25 parameters. We conservatively chose a design of size 400 as our training data set and an independently generated design of size 100

#	Parameter	Low Value	High Value	# Levels
15	Issue width	2	4	2
16	Branch predictor size*	512	8192	5
17	Register update unit size*	16	128	4
18	Instruction cache size*	8KB	128KB	5
19	Data cache size*	8KB	128KB	5
20	Data cache associativity	1	2	2
21	Data cache latency	1	3	3
22	Unified l2 cache size*	256KB	8MB	6
23	Unified l2 cache associativity*	1	8	4
24	Unified l2 cache latency	6	16	11
25	Memory latency	50	150	21

Table 2. Micro-architectural parameters considered for empirical modeling. All parameters marked "*" are *log* transformed.

as our test data set.

Generating program binaries. The first step in measuring program performance at a given design point is to generate a program binary that corresponds to the settings of variables at the design point. This is achieved by compiling the program with a set of command line flags and parameters determined by the design point. However, certain microarchitectural parameters may need special treatment. For instance, the machine description used by the compiler to guide instruction scheduling must be consistent with the functional unit configuration used at the design point. In our setup, we compiled several versions of *gcc* for the *Alpha* backend, one for every possible functional unit configuration and use an appropriate version to compile the program.

Simulation methodology. We use a modified version of Simplescalar, a detailed, cycle accurate processor simulator for the Alpha architecture, to measure the response (execution time in cycles) at selected design points. Our simulator models the memory system in detail. Specifically, the simulator accurately models store buffers, buses and the DRAM.

However, measuring program performance using detailed cycle accurate simulation is expensive. The problem is further exacerbated for the following reasons: a) the number of simulations required to build models (the size of the training data set) is likely to be large, and b) each design point may correspond to a different program binary. As a result, IPC is no longer a valid performance metric and traditional simulation time reduction techniques such as Simpoint [14] cannot be used. In this paper, we address this problem by using SMARTS based methodology [19] for simulation. SMARTS relies on statistical sampling to reduce simulation time by several orders of magnitude, allowing programs to be simulated to completion. Furthermore, SMARTS provides estimates of the potential error in measurement due to sampling. These estimates can be used to tune the sampling parameters and repeat the simulation until a desired level of accuracy is obtained.

Benchmark-Input	Linear model	MARS	RBF-RT
164.gzip-graphic	4.44	3.17	2.90
175.vpr-route	7.69	3.78	1.84
177.mesa	20.15	8.78	7.31
179.art	26.44	14.20	4.63
181.mcf	11.25	4.85	3.99
255.vortex-lendian1	9.69	6.95	5.15
256.bzip2-graphic	4.81	2.80	3.02
Average	**12.07**	**6.35**	**4.13**

Table 3. Average prediction error obtained using three modeling techniques.

For the purpose of this study, we chose the sampling window size (# number of instructions sampled in one window) of 1000 and a sampling interval of 1000 (1 in every 1000 windows is sampled). Our experiments show that this choice of sampling parameters results in $< 1\%$ error (with 99.7% confidence) in estimating execution time for all programs we tested. This reduces the simulation time from several months to a few hours.

Building models. We use the iterative procedure illustrated in Figure 1 to build models for a given program-input pair. The linear models we build incorporate individual effects between parameters and two-factor interactions between them. The cost of building a large training data set prevents us from including higher order interactions. We construct the MARS models using the *polspline* package in R, which supports the use of the GCV measure to avoid overfitting. The RBF network models we build rely on regression trees [12] for choosing the number and centers of RBF neurons. We evaluated several kernel functions and found that models based on the multi-quadratic kernel to be the most accurate. The linear and RBF modeling procedures also use the BIC criteria to avoid overfitting.

6. Experimental Results

6.1. Model Diagnostics

We estimate the accuracy of the program-specific models by using the models to predict the performance at the design points in an independently generated test data set of size 100. Table 3 shows the average percentage error in prediction achieved by the three modeling techniques for 7 programs from the SPEC CPU2000 suite. Clearly, the models based on RBF networks outperform the other modeling techniques across all programs, achieving an average error of $< 5\%$ for all programs. Also note that the MARS models

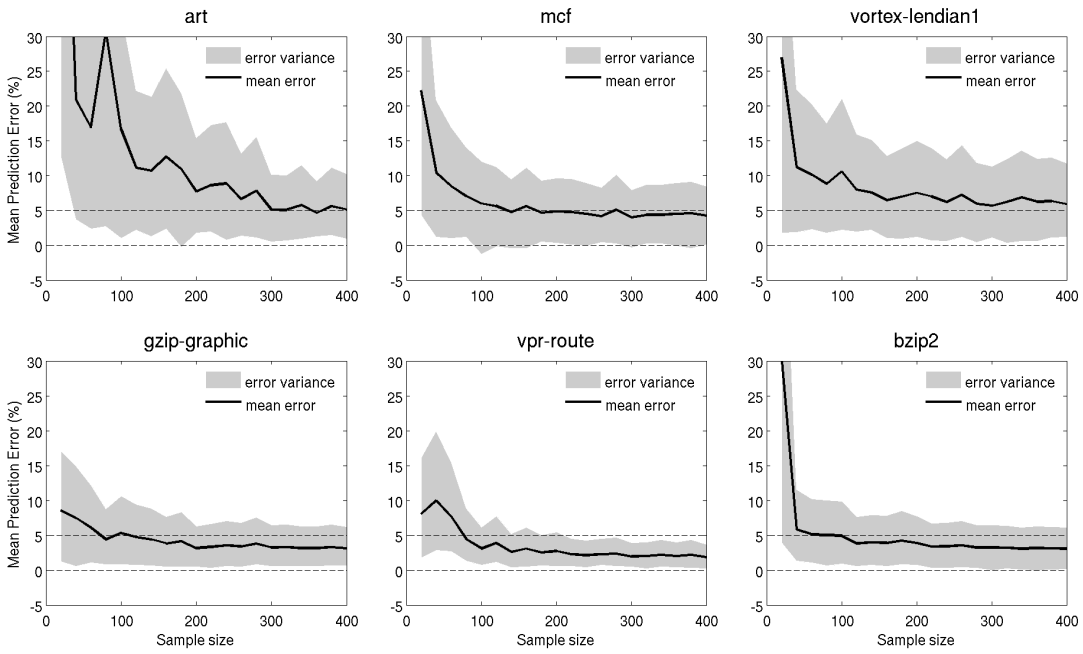

Figure 5. Effect of size of the training data set on model accuracy for benchmarks from the SPEC CPU2000 suite. The black line indicates the average error μ and the gray area represents the error variance σ.

compare well with the RBF network models for some of the programs we tested. However, linear models are plagued by high errors, which suggests that program performance may not vary linearly over large regions of the design space.

Figure 5 illustrates the effect of the training data set size on the accuracy of RBF network models. As expected, the average error in prediction tends to decrease with increasing sample sizes. Most exceptions to this rule occurs when very small sample sizes (< 100) are used. We observe that the number of samples required for the error to stabilize below the threshold of 5% varies from program to program. For a majority of the programs, this error threshold is reached between 100-200 simulation runs. Also note that the variance in error (indicated by the gray region which represents $\mu \pm \sigma$) also decreases with increasing sample sizes. However, increasing the sample size beyond this threshold yields incremental benefits.

Figure 6 throws more light on the predictive capabilities of the RBF models. Here, we plot the actual execution time vs. the predicted execution time at 100 design points in the test data set for three programs with relatively high error. We observe that all models capture high level trends in performance and no outliers are observed.

6.2. Interpreting models

Although RBF models prove to be more accurate than the MARS models, they suffer from the lack of interpretability. Therefore, MARS models may in fact be preferred in cases where they are sufficiently accurate. As aforementioned, MARS models can be represented in a form in which each predictor variable/interaction is associated with a single coefficient, which represents its significance over the entire design space. Table 4 shows the key parameters/interactions and their coefficients as inferred from the simplified form of the MARS models for program-input pairs we tested. In these models, the co-efficient value of a predictor variable/interaction is equal to one-half the change in execution time (measured in billions on cycles) caused by changing the variable(s) from their low to high value. This listing reveals several interesting insights. First, we note that the micro-architectural parameters and interactions dominate program performance, whereas compiler

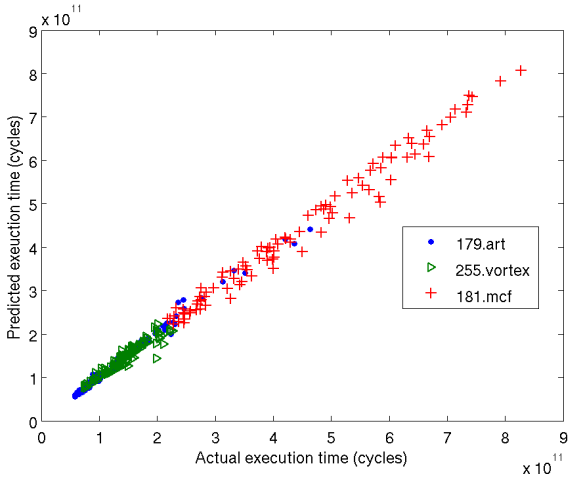

Figure 6. The actual vs. predicted execution times for three programs, *art*, *vortex* and *mcf* using RBF network based models.

optimizations have a smaller role to play. Surprisingly, the *omit-frame-pointer* and *inlining* appear to be the most significant compiler optimizations in their own right; these optimizations improve performance significantly for virtually all programs. We also find that loop optimizations (represented by the -floop-optimize flag) can hurt performance. Also, the significant positive interaction between inlining and the RUU size in *mcf* suggests that inlining is less beneficial for a microarchitecture with large reorder buffer. On the other hand, both unrolling (specifically the heuristic max-unroll-factor) and omit-frame-pointer are more beneficial when the reorder buffer is large. Also of interest are the interactions between inlining and omitting the frame pointer and between block reordering and L2 cache latency. Finally, we note that no two programs respond to compiler optimizations in similar ways i.e. the set of optimizations that improve performance varies from one program to the other. MARS models can be used to obtain a comprehensive listing of all such effects.

6.3. Model-based search for platform-specific optimization settings

Apart from their usefulness in breaking down and analyzing performance, empirical models can serve many other purposes. For instance, empirical models can predict the response at arbitrary design points at virtually no computation cost. Such a model enables efficient and systematic design space exploration, a task traditionally associated with high costs. We evaluated this feature by using our models to search for near optimal compiler and heuristic settings for a given microarchitectural configuration. If a search of this nature can be performed efficiently, it is conceivable that an empirical model (developed offline for all platforms) can be packaged with a program's compilation system. When the program is installed on a specific platform, the empirical model could be parametrized with the platform's configuration and used to search for the optimal optimization flags and heuristic settings, which would then be used to compile the program.

Parameter	Constrained	Typical	Aggressive
Issue width	2	4	4
Branch predictor size	512	2048	8192
Register update unit size	16	64	128
Instruction cache size	8KB	32KB	128KB
Data cache size	8KB	32KB	128KB
Data cache associativity	1	1	2
Data cache latency	1	2	3
Unified l2 cache size	256KB	1MB	8MB
Unified l2 cache associativity	2	4	8
Unified l2 cache latency	6	10	16
Memory latency	50	100	150

Table 5. Micro-architectural configurations for which models were used to search for optimal compiler and heuristic settings.

For our evaluation, we selected three different microarchitectural configurations that represent a large spectrum of the microarchitectural design space. These configurations, listed in Table 5, are referred to as *constrained*, *typical* and *aggressive*. The *constrained* and the *aggressive* configurations are specifically chosen to test the accuracy of the models at the edges of the design space.

We used a genetic algorithm to search for the optimal platform-specific settings of compiler flags and heuristics. Given an empirical model, we freeze the microarchitectural parameters in the model and then allow the GA to explore the rest of the design space. The GA starts with an initial, randomly generated population of optimization flags and heuristic settings. It uses the empirical model to predict performance at all design points in the population and uses this measure to eliminate 'unfit' design points and retain the fittest ones. The GA then uses the usual *crossover* and *mutation* operators to create a new generation of candidate points and repeats the process. The GA terminates either when the optimal design point is reached or the number of generations exceeds a user specified threshold, in which case the best available design point is reported.

Table 6 lists the best settings of optimizations flags and heuristics we obtained using the genetic algorithm for the three micro-architectural configurations. We observe that the optimal settings are highly program and micro-architecture dependent. These settings are also significantly different from the default O3 settings.

Figure 7 shows the speedups (over -O2) obtained by us-

Parameter/interaction	181.mcf	256.bzip2	175.vpr	255.vortex	164.gzip	179.art	177.mesa
constant (β_0)	439.30	132.84	177.28	142.85	75.52	123.40	226.72
issue width	0.00	-18.77	-7.27	-17.79	-11.73	0.00	-39.06
RUU Size	0.00	-11.05	-39.56	-16.69	-4.25	-48.03	-25.56
ul2 latency	5.54	1.91	2.90	15.32	3.40	0.00	29.33
dl1 size	0.00	-1.94	-4.09	-4.61	-2.55	0.00	-6.69
dl1 latency	0.00	2.77	2.63	3.44	1.98	0.00	0.00
dl1 assoc	0.00	0.48	-1.43	-3.07	-1.18	0.00	0.00
il1 size	0.00	-1.82	0.00	-24.17	-1.75	0.00	-50.22
ul2 size	-155.91	-12.92	-12.93	-9.74	0.00	-88.99	-5.19
ul2 assoc	8.51	0.87	-2.67	-4.23	0.00	0.00	0.00
memory latency	81.46	7.66	10.95	0.00	0.99	37.95	71.78
issue width * RUU size	0.00	-1.97	-1.93	0.00	-2.42	0.00	0.00
RUU Size * dl1 latency	0.00	-1.82	-1.59	-5.10	-1.83	0.00	0.00
RUU Size * ul2 size	0.00	3.08	7.29	0.00	0.00	42.42	0.00
RUU Size * memory latency	0.00	0.00	-4.23	0.00	0.00	-20.02	0.00
il1 size * ul2 latency	0.00	0.00	0.00	-14.74	-1.31	0.00	-33.38
ul2 size * memory latency	-79.05	-6.55	-6.44	4.25	0.00	-42.13	0.00
ul2 size * ul2 assoc	-8.60	1.25	1.83	7.25	0.00	0.00	0.00
ul2 assoc * memory latency	8.18	0.00	-1.77	0.00	0.00	0.00	0.00
inlining	-16.62	-0.90	-3.28	-2.17	-0.53	0.00	-5.81
unrolling	0.00	-0.86	0.00	0.00	-0.93	0.00	0.00
loop optimize	0.00	0.00	0.00	0.00	0.56	0.00	4.47
reorder blocks	0.00	-0.73	0.00	0.00	0.00	0.00	-3.84
gcse	0.00	0.00	0.00	-1.92	0.00	0.00	0.00
omit frame pointer	-7.82	-2.01	-2.23	-4.41	-1.27	0.00	-5.60
max-inline-insns	0.00	0.00	-2.13	0.00	1.18	0.00	0.00
max-unroll-factor	0.00	0.00	0.00	0.00	0.00	0.00	5.04
inlining * RUU Size	10.33	0.00	0.00	0.00	0.00	0.00	0.00
inlining * omit frame pointer	6.29	0.00	0.00	0.00	0.00	0.00	0.00
inlining * ul2 latency	2.17	0.00	0.00	0.00	0.00	0.00	0.00
omit frame pointer * issue width	0.00	0.76	0.00	0.00	0.69	0.00	0.00
scheduling * il1 size	0.00	-0.71	0.00	0.00	0.00	0.00	0.00
omit frame pointer * RUU Size	0.00	0.00	1.48	0.00	0.00	0.00	-8.20
reorder blocks * ul2 latency	0.00	0.00	0.00	0.00	0.00	0.00	-6.32
max-unroll-factor * RUU Size	0.00	0.00	0.00	0.00	0.00	0.00	-8.33

Table 4. Coefficients of key parameters and interactions inferred from the MARS models. The coefficients represent execution time in billion of cycles.

Program-Input	1	2	3	4	5	6	7	8	9	10	11	12	13	14
gzip-graphic	1/1/1	1/1/1	0/0/0	0/0/0	0/1/0	0/0/1	1/1/1	1/1/1	1/0/0	134/128/126	25/43/36	15/16/16	10/09/09	200/200/200
vpr-route	1/1/1	1/0/0	0/0/0	1/0/1	0/0/0	0/0/0	1/1/1	1/1/1	1/1/1	138/141/137	48/47/47	12/12/19	11/04/12	300/300/300
mesa-ref	1/1/1	0/0/1	1/0/0	0/0/0	1/0/1	1/1/0	1/1/1	1/1/1	0/0/1	108/108/105	75/75/55	16/16/16	08/08/08	151/152/166
art-ref	1/0/0	1/1/1	1/1/0	1/1/1	1/1/1	1/1/1	0/0/0	0/0/1	1/1/1	100/050/050	55/75/75	16/12/20	08/12/04	206/300/300
mcf-ref	1/1/1	1/1/1	1/1/1	0/0/1	0/1/1	1/1/1	1/1/1	0/0/1	0/0/1	080/050/050	50/50/75	16/16/12	08/08/12	200/200/100
vortex-lendian1	1/1/1	1/0/0	0/0/0	0/0/0	1/1/1	1/1/1	1/1/1	1/1/0	1/0/0	100/100/100	25/25/25	18/18/18	10/10/09	200/200/200
bzip2-graphic	1/1/1	1/1/1	0/0/0	0/0/1	0/0/0	0/0/1	1/1/1	1/1/1	0/0/0	100/100/100	58/62/60	16/16/16	10/12/09	300/299/216
default O3	**1/1/1**	**0/0/0**	**1/1/1**	**1/1/1**	**1/1/1**	**1/1/1**	**1/1/1**	**1/1/1**	**1/1/1**	**100/100/100**	**50/50/50**	**16/16/16**	**08/08/08**	**200/200/200**

Table 6. Optimization flag and heuristic settings prescribed by model-based search using RBF models for the *conservative/typical/aggressive* micro-architectural configurations.

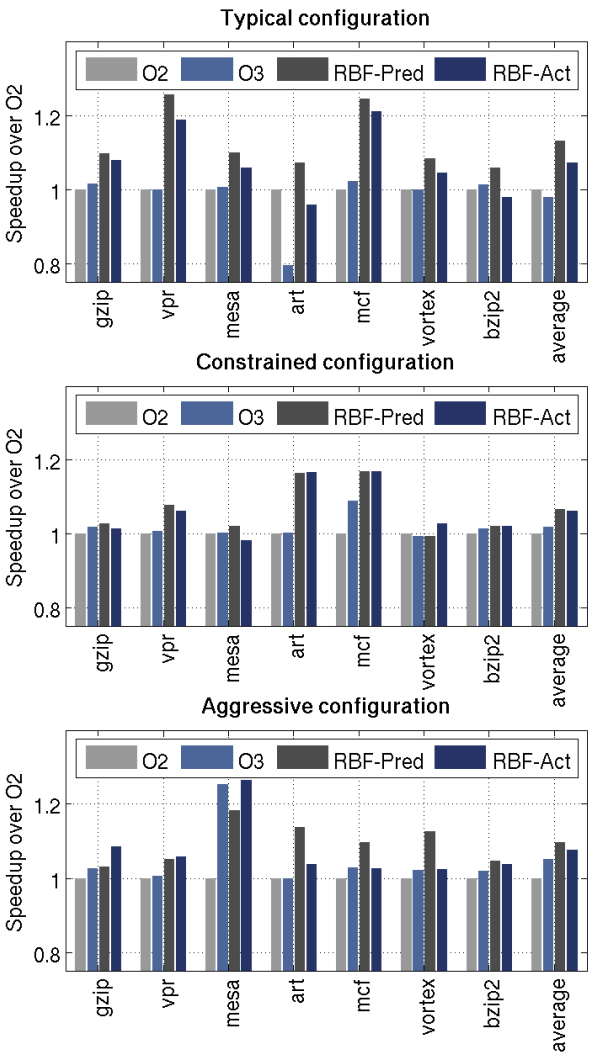

Figure 7. Predicted and actual speedup in execution time for flag and heuristic settings determined using RBF models for three different microarchitectural configurations.

Program	Train input	Actual speedup over O2 (%) Constrained	Typical	Aggressive
164.gzip	graphic	2.22	6.24	3.12
175.vpr	route	8.17	5.23	4.19
177.mesa	train	-1.89	-4.76	26.54
179.art	train	16.78	18.07	-0.01
181.mcf	train	17.37	21.40	2.43
255.vortex	lendian	-1.38	-13.45	-8.32
256.bzip2	graphic	-0.20	-2.78	1.88
Average		**5.87**	**4.28**	**4.26**

Table 7. Speedups obtained using model based search in a typical profile-guided optimization scenario for three different microarchitectural configurations.

ing the optimal settings prescribed by model-based search. First, we observe that the speedup in performance due to -O3 optimizations is fairly small across programs and microarchitectural configurations. In fact, O3 optimizations result in an average slowdown of 2% on the *typical* configuration. On the other hand, at the optimization settings prescribed by the model-based search, RBF models predict significant improvements over -O2 (an average of 14% for the *typical* configuration). Our experiments also indicate that the actual speedup at these design points is close to the predicted speedup (9.5% on average with a maximum of 19% over O3). This observation also holds for the *constrained* configuration, although the actual speedups are comparatively lower. However, we find that the model prediction errors are higher for the *aggressive* configuration. Here, the actual speedups at design points prescribed by the GA differ from the predicted speedups. This is to be expected since this design point lies at the edge of the design space, where empirical models are known to have low predictive accuracy [7]. For the other configurations, we attribute the speedups to the judicious, program-specific selection of optimization flags and heuristic values that takes into account both positive and negative interactions between optimizations and the hardware.

We also tested the model-based search in a typical profile-guided optimization scenario, where the model is built for a 'representative' input and used to identify optimization flag and heuristic settings for future runs of the program. Table 7 shows the resulting speedups (over -O2) for the three micro-architectural configurations. We find that while performance for most programs improves at settings prescribed using the train inputs (*art* and *mcf* being prime examples), a few programs are adversely affected. We speculate that this effect may be caused by the difference between the reference and the training inputs; a more detailed analysis is left for future work.

7. Related Work

The influence of interactions between optimizations and hardware on performance has long been known by compiler writers and researchers. Several researchers have proposed techniques that make compilers *interaction* aware; these techniques analytically model the costs and benefits of individual optimizations and potential interactions [18, 13, 9, 20]. The most generic of these proposals is the profit-driven framework developed by Zhao et al [21], which proposes mechanisms for modeling the influence of optimizations on

code and machine resources. These methods typically analyze optimization sites in isolation and make several simplifying assumptions such as the additive nature of costs and benefits and the lack of interactions between optimizations and hardware components such as the reorder buffer or branch predictors.

Several researchers have recognized the inherent complexity of understanding and modeling interactions between optimizations and opted for the use of statistical and machine learning techniques to address these problems instead. Haneda et al [6] propose the use of statistical inference techniques for selection of optimal optimization flags on a given platform. Search and learning techniques have been used to to identify efficient compilation sequences [17, 2, 10, 1]. and to improve optimization specific heuristics [15]. Many of these techniques are based on the key idea that the effect of compiler optimizations on a program is related to certain key program features. Cazavos et al [4] propose the use of neural networks and reaction-based modeling techniques to build empirical models for compiler optimizations. Their modeling technique is based on the assumption that a program can be characterized by the way it responds to a set of *characteristic* optimizations. In contrast to these approaches, our models are platform independent but specific to each program-input pair. Our models are designed to generalize across a complex and important but often ignored micro-architectural design space.

8. Conclusions

In this paper, we proposed the use of empirical modeling techniques to build application-specific performance models for compiler optimizations. The key feature of our modeling process is the ability to simultaneously capture the effects and interactions between several compiler optimizations, associated heuristics and microarchitectural parameters without any prior knowledge about their behavior. We show that empirical models can be effective tools in the hands of compiler writers. The models assist a compiler writer in identifying key effects and interactions that have a high impact on performance, and thus provide information useful for designing better analytical models and optimization heuristics. The models also enable efficient searches over parts of the design space, as illustrated by our use of the models for searching optimal compiler flags and heuristics for any arbitrary hardware platform.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *Proc. of CGO*, 2006.

[2] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramaniam, L. Torczon, and T. Waterman. Finding Effective Compilation Sequences. In *Proc. of LCTES*, 2004.

[3] D. S. Broomhead and D. Love. Multivariate Function Interpolation and Adaptive Networks. *Complex Systems*, 1988.

[4] J. Cavazos, C. Dubach, F. Agakov, E. 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 *Proc. of CASES*, 2006.

[5] J. H. Friedman. Multivariate Adaptive Regression Splines. *The Annals of Statistics*, pages 1–141, 1991.

[6] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic Selection of Compiler Options using Non-parametric Inferential Statistics. In *Proc. of PACT*, 2005.

[7] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A Predictive Model for Superscalar Processor Performance. In *Proc. of MICRO*, 2006.

[8] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. Construction and Use of Linear Regression Models for Processor Performance Analysis. In *Proc. of HPCA*, 2006.

[9] T. Kisuki, P. M. W. KnijnenBurg, and M. F. P. O'Boyle. Combined Selection of Tile Size and Unroll Factors using Iterative Compilation. In *Proc. of PACT*, 2000.

[10] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast Searches for Effective Optimization Phase Sequences. In *Proc. of PLDI*, 2004.

[11] D. C. Montgomery. *Design and Analysis of Experiments*. Wiley, 5th edition, 2001.

[12] M. Orr, J. Hallam, K. Takezawa, A. Murray, S. Ninomiya, M. Oide, and T. Leonard. Combining Regression Trees and Radial Basis Function Networks. *International Journal of Neural Systems*, 2000.

[13] V. Sarkar and N. Megiddo. An Analytic Model for Loop Tiling and its Solution. In *Proc. of ISPASS*, 2000.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. of ASPLOS*, 2002.

[15] M. Stephenson and S. Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *Proc. of CGO*, 2005.

[16] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O-Reily. Meta Optimization: Improving Compiler Heuristics using Machine Learning. In *Proc. of PLDI*, 2003.

[17] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler Optimization Space Exploration. In *Proc. of CGO*, 2003.

[18] M. E. Wolf, D. E. Maydan, and D. Chen. Combining Loop Transformations Considering Caches and Scheduling. In *Proc. of MICRO*, 1996.

[19] R. E. Wunderlich, B. F. T. F. Wenisch, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proc. of ISCA*, 2003.

[20] M. Zhao, B. R. Childers, and M. L. Soffa. Predicting the Impact of Optimizations for Embedded Systems. In *Proc. of LCTES*, 2003.

[21] M. Zhao, B. R. Childers, and M. L. Soffa. A Model-based Framework: an Approach to Profit-driven Optimization. In *Proc. of CGO*, 2005.