An Empirical Architecture-Centric Approach to Microarchitectural Design Space Exploration

Christophe Dubach, Timothy M. Jones, and Michael F.P. O'Boyle

Abstract—The microarchitectural design space of a new processor is too large for an architect to evaluate in its entirety. Even with the use of statistical simulation, evaluation of a single configuration can take an excessive amount of time due to the need to run a set of benchmarks with realistic workloads. This paper proposes a novel machine-learning model that can quickly and accurately predict the performance and energy consumption of any new program on any microarchitectural configuration. This *architecture-centric* approach uses prior knowledge from offline training and applies it across benchmarks. This allows our model to predict the performance of any new program across the entire microarchitecture configuration space with just 32 further simulations. First, we analyze our design space and show how different microarchitectural parameters can affect the cycles, energy, energy-delay (ED), and energy-delay-squared (EDD) of the architectural configurations. We show the accuracy of our predictor on SPEC CPU 2000 and how it can be used to predict programs from a different benchmark suite. We then compare our approach to a state-of-the-art *program-specific* predictor and show that we significantly reduce prediction error. We reduce the average error when predicting performance from 24 percent to just seven percent and increase the correlation coefficient from 0.55 to 0.95. Finally, we evaluate the cost of offline learning and show that we can still achieve a high coefficient of correlation when using just five benchmarks to train.

Index Terms—Microprocessors and microcomputers, performance analysis and design aids, modeling techniques, machine learning.

1 Introduction

Architects use cycle-accurate simulators to accurately explore the design space of new processors. These simulators allow the designer to evaluate the performance and power consumption of a variety of applications as the parameters of the processor are varied. This establishes the trends within the space and enables the identification of sweet spots where performance and power are optimally balanced. However, the number of different variables and the range of values they can take make the design space too large to be completely evaluated. Furthermore, the desire to simulate many benchmarks using realistic workloads that require long simulation times means that designers are constrained to only study small subsets of the space.

Recently, several techniques based on sampling have been developed to reduce the time taken for simulation, such as SimPoint [1] and SMARTS [2]. Although these schemes increase the number of simulations possible within a given time frame, given the huge size of the design space to be explored, a full evaluation remains unrealistic.

One technique that can help reduce simulation time is through the use of analytic models [3], [4], [5], [6]. These describe a particular microarchitectural component in detail and can be combined to provide a model of the entire processor. However, they are costly to construct and, as the complexity of each system component increases, become progressively more difficult to build.

Several studies have proposed the use of machine learning to help evaluate these massive spaces [7], [8], [9], [10], [11]. These techniques are attractive because they provide a scalable approach to design space exploration whereby a modest reduction in detail is traded for substantial gains in speed and tractability. These schemes require a number of simulations of a benchmark to be run, the results from which are used to train a predictor. This can then be used to determine the rest of the design space without the need for further simulation. However, existing techniques suffer from two major drawbacks:

- Whenever a new program is considered, a new predictor must be trained and built, meaning there is a large overhead even if the designer just wants to compile with a different optimization level [12]. Our approach learns across programs and captures the behavior of the architecture rather than the program itself.
- A large number of training simulations are needed to use these existing predictors, offsetting the benefits of the schemes. In our approach, having previously trained offline on a small number of programs, we only need a few simulations, called *responses*, in order to characterize each new program we want to predict. We show that, in fact, this can be as low as just 32 simulations to enable us to predict for any new program.

This paper presents a new and different approach to design space exploration using machine learning. We use existing knowledge to predict a new program on any given architecture configuration, something no other research has successfully attempted. Using leave-one-out cross validation, we train our *architecture-centric* model offline on all but one of our benchmark programs. Next, using a completely new program never seen before, we run just 32 simulations of the new program. We can then predict the rest of the design space of 18 billion configurations. This means that encountering a new program, or simply exploring the

- *The authors are with the School of Informatics, University of Edinburgh. E-mail: christophe.dubach@ed.ac.uk, {tjones1, mob}@inf.ed.ac.uk.*

Manuscript received 15 July 2009; revised 11 Dec. 2009; accepted 22 June 2010; published online 15 Dec. 2010.
Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-07-0326.
Digital Object Identifier no. 10.1109/TC.2010.280.

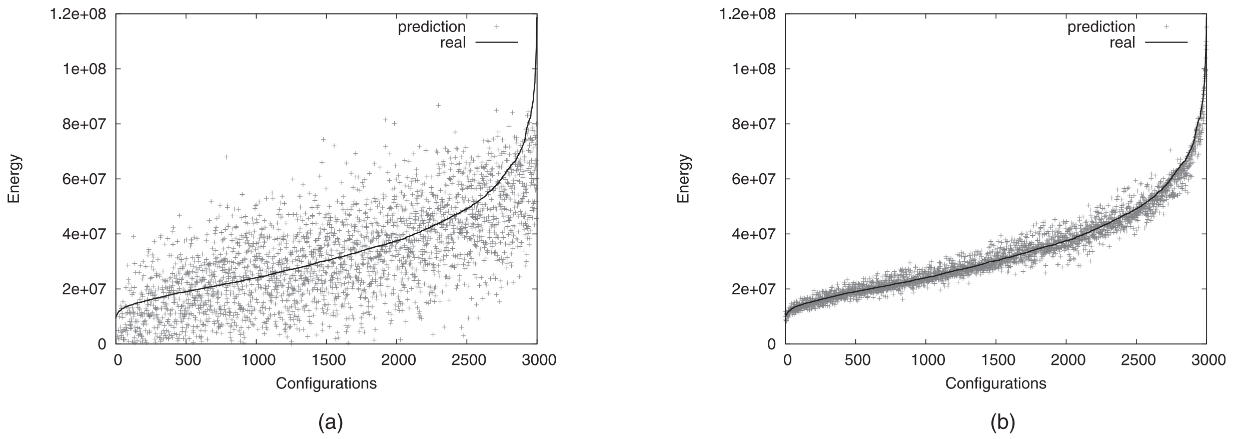


Fig. 1. The design space of *applu* when considering energy. We show the predictions given by a program-specific predictor and by our architecture-centric approach. Both models are given the same 32 simulations from this program, with the architecture-centric predictor having also been trained offline on different benchmarks. (a) Program-specific predictor. (b) Architecture-centric predictor.

compiler optimization space, can be done efficiently, at the same time as microarchitectural design space exploration, with low overhead. This is an order of magnitude less than the current state-of-the-art, *program-specific* approaches [7], [8], [9], [10], [11] and shows the ability to learn across applications, using prior knowledge to predict new programs.

Although absolute error is an important metric to evaluate the accuracy of our predictor, in this setting of design space exploration, correlation is equally important. This shows how the model can follow the trends in the space. Hence, we show the error and correlation for our architecture-centric predictor and prove that it is better than other existing approaches. We use our predictor to predict the MiBench benchmarks when trained on SPEC CPU 2000, proving that it works equally well across benchmark suites. We then compare our model against an existing program-specific predictor [7], showing that it achieves same accuracy using far fewer simulations.

One reasonable criticism of our work could be that the cost of offline training is too high. We address this by considering the use of just a few randomly selected training programs showing that our predictor achieves accurate results. We further demonstrate that even with limited offline training, our approach still outperforms existing techniques.

The rest of this paper is structured as follows: Section 2 provides a simple example showing the accuracy of our predictor. We describe our design space in Section 3 and then analyze the program characteristics in Section 4. Section 5 presents our model and Section 6 evaluates its optimal parameters. Then, Section 7 evaluates our model on SPEC CPU 2000 and MiBench and compares it against the state of the art. Section 8 addresses the cost of offline training. Finally, we describe work related to ours, especially the program-specific predictor that we compare against throughout this work, in Section 9 and conclude this paper in Section 10.

2 Motivation

This section provides a simple motivating example, illustrating the superior accuracy of our scheme in predicting architecture performance.

Fig. 1 shows the energy design space for a typical application, *applu*. We show the resulting prediction for a program-specific predictor and our scheme. The program-specific model has been trained with 32 simulations from this benchmark, whereas our architecture-centric model has been trained offline with other benchmarks and given the same 32 simulations as *responses* for this new program.

In each graph, the different microarchitectural configurations are plotted along the x-axis in order of increasing energy which is plotted along the y-axis. We show the real value of energy as a line in each graph and each model then provides a prediction for that configuration which is plotted as a point. The closer the point is vertically to the line, the more accurate the prediction is.

Fig. 1a shows how the existing program-specific technique performs when predicting this space and Fig. 1b shows how our architecture-centric scheme performs. For the same number of simulations from this program, the program-specific predictor has a high error rate and cannot determine the trend within the design space. Our architecture-centric model, however, applies prior knowledge from previously seen benchmarks and has a low error rate, accurately following the shape of the space. From these graphs, it is clear that our model provides more accurate predictions than the existing scheme.

3 Experimental Setup

This section describes the design space of microarchitectural configurations that we explore in this paper. We also present our simulation environment and benchmarks as well as the impact of the parameters on the target metrics.

3.1 Microarchitecture Design Space

This paper proposes a scheme that accurately models the design space of new programs. We chose to vary 13 different parameters in a superscalar simulator to give an overall total of 63 billion different configurations of the processor core. With a design space as large as this, it would be impossible to simulate every configuration, motivating the need for fast and accurate models.

The parameters we varied, shown in Table 1, are similar to those other researchers have looked at [7], [10] which allows meaningful comparisons with previous work. The first column describes the parameter and the second column gives the range of values the parameter can take along with

TABLE 1
Microarchitectural Design Parameters that Were Varied with Their Range, Steps and the Number of Different Values They Can Take, Also Included is the Baseline Configuration

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

TABLE 2
Microarchitectural Design Parameters that Were Not Explicitly Varied, either Remaining Constant or Varying According to the Width of the Machine (a) Constant, (b) Related to Width

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)

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)

the step size between the minimum and maximum. The third column gives the number of different values that this range gives. For example, the reorder buffer (ROB) has a minimum size of 32 entries and a maximum size of 160 entries varied in steps of 8, meaning 17 different design points.

Table 2a describes the processor parameters that remained constant in all of our simulations. Table 2b describes the functional units which varied according to the width of the processor. So, for a four-way machine, we used four integer ALUs, two integer multipliers, two floating point ALUs, and one floating point multiplier/divider.

Within our design space of 63 billion points, we filtered out configurations that did not make architectural sense. So, for example, we did not consider configurations where the reorder buffer was smaller than the issue queue. This reduced the total design space to 18 billion points.

3.2 Benchmarks and Simulator

For our experiments, we used the entire SPEC CPU 2000 benchmark suite [13] compiled with the highest optimization level and run with the *reference* input set. To accurately represent each program, we used SimPoint [1] with an interval size of 10 million instructions and a maximum of 30 clusters per program. Each interval was run after warming the cache and branch predictor for 10 million instructions.

In addition to SPEC CPU 2000, we also used the MiBench benchmark suite [14] in the latter sections of this paper. All of these benchmarks were compiled with the highest optimization level and run to completion using the *small* input set. We have omitted *ghostscript* because this program would not compile correctly in our environment.

Our simulator is based on Wattch [15] (an extension to SimpleScalar [16]) and Cacti [17] and contains detailed models of energy for each structure within the processor. We accurately modeled the latencies of microarchitecture components using the Cacti timing information to make our simulations as realistic as possible.

In the following sections, we use cycles as a metric for program performance and energy consumption (in nJ) as gained from Cacti and Wattch. We also show the energy-delay (ED) and energy-delay-squared (EDD) products to determine the trade-offs between performance and energy consumption, or efficiency. These are important metrics in microarchitecture design because they indicate how efficient the processor is at converting energy into speed of operation, the lower the value the better [18]. The ED product implies that there is an equal trade-off between energy consumption and delay, whereas the EDD product emphasizes performance over energy.

3.3 Sampling the Design Space

There are 18 billion design points in our space which are obviously too many to simulate in total. Therefore, we used uniform random sampling to pick 3,000 architectural configurations and simulated these for each benchmark. Using uniform random sampling means that we have a fair and representative sample of the total design space. While it is difficult to estimate the optimal number of samples that would be necessary to represent a distribution in general, we noticed that running more than 2,000 simulations did not significantly change the distribution of the sampled design points. Therefore, we conclude that 3,000 samples offer some evidence that our technique would work throughout the space. These 3,000 sampled architectures are later used to conduct our analysis of the design space and evaluate our technique.

3.4 Impact of Parameters

This section describes the impact of varying the microarchitectural parameters on the performance of the SPEC CPU 2000 benchmark suite. We want to explore whether particular parameter values lead to exceptionally good or bad performace. We chose the top and bottom one percent of the space as the design points that meet these criteria. If a particular value occurs often, then it is likely that it strongly contributes to the configuration achieving high or low performance.

Fig. 2 shows some of the parameters that we vary and how they influence the number of cycles required for all programs within SPEC CPU 2000. The x-axis shows the parameter values and the y-axis represents the frequency that each point occurs in the top or bottom one percent of

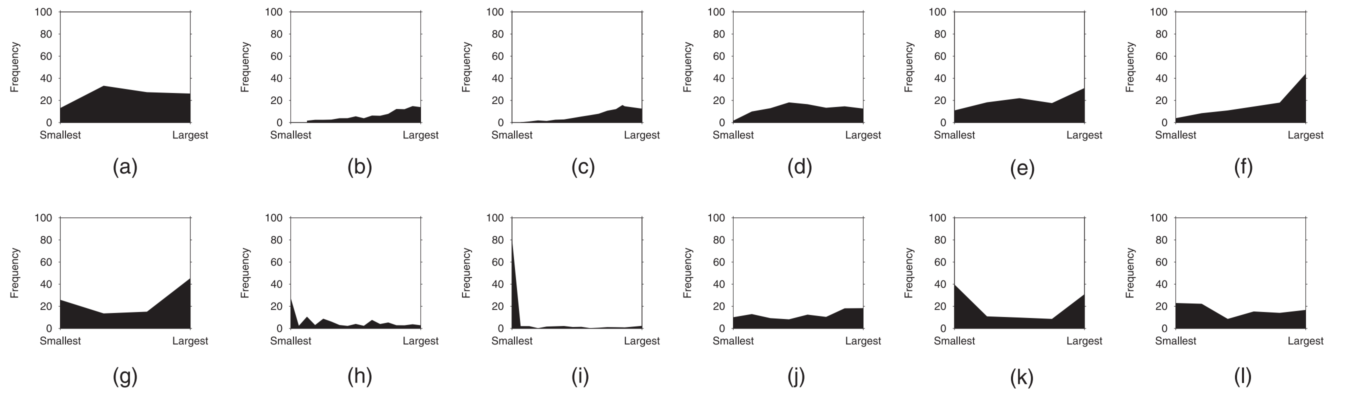


Fig. 2. The frequency each parameter design point occurs in the one percent of configurations for each benchmark having the best (a-f) and worst (g-l) number of cycles. (a) Width, (b) ROB, (c) RF, (d) RF read, (e) L2 cache, (f) Bpred, (g) Width, (h) ROB, (i) RF, (j) RF read, (k) L2 cache, and (l) Bpred.

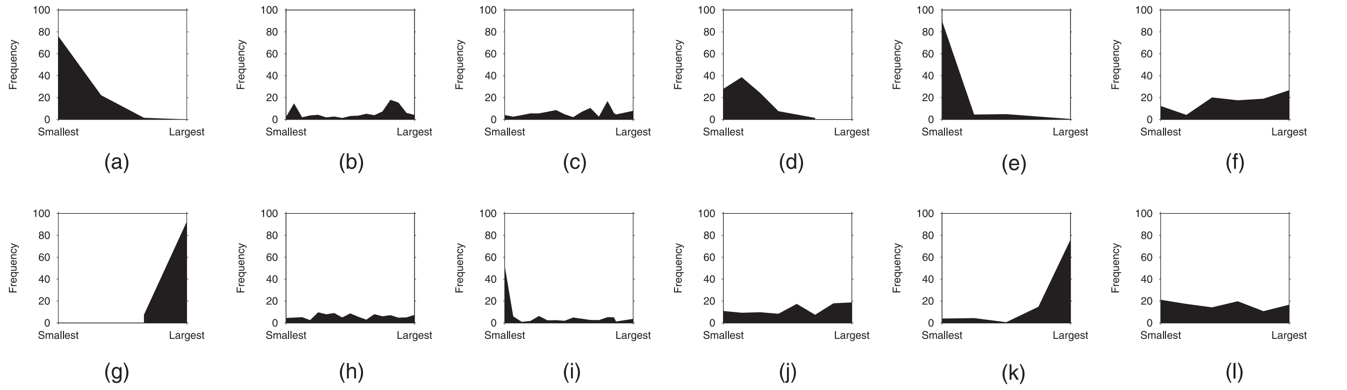


Fig. 3. The frequency each parameter design point occurs in the one percent of configurations for each benchmark having the best (a-f) and worst (g-l) energy. (a) Width, (b) ROB, (c) RF, (d) RF read, (e) L2 cache, (f) Bpred, (g) Width, (h) ROB, (i) RF, (j) RF read, (k) L2 cache, and (l) Bpred.

the space. Figs. 2a, 2b, 2c, 2d, 2e, and 2f show the top one percent for cycles (so best performance) and Figs. 2g, 2h, 2i, 2j, 2k, and 2l show the worst one percent.

From these diagrams, we can see that the parameter having the greatest impact on performance is the size of the register file (Figs. 2c and 2i). This parameter is highly correlated with overall performance. In the worst performing one percent, a small register file is common (in 81 percent of them, it has just 40 registers). This confirms the widely known fact that register files are critical components in the microarchitecture. However, the best performing one percent of configurations have register files ranging from midsized to large, suggesting that a small register file is a bottleneck to performance, but a large register file does not necessarily mean high performance. The best configurations for cycles tend to have a wide pipeline (six or eight instructions per cycle, Fig. 2b), have a large reorder buffer (Fig. 2b), branch predictor (Fig. 2f), and second level cache (Fig. 2e). This makes sense because the first two allow the extraction of ILP through branch speculation.

In terms of energy (Fig. 3), the characterization of the space is more clear cut with large differences between the minimum and maximum of the space. The configurations with the highest energy consumption have a wide pipeline (Fig. 3g), small register file (Fig. 3i), and large second level cache (Fig. 3k). The low energy configurations tend to have a pipeline only a couple of instructions wide (Fig. 3a), only a few register file read ports (Fig. 3d), and a small L2 cache (Fig. 3e). However, they have moderately sized register files with only a few read and write ports into them, and average-to-large-sized branch predictors. These configurations trade off dynamic energy consumption for static energy savings. Were the structures smaller, then performance would drop and static energy consumption would rise, outweighing the benefits of lower dynamic energy consumption.

Having explored the impact of the parameters on the performance and energy of the benchmarks overall, the next section considers the variation across the design space on a per-program basis.

4 Program Characteristics

One of the main features of our predictor is that it uses the knowledge gathered from other programs to predict the behavior of a new program. As this section shows, similarities do exist between programs in the design space and are exploited later in Section 5 to build a machine-learning model that predicts across programs.

4.1 Per-Program Variation

Fig. 4 shows the characteristics of the design space on a per-program basis for cycles, energy, ED, and EDD. In each graph, we show the maximum of the space for each benchmark, then the 75 percent quartile, median, 25 percent quartile, and minimum. Note that the y-axis in each graph is on a logarithmic scale. Here, we have normalized each benchmark to a phase of 10 million instructions (the size of the SimPoint intervals) to allow comparisons between benchmarks.

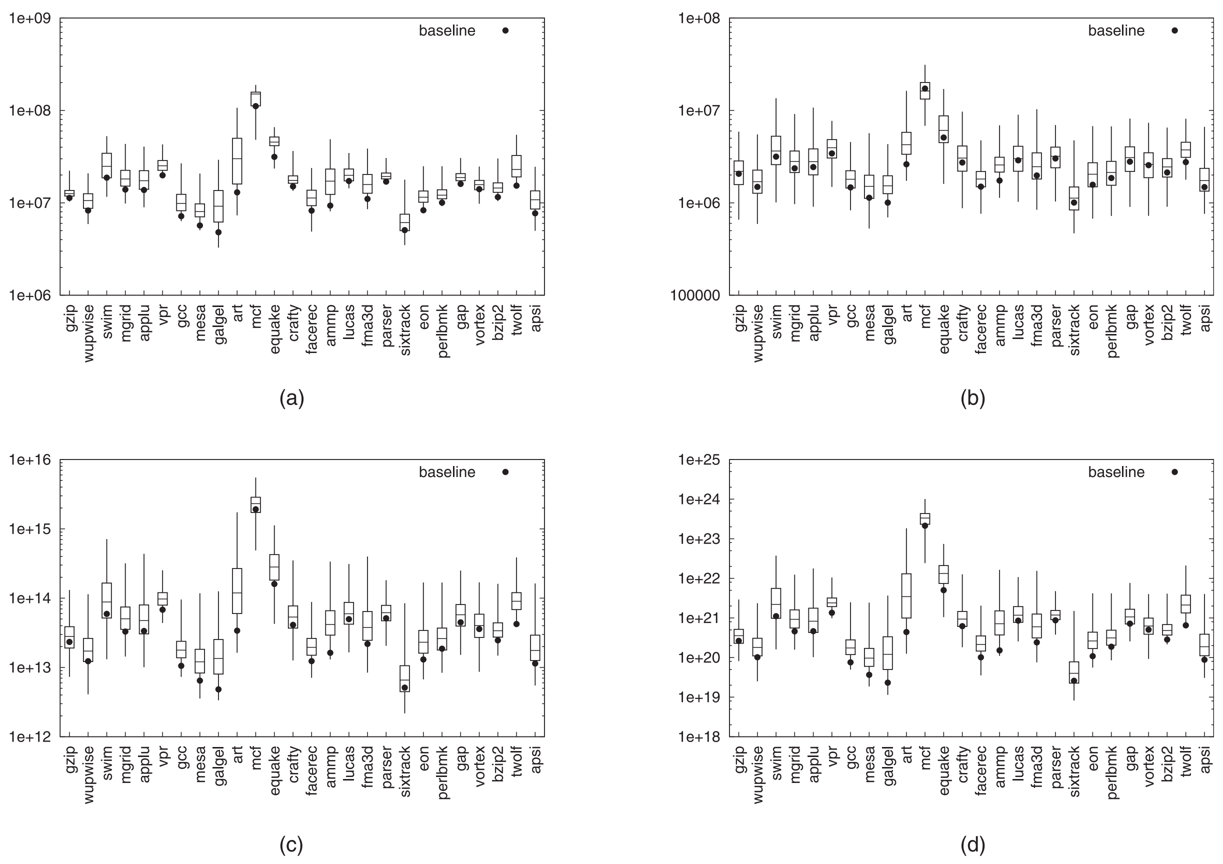


Fig. 4. Characteristics of the design space for the SPEC CPU 2000 programs for cycles, energy, ED, and EDD. Each graph shows the median, quartiles for 25 percent and 75 percent, minimum and maximum values for each benchmark, using a logarithmic y-axis. Also, shown is the performance of the baseline architecture for each program. (a) Cycles, (b) Energy, (c) ED, and (d) EDD.

As can be seen in Fig. 4a, the number of cycles taken for each simulation varies considerably between programs ranging from the longest, $2{*}10^8$ cycles, to the shortest, $2{*}10^6$ cycles. Some programs vary enormously, for example, *art* which varies between $7{*}10^6$ and $1{*}10^8$ cycles. Other programs, such as *parser*, vary only slightly (between $1.5{*}10^7$ and $2{*}10^7$). It is a similar story for energy, ED, and EDD too.

4.2 Program Similarities

As we wish to build a machine-learning model that can predict across programs, we need to determine similarities between programs that the model can learn. This similarity is expressed as the distance two programs' design spaces are from each other. We use the euclidean distance as a measure of distance using the 3,000 randomly selected microarchitectural configurations considered. This differs from previous work on measuring program similarities [19], [20], [21] where dynamic features (such as instruction mix or branch direction) were used. Here, we determine program similarity by directly using the results of running our benchmarks on randomly selected configurations from our design space.

Fig. 5 shows the dendrogram resulting from applying hierarchical clustering[1] on the programs. This shows the similarities between programs and has been used by other researchers in the field [22]. The horizontal lines join two branches together and the height on the y-axis gives the average distance between them. For example, for ED, the program *art* is a distance of 500 away from all other benchmarks. The higher the separation between branches, the less similar the programs in each branch are from each other.

1. The standard *hclust* function from the statistical package R-2.9.1 with the "average" method was used. Each data point was normalized on the baseline architecture.

Fig. 5 shows that across all metrics, *art* is very different from the other programs. Furthermore, it can be seen that *mcf* is significantly different from the others, especially when considering energy (Fig. 5b). From these observations, we can see that *art* and *mcf* are significantly different from the other benchmarks and will, therefore, be more difficult to predict. Despite these differences, many programs appear to be clustered and, therefore, similar. This will be exploited in the next section to build a model based on a linear regressor.

4.3 Summary

This section has presented the characteristics of our design space. We have explored how each program varies across the design space and presented the similarities between benchmarks using a hierarchical clustering approach. We now want to consider how these program similarities can be exploited by building a machine-learning model that can predict any point in the design space for a completely new program.

5 Predicting a New Program

We now describe our scheme where we use prior information about a number of previously seen programs to quickly and accurately predict the number of cycles, energy, and the

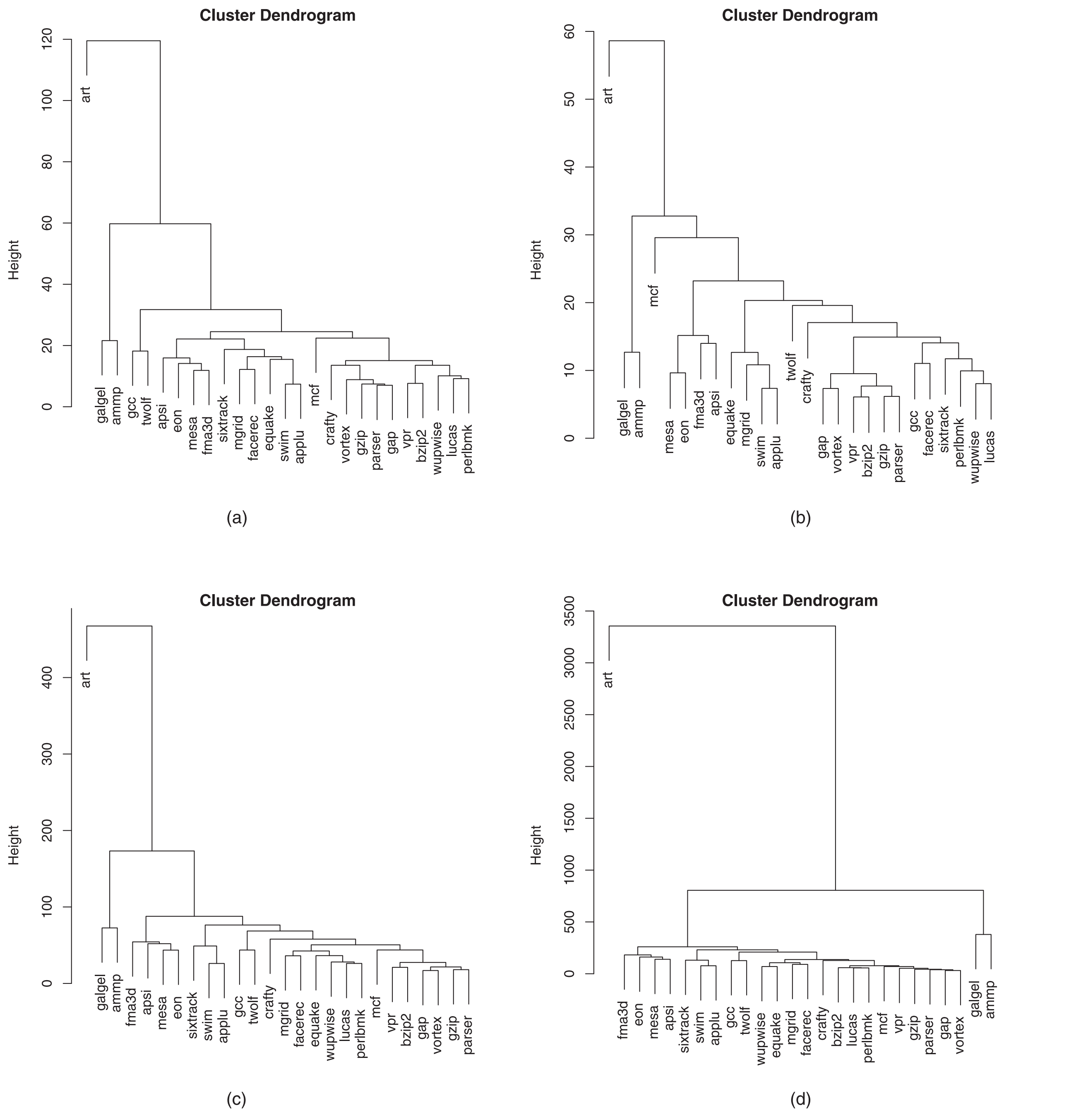


Fig. 5. Hierarchical clustering of SPEC CPU 2000 benchmarks using the euclidean distance. The average distance between the design space of any two groups of programs can be determined by looking at the height of the branch that connects them. For example, for cycles there is an average distance of 120 between *art* and the two programs *galgel* and *ammp* that differ themselves by a distance of 20. (a) Cycles, (b) Energy, (c) ED, and (d) EDD.

ED and EDD products of any new program within our design space. The model developed in this section is based on the observation that, while the program design spaces are highly complex and nonlinear, it is possible to express one such space as a simple linear combination of other program spaces.

5.1 Overview

Our model is based on a simple linear combination of the design spaces of several individual programs from the training set. Given this linear combination, we can accurately model the space of any new program. This assumption of linearity is based on the observations of Section 4.2 on program similarities.

Fig. 6 gives an overview of how our model works. First, the microarchitectural configuration of the new program we want to predict for is expressed in the form of a vector of parameters and is fed into the trained program-specific models. The output of these program-specific models is then used as an input to the linear regressor which predicts the performance of the new program for the particular configuration. This prediction is made possible by the extraction of *responses* from the new program which allow us to train the linear regressor. The next two sections explain in more detail how the model works.

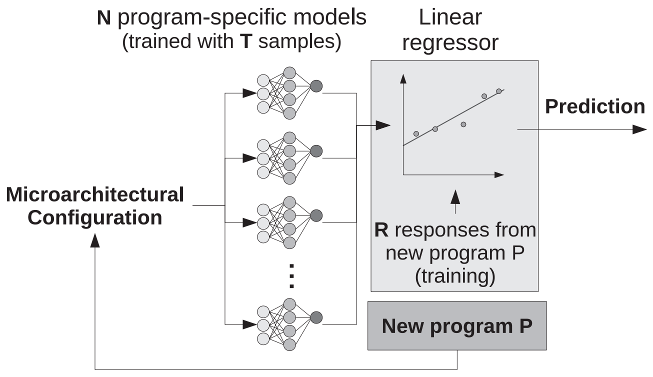

N program-specific models
(trained with T samples)
Linear regressor
Prediction
Microarchitectural Configuration
R responses from new program P (training)
New program P

Fig. 6. Our architecture-centric model. We train **N** program-specific predictors (one for each training program) offline with a number **T** of training simulations. The results are fed into a linear regressor along with a *responses* vector consisting of **R** simulations from a new program P to provide a prediction for any configuration in the microarchitectural design space.

5.2 Program-Specific Models

Our scheme builds on top of program-specific predictors. We use artificial neural networks (ANNs) [23] to build these predictors similar to those used in [7], although we could have used any other related approach [8], [9], [10], [24]. They consist of a multilayer perceptron with one hidden layer of 10 neurons and use the sigmoid activation function for the input and hidden layers and the linear function for the output layer. We train each predictor offline on a number of simulations, **T**, from the training programs.

5.2.1 Artificial Neural Networks

ANNs map the input variables to a response or prediction in a nonlinear way. It uses a network of neurons (simple elements that sum their inputs) connected to each other by a weighted edge. An example is shown in Fig. 7. In our case, the input $\mathbf{x}$ of the network is a vector representing the architectural configuration. For instance, the baseline architecture whose parameters are shown in Table 1 is encoded as the vector $\mathbf{x_{baseline}} = (4, 96, 32, 48, 96, 8, 4, 16, 4, 16, 32, 32, 2)$. The output $\hat{y}$ of the network is the target metric we want to predict: cycles, energy, ED, or EDD.

Technically, the ANN is composed of a *feed-forward network* which uses the *back-propagation* algorithm to train and update the weights of the neurons according to a learning rule. Once trained, the output of each neuron is computed as follows:

$$f(\mathbf{x}) = g\left(\sum_i \omega_i \cdot \mathbf{x}_i\right), \tag{1}$$

where $g()$ is an activation function. This activation function 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 normalize 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

$$\hat{y} = \sum_i \left(\omega_i^o \cdot tanh\left(\sum_j \omega_{j,i}^h \cdot x_i\right)\right), \tag{2}$$

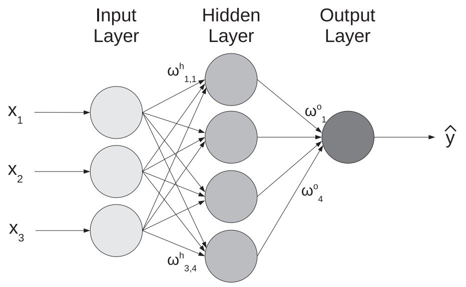


Fig. 7. Example of an artificial neural network where a simple multilayered neural network is shown. This network is composed of three layers; input, hidden, and output.

where ω^o are the weights associated with the output layer and ω^h the weights of the hidden layer, as can be seen in Fig. 7.

The training phase consists of finding the optimal weights ω that minimize the prediction error. This is achieved using the *back-propagation* algorithm. The interested reader can find more details in [23].

5.3 Architecture-Centric Predictor

Our technique for predicting a new program using prior knowledge is based on offline training of the program-specific predictors combined with a small number of simulations, **R**, from the design space of the new program. We call these few simulations the *responses*.

The architecture-centric model is a simple linear regressor shown in Fig. 6. In effect, the behavior of the architecture space on a new program can be modeled as a linear combination of their behavior on previously seen programs. The small number of simulations **R**, called *responses*, are used to find weights which determine the right combination of previously seen models that best capture the behavior of the new program. This surprisingly simple approach is actually highly accurate, as we show in Section 6.

5.3.1 Linear Regression

As its name suggests, this technique 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 minimize the squared error between real outputs $\mathbf{y}$ and the predictions $\hat{\mathbf{y}}$. This sum is computed as follows:

$$\hat{\mathbf{y}} = \beta_0 + \beta_1 \cdot \mathbf{X}_{,1} + \cdots + \beta_m \cdot \mathbf{X}_{,m}. \tag{3}$$

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

$$\sum_{i=1}^{n} \sum_{j=1}^{m} (\mathbf{X_{i,j}} \cdot \beta_j - y_i)^2. \tag{4}$$

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

$$\beta = (\mathbf{X} \cdot \mathbf{X}^T)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}. \tag{5}$$

Fig. 8 shows an example of linear regression. The thick line minimizes the total squared error. In this example, the line that estimates the data is defined by $\hat{y} = \beta_0 + \beta_1 \cdot x$. The weight β_0 is in fact the intercept and β_1 the gradient of the linear equation.

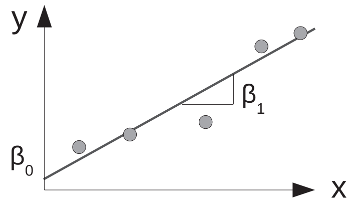


Fig. 8. Example of linear regression where the resulting regression line is shown for five points. This line is defined as $y = \beta_0 + \beta_1 \cdot x$ where $\beta_0 = 0.59$ and $\beta_1 = 0.21$.

In the case of our model, the $\mathbf{R}$ *responses* form the vector $\mathbf{y}$ and are used to find the weights β associated with each training program. These *responses* represent the target metric (cycles, energy, ED, or EDD) from the new program. For each training program j, $\mathbf{R}$ *responses* are also extracted to build the vector $\mathbf{X_{.j}}$ (these do not require any new simulations, since they have already been performed during the offline training phase). Armed with the vectors $\mathbf{X_{.j}}$ that form the matrix $\mathbf{X}$ and the vector $\mathbf{y}$, the weights β can be easily determined using (5).

6 Choice of Optimal Model Parameters

This section now evaluates the optimal parameters of the model presented in the previous section. The evaluation methodology is first reviewed followed by the exploration of the optimum model parameters. The performance of the model is later evaluated in Section 7 individually for each of the SPEC CPU 2000 and MiBench programs.

6.1 Methodology

In order to evaluate the accuracy of our predictors, we use the *relative mean absolute error* (rmae) defined as $rmae = |(\mathbf{prediction} - \mathbf{actual})/\mathbf{actual}| \cdot 100\%$. This metric tells us how much error there is between the predicted and actual values. For example, an rmae of 100 percent would mean that the model, on average, would be predicting a value that was double the actual value.

Although rmae is important, it is not a good measure of how accurately the model predicts the shape or trend of the space. Since we want to use our predictor to distinguish between good and bad architectures (i.e., low or high cycles, energy, ED, or EDD), we need a metric that describes how accurately the predictor models the shape of the space.

To analyze the quality of our models, we, therefore, use the correlation coefficient. The correlation between two variables is defined as $corr = \mathbf{cov}(\mathbf{X}, \mathbf{Y})/\sigma_X \cdot \sigma_Y$, where σ_X and σ_Y represent the standard deviation of variables X and Y, respectively, and $\mathbf{cov}(X, Y)$ is the covariance of variables X and Y. The correlation coefficient only produces values between -1 and 1. At the extreme, a correlation coefficient of 1 means the predictor perfectly models the shape of the real space. A correlation coefficient of 0 means there is no linear relation between the predictions and the actual space.

Unless otherwise stated, all our predictors are validated using the 3,000 sampled configurations discussed in Section 3.3.

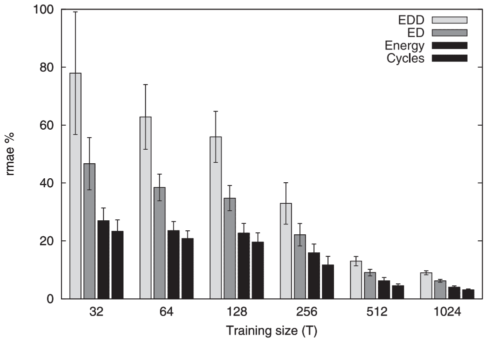
6.2 Model Parameters

Fig. 9 shows the rmae and correlation coefficient when varying the number of training simulations (per training program) $\mathbf{T}$, to use for our program-specific predictors to predict EDD, ED, energy, and cycles. We selected the training simulations from the design space using uniform random sampling. In Fig. 9a, we can see that, as expected, the rmae decreases as the size of the training set increases. The same is shown in Fig. 9b for the correlation coefficient, i.e., as you increase the size of the training data, the error gets smaller and the correlation increases.

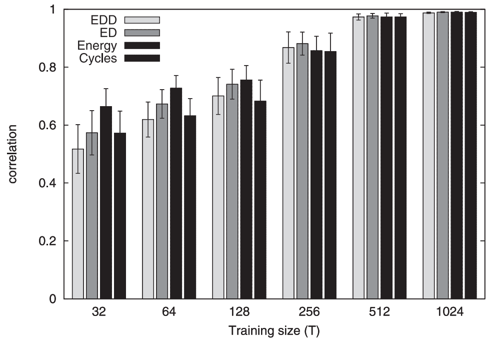
From the graphs in Fig. 9, we can conclude that we should use $\mathbf{T} = 512$ configurations per training program as input to our model since this provides low rmae and high correlation for EDD, ED, cycles, and energy. Increasing the number of configurations per training program gives only minor improvement.

Now that the optimum number of training configurations for the program-specific predictors has been determined, we wish to find the optimum number of *responses* $\mathbf{R}$, needed from the new program to complete our architecture-centric model. Fig. 10 shows the rmae and correlation coefficient for EDD, ED, cycles, and energy for different number of *responses* from the new program when all other benchmarks have been used as training with $\mathbf{T} = 512$ configurations.

It is immediately clear, looking at both Figs. 10a and 10b that using more than 32 *responses* does not bring further benefits in terms of either rmae or correlation coefficient.

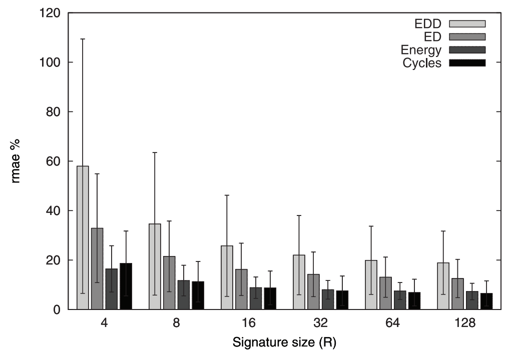


(a)

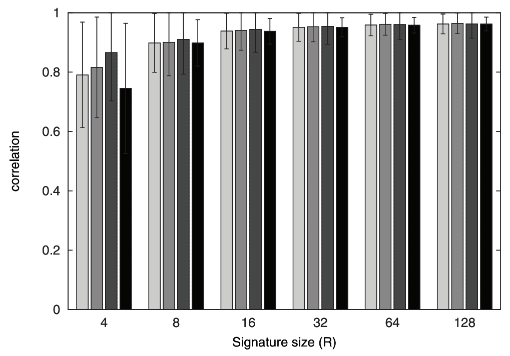


(b)

Fig. 9. The rmae and correlation (along with standard deviation) of the program-specific predictors when using varying numbers of training configurations $\mathbf{T}$. We average across all programs and show results for EDD, ED, cycles, and energy. This shows that $\mathbf{T} = 512$ is a good trade-off in terms of correlation and accuracy against the number of training configurations required. (a) Relative mean absolute error. (b) Coefficient of correlation.



(a)



(b)

Fig. 10. The rmae and correlation (with standard deviation) of our architecture-centric predictor when varying the number of *responses* **R**, from the new program. In these graphs, we have fixed the number of training configurations to 512. We average across all programs and show results for EDD, ED, cycles, and energy. This shows that beyond a size of 32, we do not get significant further improvement; hence, we fix the number of *responses* to be 32 simulations. (a) Relative mean absolute error. (b) Coefficient of correlation.

Using $\mathbf{R} = 32$, we obtain a correlation coefficient of 0.95 for all four metrics and an rmae of 7, 7, 14, and 22 percent for cycles, energy, ED, and EDD, respectively. Hence, we fix the number of *responses* to be $\mathbf{R} = 32$, along with the number of training configurations which we have already fixed at $\mathbf{T} = 512$. We, thus, show that in our space, we only need 32 simulations from any new program to characterize it. Therefore, these 32 *responses* enable us to accurately predict the entire design space for the new program.

6.3 Summary

This section has evaluated the optimal model's parameters. As seen, $\mathbf{T} = 512$ simulations are used to train each of the program-specific predictor. These are then used to predict the design space of any new program using as few as $\mathbf{R} = 32$ simulations; the *responses*. In the next section, the performance of the model is evaluated for each program of SPEC CPU 2000 and MiBench when using these parameters.

7 Evaluation on SPEC2K and MiBench

Having determined the model's optimum parameters, this section now evaluates its performance on the SPEC CPU 2000 and MiBench benchmark suites.

7.1 Evaluation Methodology

We evaluate our model using *N-fold cross validation* with $\mathbf{T} = 512$ randomly selected configurations for the training of the program-specific predictors. We repeat this 20 times. This common process ensures that different configurations are used during training and testing, allowing us to generalize our conclusions to the whole design space.

Leave-one-out cross validation is used to build the program training set by leaving out one application at a time. So if we have N programs, the training set will be composed of the $N - 1$ programs and the test set will consist of the unique Nth program left out. This is then repeated for all programs. It ensures a fair evaluation of our models and is common practice within the machine-learning community.

7.2 Prediction Error for SPEC CPU 2000

Fig. 11 shows the training and testing error of our model for each of the four metrics for each program in the SPEC CPU 2000 benchmark suite. The training error is derived from the error of the model on the training data (32 *responses* for each program) while the testing error is the error 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 eight percent for cycles and energy, 14 percent for ED, and 21 percent for EDD. Some programs have a larger error in comparison with others. For instance, program *art* has an error of 32 percent for cycles and 19 percent for energy and program *mcf* an error of 16 percent for cycles and 17 percent for energy. As seen in Section 4.2, these programs are very different from the others. Therefore, it is difficult to use the knowledge gathered from the training programs and this inevitably leads to a higher error for these programs.

Interestingly, it is possible to use the training error as an indicator of the model's performance: the higher the training error, the higher the 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. This approach provides the designer with additional information that he can use to decide whether a program-specific model should be built in order to achieve higher accuracy for programs with unique behavior.

7.3 Predicting MiBench from SPEC CPU 2000

So far, leave-one-out cross validation was used to verify and assess the performance of our model. 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 MiBench. Moreover, since MiBench benchmarks are mainly targeted at embedded systems, it enables testing of the models across a different application domain.

Fig. 12 shows the model's error when predicting MiBench from SPEC CPU 2000. The average error is about six percent for cycles, seven percent for energy, 12 percent for ED, and 18 percent 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 very different behavior from the others. Therefore, the model's accuracy for these programs is worse than for the others, resulting in an increase in the average

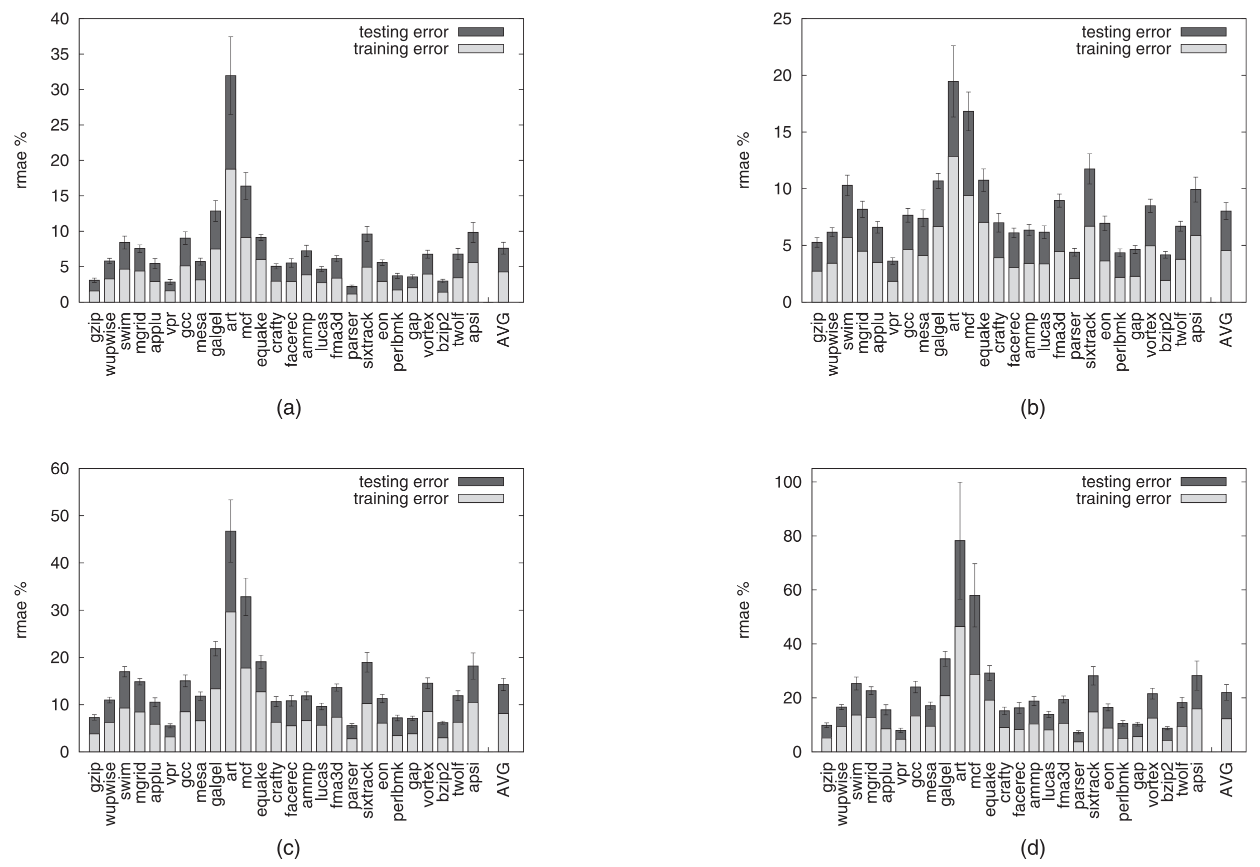


Fig. 11. Training and actual mean error for each program of SPEC CPU 2000 (the lower the better). The actual error corresponds to the prediction error when testing on the remaining points of the space not used for training. The standard deviation is also shown since the training has been repeated 20 times picking each time different samples. (a) Cycles, (b) Energy, (c) ED, and (d) EDD.

error. However, if one dismisses these two programs, there is no fundamental difference in terms of error between these two benchmark suites.

Once again, the training error can be used to identify programs for which the model leads to a higher error rate. Programs *tiff2rgba* and *patricia*, for instance, show a higher training error than for the others. Based on this information, program-specific models could be built instead, in order to achieve a higher level of accuracy.

If we build the dendrogram diagrams showing the distances between the programs from MiBench and those from SPEC (as seen in Section 4.2 but not shown here for space reasons), we can clearly see that these few MiBench programs with a higher prediction error are in fact quite different from any within SPEC. In fact, our model is able to determine this automatically using the *training error*. As already mentioned in Section 7.2, this information can be used to detect programs that behave significantly differently from those encountered during the training phase. As such, this has the potential to give new insights to the architect about the design space.

7.4 Comparison with Program-Specific Predictors

Given an equal number of simulations from a new program, we are interested in seeing how our scheme performs compared with the program-specific predictor proposed by İpek et al. [7]. The purpose of this comparison is to see what the effects of using prior knowledge from the training programs are when predicting the design space of a new program.

Fig. 13 shows the coefficient of correlation and the error for cycles, energy, ED, and EDD when varying the number of simulations required by the two models, averaged across all programs. For the program-specific model, the simulations are used as training data whereas in our model, they are used as the *responses* for the new program. As can be seen, for each metric, our scheme is more correlated with the actual data than the program-specific approach, because our scheme can apply knowledge from the programs it has previously been trained on. The error is also much lower when a small number of simulations are used.

When considering cycles, our model has a relative mean absolute error of just seven percent and a correlation coefficient of 0.95 with 32 simulations or *responses*. The program-specific model performs significantly worse with a prediction error of 24 percent and a correlation coefficient of only 0.55. If we consider EDD, we achieve an error rate of just 20 percent, on average, with as few as 32 simulations, whereas the error of program-specific predictor is 75 percent for the same number of simulations from the new program. If we wish to obtain the same accuracy with the program-specific predictor, we would need about 350 simulations—an order of magnitude more than our model. This clearly demonstrates the benefit of using prior knowledge in order to drastically reduce the number of simulations required to explore the design space of new programs.

Note that the program-specific models are more accurate for a large number of simulations. This is due to the fact that some programs differ significantly from others, as seen in Fig. 5. As the number of training samples for the program-specific predictor increases, it can learn more about these

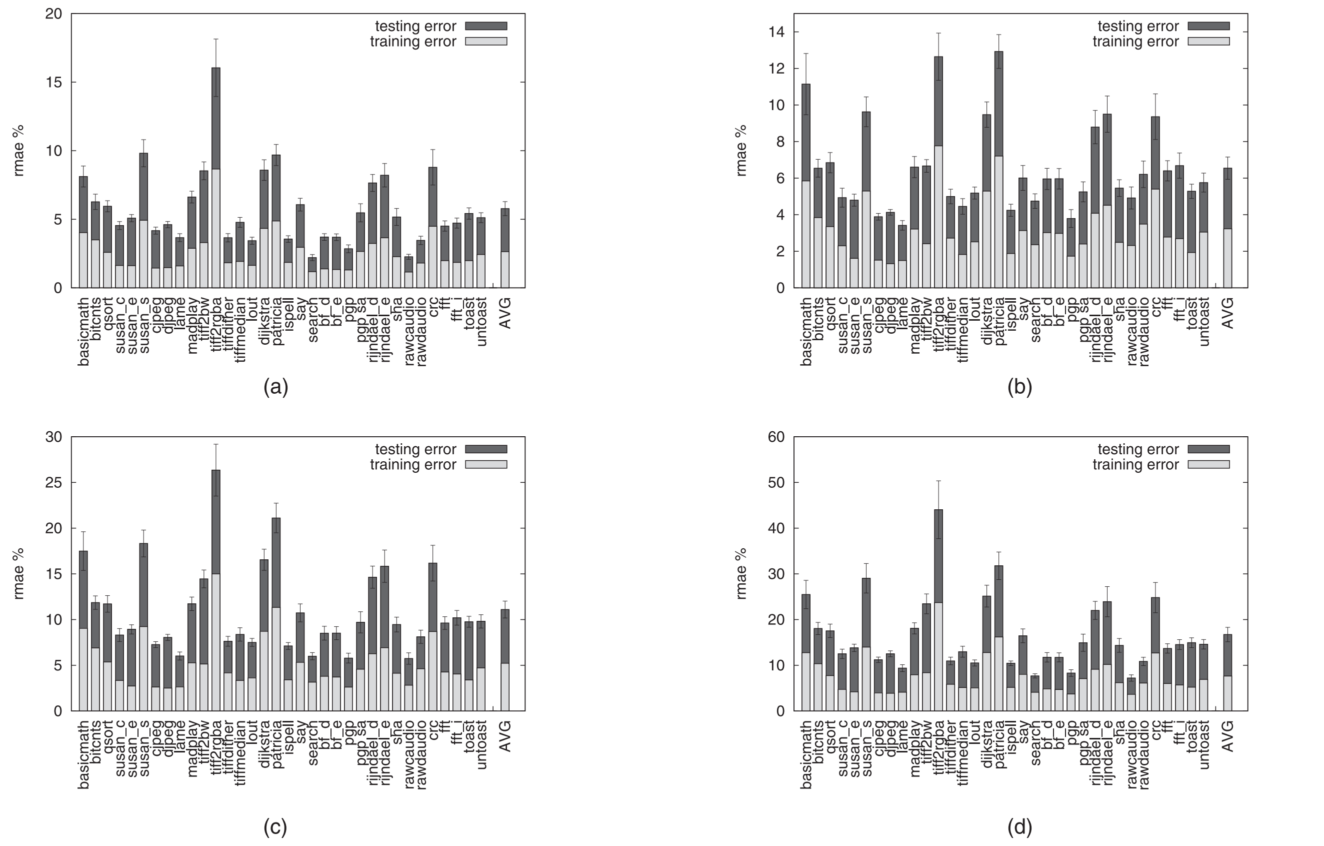


Fig. 12. Training and actual mean error of the model trained on SPEC CPU 2000 for each program in MiBench (the lower the better). (a) Cycles, (b) Energy, (c) ED, and (d) EDD.

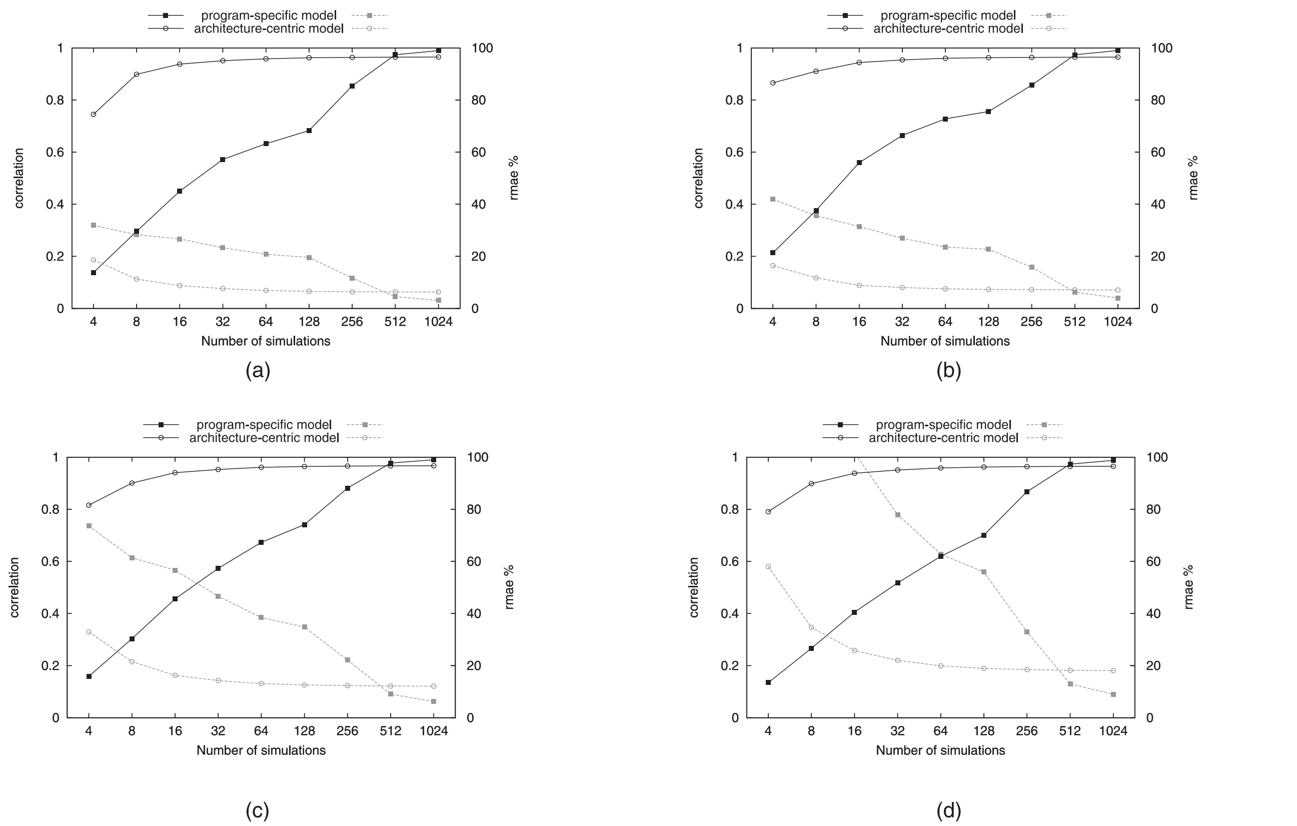


Fig. 13. Coefficient of correlation (in black) and error (rmae in gray) when varying the number of simulations for both the program-specific model and our architecture-centric approach. In the program-specific model, these simulations are used as training data; in our model, they are used as *responses* for the new program. We average across all programs and show results for cycles, energy, ED, and EDD. (a) Cycles, (b) Energy, (c) ED, and (d) EDD.

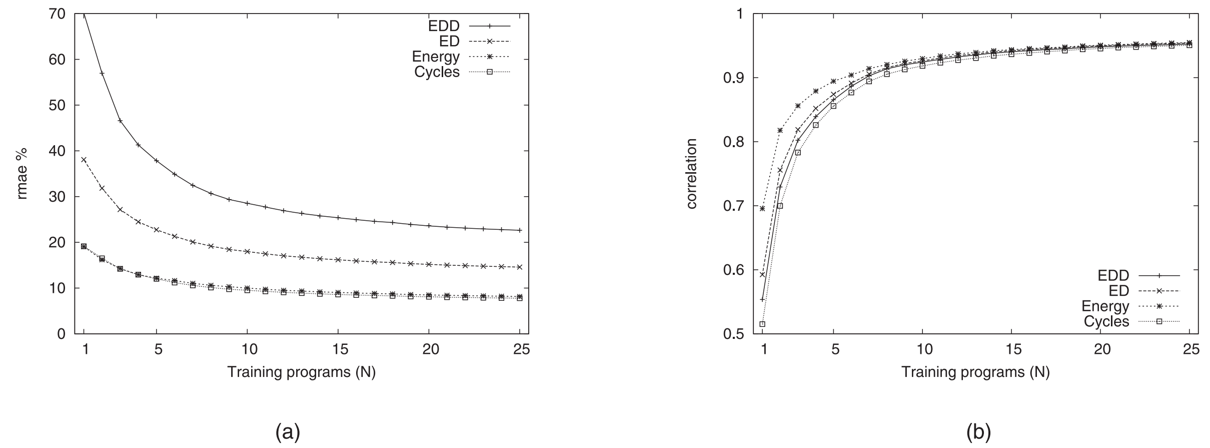


Fig. 14. The rmae and correlation for our architecture-centric predictor when varying the number of offline training programs. The x-axis shows the number of offline training programs and the y-axis shows the results when predicting on the remaining benchmarks. We show results for EDD, ED, energy, and cycles. Each training program is run 512 times and the new program to be predicted given 32 *responses*. (a) Relative mean absolute error. (b) Coefficient of correlation.

programs because it is specialized to them. On the other hand, no matter how many more training samples are given to the architecture-centric approach, it cannot learn more about these programs. This is because there are no similar programs to learn from within the rest of the benchmark suite and it must generalize from the programs that it has been trained on.

7.5 Summary

This section has demonstrated that our model can be used to accurately predict the design space of new programs. Furthermore, we have shown how the model can be trained on one benchmark suite, SPEC CPU 2000, and predict the design space of programs from another suite, MiBench. In addition, a comparison with a program-specific model has been conducted showing that our approach predicts the design space of new programs using an order of magnitude fewer simulations. The next section investigates the costs of offline training and shows how this can be reduced if needed.

8 Reducing the Training Costs

The training of our architecture-centric model is performed offline and, as such, is not taken into account when evaluating new programs. One criticism could be that this training does not come for free, so we have considered the extent to which we can reduce it and its effect on the accuracy of prediction for new programs.

The training cost of our model can be reduced by using fewer programs to train with. As seen in Section 4.2, many programs are similar to others; therefore, only a subset of all the programs are needed to build an accurate predictor. Fig. 14 shows the accuracy of our model in terms of error and correlation coefficient when varying the number of programs in the training set using 32 *responses*. We chose the training programs at random, repeating each experiment 20 times. However, greater accuracy could be achieved by selecting the benchmarks based on the information in Fig. 5. As the number of programs increases, both the rmae and the correlation coefficient tend to improve. With 15 programs, both metrics reach a plateau and adding more programs makes only a negligible difference. When we use five training programs, the correlation coefficient is higher than 0.85 for all the metrics, which is high enough to use this predictor accurately.

This shows that the one-off cost of training on between five and 15 programs results in a highly accurate predictor, capable of predicting the performance of all architecture configurations on any new program with just 32 further simulations.

9 Related Work

This section considers the work related to the design space exploration of new microarchitectures. It first reviews the work in benchmark suite characterization, explores the existing methodologies that reduce simulation time, and finally presents work related to analytic models and performance predictors.

9.1 Benchmark Suite Characterization

One way of performing efficient design space exploration consists of analyzing the programs and extracting key characteristics from them. Using this characterization, it is possible to group similar programs and reduce the number of simulations.

Saavedra and Smith [25] extracted dynamic program information to estimate execution time and find similar programs. They used the number and type of instructions executed, such as arithmetic 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 simple analytic models. Eeckhout et al. [26] later extended this work by adding more program features; in particular, branch prediction statistics, ILP, and cache miss rates.

Finally, Bird et al. [27] characterized the SPEC CPU 2000 benchmark suite in terms of how the programs stress the branch predictor, caches, and features specific to the processor they use. Our space characterization in Section 3.4 differs from theirs in that we vary the microarchitectural parameters and evaluate the effects on different microarchitectural structures.

9.2 Simulation Methodologies

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 two main techniques used to reduce simulation time.

Sampling techniques [1], [2] approach the problem by reducing the number of instructions needing to be simulated in cycle-accurate mode. This dramatically decreases the time required for simulation. With SimPoint [1], a statistical analysis of the program's trace is performed to combine results from the instructions actually run. With SMARTS [2], small samples from the program are taken over a set number of instructions. These approaches are orthogonal to our technique and, in fact, we do use SimPoint to speed up the simulations we perform both to train our predictor and to verify it.

Statistical simulation [28], [29], [30] is based on the same idea as sampling but goes a step further. The simulator is modified so that it symbolically executes the instructions based on a statistical model. This technique requires the extraction of program characteristics that are related to the microarchitecture under consideration. Thus, if any major changes occur, new features could be needed to continue to characterize the program. Eyerman et al. [31] focus their two-phase search using statistical simulation. Search techniques such as this can be used seamlessly with our approach.

9.3 Analytic Models

Analytic models have been proposed [3], [4] to reduce simulation cost while maintaining enough accuracy for design space exploration. More specialized models, such as cache simulators [5] have also been proposed. Unfortunately, these approaches require a large amount of knowledge about the microarchitecture they attempt to explore and, furthermore, they need to be built by hand. When major changes are made to the microarchitecture design, the models need to be updated [6]. In contrast, our technique focuses on building such a predictor automatically and can easily accommodate any future microarchitecture design changes.

9.4 Single-Program Predictors

There have been many recently proposed schemes for microarchitecture design space exploration based on linear regressors [8], artificial neural networks [7], [32], radial basis functions [24], and spline functions [9], [10]. The linear regressor approach [8] is, in fact, simply used to identify the key parameters in the design space. No measurements are given as to its accuracy and as such it can only be used to give hints to the designer. The other schemes are similar to each other in terms of accuracy [11], [12].

A fundamental difference between our scheme and these performance predictors resides in the fact that our model characterizes the architecture space independently of the programs being simulated, rather than modeling the program-specific architectural space. This enables our approach to predict new programs with low overhead.

9.5 Trans-Program Predictors

An interesting approach taken by Hoste et al. [20] uses a linear model to combine program design spaces, clustering benchmarks based on program features. We do not use features in our approach because they can be difficult to identify and might vary depending on the architecture under consideration. Hence, our scheme is more versatile since it can be applied to any program and architecture.

Close to our work is a cross-program learning approach to predict the performance of a new program on an unseen architecture [33]. The model, however, has to be retrained when a new program is considered and no comparison is made with existing approaches. In addition, the predictor only achieves a two percent improvement over simply predicting the average of the training data, thus showing little cross-program learning. Other work has applied this type of learning to the software optimization space for learning across programs [34].

Finally, this present paper is an extension to the work published in Micro [35] in 2007. It presents additional results for the EDD target metric, extends the analysis section with a thorough evaluation of the SPEC CPU 2000 benchmark suite in terms of program similarities, and shows how the predictor can be trained on one benchmark suite, SPEC CPU 2000 to predict another, MiBench.

10 Conclusions

This paper has proposed a novel approach to design space exploration using prior knowledge to predict energy, cycles, ED, or EDD. Our model can accurately predict the design space of any of the SPEC CPU 2000 programs and even across benchmark suites by predicting MiBench when trained on SPEC CPU 2000. We show when predicting performance that our architecture-centric model has a relative mean absolute error of just seven percent and a correlation coefficient of 0.95. This significantly outperforms a recently proposed program-specific predictor which has an rmae of 24 percent and correlation coefficient of 0.55 given the same number of simulations. We address the cost of offline training that our model requires and show that, given the same training budget, our predictor still has a better rmae than a program-specific predictor.

In conclusion, our architecture-centric approach can accurately predict the performance, energy, ED, or EDD of a range of programs within a massive microarchitectural design space, requiring just 32 simulations, known as *responses*, from any new program and outperforming all other approaches.

Acknowledgments

This work has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF) [36]. The ECDF is partially supported by the eDIKT initiative [37].

References

[1] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

[2] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe, "Smarts: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," *Proc. 30th Ann. Int'l Symp. Computer Architecture (ISCA)*, 2003.

[3] T.S. Karkhanis and J.E. Smith, "A First-Order Superscalar Processor Model," *Proc. 31st Ann. Int'l Symp. Computer Architecture (ISCA)*, 2004.

[4] D.B. Noonburg and J.P. Shen, "Theoretical Modeling of Superscalar Processor Performance," *Proc. 27th Ann. Int'l Symp. Microarchitecture (MICRO)*, 1994.

[5] R.A. Sugumar and S.G. Abraham, "Set-Associative Cache Simulation Using Generalized Binomial Trees," *ACM Trans. Computer Systems,* vol. 13, no. 1, pp. 32-56, 1995.

[6] D. Ofelt and J.L. Hennessy, "Efficient Performance Prediction for Modern Microprocessors," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems,* 2000.

[7] E. İpek, S.A. McKee, R. Caruana, B.R. de Supinski, and M. Schulz, "Efficiently Exploring Architectural Design Spaces via Predictive Modeling," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII),* 2006.

[8] P.J. Joseph, K. Vaswani, and M.J. Thazhuthaveetil, "Construction and Use of Linear Regression Models for Processor Performance Analysis," *Proc. 12th Int'l Symp. High-Performance Computer Architecture (HPCA),* 2006.

[9] B.C. Lee and D.M. Brooks, "Illustrative Design Space Studies with Microarchitectural Regression Models," *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture (HPCA),* 2007.

[10] B.C. Lee and D.M. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII),* 2006.

[11] 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," *Proc. 12th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP),* 2007.

[12] K. Vaswani, M.J. Thazhuthaveetil, Y.N. Srikant, and P.J. Joseph, "Microarchitecture Sensitive Empirical Models for Compiler Optimizations," *Proc. Int'l Symp. Code Generation and Optimization (CGO),* 2007.

[13] "The Standard Performance Evaluation Corporation (SPEC) CPU 2000 Benchmark Suite," http://www.spec.org/cpu2000/, 2011.

[14] 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," *Proc. IEEE Int'l Workshop Workload Characterization (WWC-4),* 2001.

[15] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Int'l Symp. Computer Architecture (ISCA),* 2000.

[16] T. Austin, "The Simplescalar Toolset," http://www.simplescalar.com, 2011.

[17] D. Tarjan, S. Thoziyoor, and N.P. Jouppi, "Cacti 4.0," Technical Report HPL-2006-86, HP Laboratories Palo Alto, 2006.

[18] C.-H. Hsu, W. chun Feng, and J.S. Archuleta, "Towards Efficient Supercomputing: A Quest for the Right Metric," *Proc. First IEEE Workshop High-Performance, Power-Aware Computing (in Conjunction with 19th Int'l Parallel and Distributed Processing Symp. (IPDPS)),* 2005.

[19] A. Phansalkar, A. Joshi, L. Eeckhout, and L.K. John, "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* 2005.

[20] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L.K. John, and K.D. Bosschere, "Performance Prediction Based on Inherent Program Similarity," *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2006.

[21] A. Phansalkar, A. Joshi, and L.K. John, "Subsetting the SPEC CPU2006 Benchmark Suite," *ACM SIGARCH Computer Architecture News,* vol. 35, no. 1, pp.69-76, Mar. 2007.

[22] A. Joshi, A. Phansalkar, L. Eeckhout, and L. John, "Measuring Benchmark Similarity Using Inherent Program Characteristics," *IEEE Trans. Computers,* vol. 55, no. 6, pp. 769-782, June 2006.

[23] C. Bishop, *Neural Networks for Pattern Recognition.* Oxford Univ. Press, 2005.

[24] P.J. Joseph, K. Vaswani, and M.J. Thazhuthaveetil, "A Predictve Performance Model for Superscalar Processors," *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO-39),* 2006.

[25] R.H. Saavedra and A.J. Smith, "Analysis of Benchmark Characteristics and Benchmark Performance Prediction," *ACM Trans. Computer Systems,* vol. 14, no. 4, pp. 344-384, 1996.

[26] L. Eeckhout, H. Vandierendonck, and K.D. Bosschere, "Workload Design: Selecting Representative Program-Input Pairs," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2002.

[27] S. Bird, A. Phansalkar, L.K. John, A. Mericas, and R. Indukuru, "Performance Characterization of SPEC CPU Benchmarks on Intel's Core Microarchitecture Based Processor," *Proc. SPEC Benchmark Workshop,* 2007.

[28] 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," *Proc. 31st Ann. Int'l Symp. Computer Architecture (ISCA),* 2004.

[29] M. Oskin, F.T. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs," *Proc. 27th Int'l Symp. Computer Architecture (ISCA),* 2000.

[30] R. Rao, M. Oskin, and F.T. Chong, "Hlspower: Hybrid Statistical Modeling of the Superscalar Power-Performance Design Space," *Proc. Int'l Conf. High Performance Computing (HiPC),* 2002.

[31] S. Eyerman, L. Eeckhout, and K.D. Bosschere, "Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors," *Proc. Design, Automation and Test in Europe (DATE),* 2006.

[32] E. İpek, B.R. de Supinski, M. Schulz, and S.A. McKee, "An Approach to Performance Prediction for Parallel Applications," *Proc. Int'l Euro-Par Conf.,* 2005.

[33] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, "Using Predictive Modeling for Cross-Program Design Space Exploration in Multicore Systems," *Proc. 16th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2007.

[34] 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," *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems (CASES),* 2006.

[35] C. Dubach, T. Jones, and M. O'Boyle, "Microarchitectural Design Space Exploration Using an Architecture-Centric Approach," *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO-40),* 2007.

[36] "The Edinburgh Compute and Data Facility (ECDF)," http://www.ecdf.ed.ac.uk, 2011.

[37] "The eDIKT Initiative," http://www.edikt.org, 2011.

Christophe Dubach received the MSc degree in computer science from EPFL, Switzerland, and the PhD degree in informatics from the University of Edinburgh in 2009. He is an RAEng/EPSRC research fellow in the Institute for Computing Systems Architecture at the University of Edinburgh and is currently a visiting researcher at IBM Watson. His research interests include codesign of both computer architecture and optimizing compiler technology, adaptive microprocessor and software, and the application of machine learning in these areas.

Timothy M. Jones received the MEng degree in computer science from the University of Bristol and the PhD degree in informatics from the University of Edinburgh. He is a postdoctoral researcher at the University of Edinburgh where he holds a research fellowship from EPSRC and the Royal Academy of Engineering. His research interests include computer architecture and compiler optimization, with an emphasis on power reduction and the application of machine learning to compiler and microarchitectural design. He is currently on sabbatical at Harvard University for the whole of 2010.

Michael F.P. O'Boyle graduated with the PhD degree in computer science from the University of Manchester in 1992. Prior to moving to a lectureship at Edinburgh in 1997, he was a visiting scientist at INRIA and a visiting research fellow at the University of Vienna. Since then, he has been a visiting scholar at Stanford University and a visiting professor at UPC Barcelona. In 2006, he was awarded a Personal Chair in Computer Science. His main research interests are in adaptive compilation, compiler/architecture codesign, and automatic compilation for multicore systems.