Rapid Early-Stage Microarchitecture Design Using Predictive Models

Christophe Dubach, Timothy M. Jones, Michael F.P. O'Boyle
Members of HiPEAC
School of Informatics, University of Edinburgh, UK
christophe.dubach@ed.ac.uk, {tjones1,mob}@inf.ed.ac.uk

***Abstract*— The early-stage design of a new microprocessor involves the evaluation of a wide range of benchmarks across a large number of architectural configurations. Several methods are used to cut down on the required simulation time. Typically, however, existing approaches fail to capture true program behaviour accurately and require a non-negligible number of training simulations to be run.**

We address these problems by developing a machine learning model that predicts the mean of any given metric, e.g. cycles or energy, across a range of programs, for any microarchitectural configuration. It works by combining only the most representative programs from the benchmark suite based on their behaviour in the design space under consideration. We use our model to predict the mean performance, energy, energy-delay (ED) and energy-delay-squared (EDD) of the SPEC CPU 2000 and MiBench benchmark suites within our design space. We achieve the same level of accuracy as two state-of-the-art prediction techniques but require five times fewer training simulations. Furthermore, our technique is scalable and we show that, asymptotically, it requires an order of magnitude fewer simulations than these existing approaches.

I. Introduction

During early-stage design of new processors, architects evaluate a large number of architectural configurations across a range of benchmarks, searching for designs that meet the constraints of the project and can be put forward for further, detailed evaluation. At this stage in the design they look for configurations that perform well on average across a number of programs, rather than looking at the individual performance of each benchmark.

Cycle-accurate simulators play a crucial role in this process. However, detailed modelling of the microarchitecture and execution of large benchmark suites with realistic workloads means that simulation is slow. Any technique that can accurately reduce the required simulation time is beneficial.

Researchers have tackled this by exploiting similarities between programs [1], using microarchitecturally-independent features to cluster benchmarks and select only a few representative programs for simulation. However, as we show in this paper, although benchmarks are similar when considering their program features, they may behave quite differently in terms of execution time or energy across a microarchitectural design space. As we shall see, clustering of programs needs to be performed using their actual behaviour in the design space, rather than using independent metrics.

Recently, machine learning has also been proposed to further reduce simulation time [2], [3], [4], [5], [6]. These techniques model the entire design space by training a predictive model using several simulation runs from the program. However, existing techniques can only be used to predict the program with which they have been trained. Subsequent work has considered training on one set of benchmarks to predict another [7], [8]. However, these schemes either require a large amount of off-line training or need to be retrained when encountering a new program.

This paper presents a novel approach to reducing simulation time that seeks to address these issues. We use machine learning to build a predictor for the mean of any metric for a suite of programs, rather than just one benchmark at a time. The training requirements of our model are reduced to a minimum by automatically identifying representative programs; those whose *behaviour* in the design space can accurately represent the whole benchmark suite. We capture the behaviour using only a few simulations in the microarchitectural design space. This is fundamentally different from previous work [1] which focused on *characterising* programs independently of the space. By considering program behaviour in the design space, we can efficiently and accurately predict the average behaviour of a whole suite of programs.

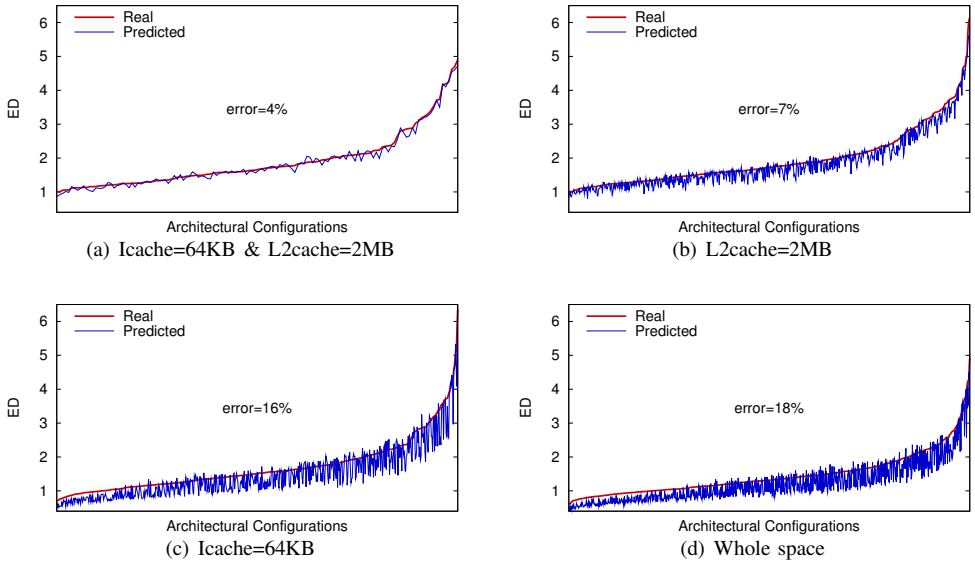
In comparison with two state-of-the-art techniques [2], [9], our predictor achieves the same level of accuracy, but requires five times fewer training simulations. Asymptotically, for a large number of programs, our approach needs an order of magnitude fewer simulations than these schemes.

The rest of this paper is structured as follows. Section II motivates the need to consider program behaviour on the metric of interest when selecting representative benchmarks. Section III then gives an overview of our model. Section IV describes our experimental setup and evaluates the optimum parameters for our model. Section V compares our predictor with state-of-the-art approaches. Finally section VI discusses related work and section VII concludes.

II. Motivation

Before describing our technique, we wish to show why previous approaches fail under certain circumstances and are therefore not suitable for microarchitecture design space exploration. These previous techniques characterise programs using microarchitecturally-independent features. We consider the features developed by Eeckhout et al. [1], which were used to perform benchmark subsetting and predict the performance of different systems and architectures [9]. We have used these techniques for microarchitecture design space exploration, as described in section V-A.

978-1-4244-5028-2/09/$25.00 ©2009 IEEE



(a) Icache=64KB & L2cache=2MB (b) L2cache=2MB

(c) Icache=64KB (d) Whole space

Fig. 1. Using microarchitecturally-independent features to characterise programs for design space exploration works well in (a) because they contain information about the parameters that are varied. However, there is no information that relates to the instruction cache and L2 cache sizes, so when they are varied in (b), (c) and (d) the prediction accuracy gets progressively worse.

Figure 1 shows the results of using these microarchitecturally-independent features for performance prediction for our design space described in section IV. The results are shown for the energy-delay (ED) metric averaged across the whole of SPEC CPU 2000. In the first case, figure 1(a), we vary all microarchitectural parameters but fix the size of the instruction cache and L2 cache to be 64KB and 2MB respectively. Figures 1(b) and 1(c) then show the same space where one of the cache sizes is varied but the other remains fixed. Finally, figure 1(d) shows the whole space where all parameters are varied. We show the real space and the predictions made by the model. The model's prediction error is also shown in the centre of each figure.

As can be seen in figure 1(a), this approach has good accuracy when the instruction and L2 cache sizes are fixed. The predictions from this scheme accurately track the real ED space. However, as figures 1(b) and 1(c) show, as the cache sizes are allowed to vary, the error rate of the technique increases from 4% to 7% or 16%. If both sizes can be altered, the error rate rises to 18%, meaning that it is very difficult to distinguish good configurations from bad.

The reason that this technique fails when the cache sizes are allowed to vary is because the microarchitecturally-independent features do not capture any information about the instruction or L2 cache usage. As seen in figure 1, when these two parameters are not included in the design space (figure 1(a)), this approach works well. However, when they are allowed to vary (figures 1(b), 1(c), 1(d)), the model becomes inaccurate because it has no knowledge about how program behaviour changes due to these parameters. Furthermore, if we were to add extra features, there would still be architectural parameters whose effects could not be captured. In other words, it is nearly impossible to find a finite set of features that would work across every possible design space. This claim is fundamental to our paper. Our approach performs better than existing techniques because we characterise programs directly in the design space we are considering.

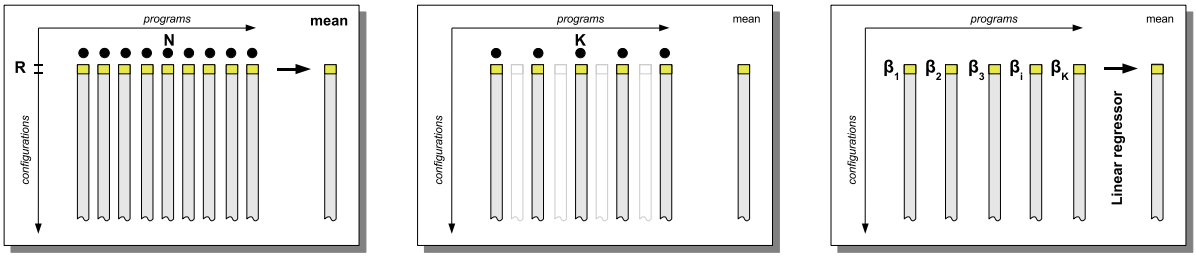
In this paper we propose a different solution. Instead of characterising programs, we directly consider their behaviour in the microarchitecture design space. This ensures that similarities between programs are captured no matter which parameters we are altering. In addition, we focus on predicting the average behaviour of the whole benchmark suite rather than individual performance of each program. This ensures that the number of simulations can be reduced to a minimum, allowing the designer to identify interesting design points quickly during early-stage design space exploration. The next section describes how we build this model which we then evaluate in section V.

III. Predictive Models for Early-Stage Design

This section presents our predictive model that directly predicts the mean of any metric (e.g. cycles or energy) within a design space for a whole suite of programs. It differs from other, recently proposed schemes [2], [3], [4], [5], [7], [8] that only model the design space of one program at a time. In addition, it reduces its training requirements by choosing only the most representative benchmarks based on their behaviour for the metric to predict. The fundamental premise of our approach is that a small sample of the design space can be used to capture the behaviour of each program.

A. Overview

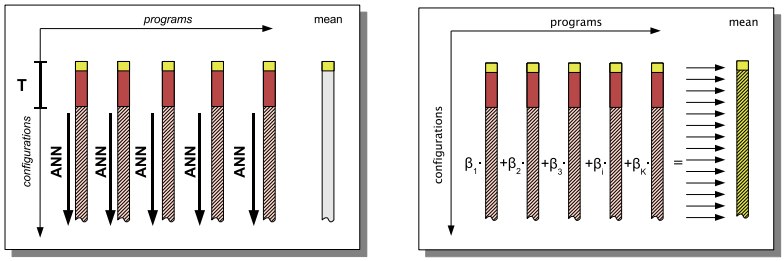
Our predictor is built in five different stages, as shown in figure 2. We first simulate R randomly selected configurations for each benchmark within the suite (typically R=32). This is shown in figure 2(a). From these we can trivially calculate the mean for the metric of interest. The mean can be



(a) Mean calculated from R configurations

(b) Selection of K most representative programs using a greedy algorithm

(c) Linear combination of K programs to approximate the mean, estimating weights β



(d) Predictors estimate the rest of the space

(e) Predicting the mean using the weights β.

Fig. 2. Overview of our predictor. We first run R randomly-selected simulations for each program and calculate the mean (a). Then we select K programs to represent the whole suite, reducing our training requirements (b). We find a linear combination of these K programs to the mean (c). We then simulate T configurations and build artificial neural networks (ANN) (d). Using the linear combination we predict the mean for any point in the design space (e).

the arithmetic, geometric, harmonic, or any other statistical property, such as standard deviation.

We then select K representative programs from the suite, typically K=5, using a greedy algorithm (figure 2(b)). A weighted sum of their execution times can be used to approximate the mean of the whole suite. Since we know the real mean for the R configurations, we can construct a linear function that approximates the execution time of the N programs as a combination of K, where K<N (figure 2(c)).

Next, as shown in figure 2(d) we build a predictor based on an artificial neural network (ANN) for each of the K remaining programs using T randomly-selected training simulations, typically T=512. Finally, we use these predictors to predict the metric of interest for all remaining configurations, combining them using the mapping to give the mean of any configuration in the design space, as shown on figure 2(e). The following sections describe this process in more detail.

B. Computing The Mean

The first step consists of selecting a small number of configurations from the design space, R, using uniform random sampling. We then simulate all programs on these configurations. This allows us to compute the mean for each of the R configurations (figure 2(a)). Our model can be used to predict any type of mean required by the user. In this paper we focus on the geometric mean, defined as $\mu_g = \sqrt[N]{\prod_i^N x_i}$, where N is the number of benchmarks in the suite and x_i is the metric for benchmark i.

C. Choosing K Relevant Programs

We assume that we can predict a whole suite of benchmarks using only K representative programs. We use the R configurations we have already simulated to choose these programs, so choosing K does not imply more simulations.

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

Fig. 3. The greedy algorithm that selects the K most representative programs from the whole benchmark suite.

We wish to retain only the K programs that best represent the space we want to predict (figure 2(b)). To do this we use a greedy algorithm, shown in figure 3. This algorithm chooses benchmarks based on the behaviour of each program in the design space we are considering. It assumes that if we know the optimal set of K+1 programs we can remove the benchmark that leads to the smallest prediction error to obtain the optimal set of K programs.

The actual choice of K depends on the number of programs in the benchmark suite and the prediction error that the user requires. It is evaluated in section IV-E.

D. Mapping K Programs To The Mean

In predicting the mean, we must account for the programs not retained within K. We do this by learning a linear mapping between the K representative programs and the mean for the R configurations for which we have an exact value (section III-B). This is shown in figure 2(c).

Given R values of the performance metric for each of the K programs, we want to build a linear estimator $\hat{\mu}$ of μ which uses K rather than all N (K $<$ N) values to estimate

TABLE I

MICROARCHITECTURAL DESIGN PARAMETERS WITH THE RANGE, STEPS AND THE NUMBER OF DIFFERENT VALUES THEY CAN TAKE.

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	$8, 16, 24, 32$	4	16
IL1 size	$8K \rightarrow 128K : 2*$	5	32K
DL1 size	$8K \rightarrow 128K : 2*$	5	32K
L2 size	$256K \rightarrow 4M : 2*$	5	2M
Total		63bn	

the mean. Since the geometric mean is in product form, we use a simple change of variables to allow linear modelling i.e. $ln(\hat{\mu}^N) = \sum_i^K \beta_i \cdot ln(x_i)$, where β_i is the weight for benchmark i and x_i is the metric for benchmark i.

If all the information is available to the model (K=N), all the weights (β_i) have a value of 1 and the estimation is perfect i.e. $\hat{\mu} = \mu$. When the number of programs, K, used to estimate the weights β_i is smaller than N, then the weights β_i, are updated to account for the removed programs.

E. Artificial Neural Networks

The final step in building our predictor enables us to predict any point in the microarchitectural design space. To do this we create a predictor for each of the K representative programs found in section III-C. We then combine them using the mapping described in section III-D.

The predictors are shown in figure 2(d) and are based on a state-of-the-art microarchitecture performance predictor [2]. The model is a multi-layer perceptron (i.e. artificial neural network) with one hidden layer of 10 neurons. It is trained with back-propagation. The activation function of the neurons in the input and hidden layers is a sigmoid function whereas the linear function is used for the output.

These predictors allow us to predict the entire design space of each of the K representative programs using only T randomly-selected training simulations from each. The predictions can then be combined using the weighted sum to estimate the mean for the whole suite of programs, as shown in figure 2(e).

IV. EXPERIMENTAL SETUP

This section describes our microarchitectural design space, simulation environment and the metric we used to evaluate our predictor throughout this paper.

A. Benchmarks And Simulator

We used the entire SPEC CPU 2000 benchmark suitecompiled with the highest optimisation level. To represent each program accurately we used SimPoint [10] using an interval size of 10 million instructions and a maximum of 30 clusters per program. We ran our experiments using the *reference* input set, warming the cache and branch predictor for 10 million instructions before performing detailed simulation. In section V-E we also include the MiBench [11] benchmark suite. These programs are compiled with the highest optimisation level and run to completion with the *small* input set. Here we used all programs except *ghostscript* which wouldn't compile and run correctly.

Our cycle-accurate simulator is based on Wattch [12], an extension to SimpleScalar [13]. We used Cacti [14] to model the energy and access latencies of the microarchitectural components accurately to make our simulations as realistic as possible.

We used cycles as a metric for program performance and the energy (in nJ) gained from Cacti and Wattch. We also evaluated the energy-delay (ED) and energy-delay-squared (EDD) products, which allowed us to explore the trade-offs between energy consumption and performance.

TABLE II

MICROARCHITECTURAL DESIGN PARAMETERS THAT WERE NOT EXPLICITLY VARIED, EITHER REMAINING CONSTANT OR VARYING ACCORDING TO THE WIDTH OF THE MACHINE.

(a) Constant

Parameter	Configuration
BTB assoc.	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

(b) Related to width

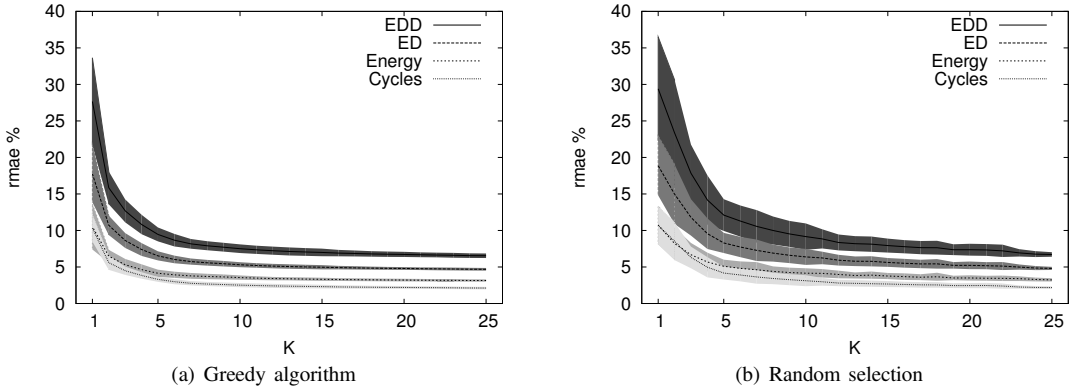
Parameter	Number			
Machine width	2	4	6	8
IntALUs	2	4	5	6
IntMuls	1	2	2	3
FPALUs	1	2	3	4
FPMulDiv	1	1	2	2

B. Microarchitecture Design Space

Our microarchitectural design space contains 63 billion different configurations, created by varying 13 different parameters within the simulator. These are shown in table I. They are similar to those other researchers have investigated [2], [4], allowing meaningful comparisons with previous work. The left-hand column describes the parameter and the second column gives the range of values the parameter can take. Also shown is a step size between the minimum and maximum values. The third column enumerates the number of different possible values. In the right-hand column is the baseline configuration, discussed in section IV-C.

Although our design space is 63 billion points in total, some of them do not make architectural sense. For example, the reorder buffer should not be smaller than the issue queue or load/store queue. We removed these configurations and thus reduced the total design space to 18 billion points.

Some parameters within the processor core remained constant in all simulations and these are described in table II(a). The functional units varied with the width of the pipeline are shown in table II(b). The predictive model that we develop in this paper is independent of the microarchitectural design



(a) Greedy algorithm (b) Random selection

Fig. 4. Error and its standard deviation (in the shaded areas) as a function of K, the number of representative programs needed to build the linear model. Our greedy algorithm performs better than random selection in terms of error for small values of K, and has a much lower standard deviation overall.

space under consideration. Therefore, we could easily add additional parameters (e.g. core frequency) without any loss of accuracy in our approach.

C. Choice Of A Baseline

In order to give each program within the benchmark suite equal importance in the calculation of the mean, we normalise to a baseline configuration. However the choice of baseline does not affect the accuracy of our predictor.

We chose a baseline configuration similar to the Intel Core microarchitecture, parameters for which are shown in table I. It is a balanced microarchitectural configuration, lying comfortably within the top 10% of this space for cycles, ED and EDD, and within the top 20% for energy.

D. Evaluation Methodology

The *relative mean absolute error* (rmae) defined as $\frac{1}{N}\sum_i^N \left|\frac{predicted\ value_i - real\ value_i}{real\ value_i}\right|$ is used to evaluate our models. It tells us how much error there is between the prediction and the actual value.

All predictors are trained using randomly selected configurations from our design space and validated on a further randomly selected 2500. Since the selection of the training simulations was performed randomly, each experiment was repeated 20 times, using different random simulations each time. The results presented are thus an average and whenever possible the standard deviation is shown alongside.

E. Evaluating Optimum Model Parameters

Ideally we would like to examine the the impact of the parameters present in our model on prediction accuracy. However, due to space constraints this can only be shown for K. As in our prior work [7], we have found that R=32 and T=512 gives robust results for predicting all SPEC CPU 2000 benchmarks in this space.

Varying K Representative Programs: Having randomly selected 32 configurations, R, for each program and learned a linear mapping to the mean, we then need to select K representative programs. This is done by using our greedy algorithm (figure 4(a)). We also compare against a straightforward random approach (figure 4(b)), for varying values of K. We built ANNs for each of the K programs using 512 randomly selected configurations for each benchmark (T) and verified as described in section IV-D. These graphs show that the greedy algorithm has a lower standard deviation than using a random selection for K and that there is not a significant decrease in error with our scheme after K=5.

V. Comparison With State-Of-The-Art

In this section, we conduct a comparison of our technique against two state-of-the-art approaches. We show that our scheme achieves a lower error than the other techniques and that as the number of benchmarks rises, we require, asymptotically, an order of magnitude fewer training simulations to achieve the same error.

A. Features-Based Predictor

The first prediction technique we chose to compare with selectively reduces the number of training programs in a benchmark suite. Eeckhout et al. [1] describe a method to select the most representative subset of benchmarks using microarchitecturally-independent program features, unlike our scheme that uses the actual execution time or energy consumption of different configurations. This approach has recently been applied to performance prediction across different computing systems using offline training [9]. We adapted and extended their technique to our problem of predicting the mean performance. We extracted the program features using the authors' own tool and selected the most representative subset of benchmarks from the whole suite. We then constructed artificial neural networks (using T=512) for each of these programs, combining the predictions to get the mean using weights obtained from clustering. While this new approach is different from the scheme the authors proposed, it is based on their original idea of characterising programs using architecture-independent features.

B. Single-Program Predictors

The second state-of-the-art technique consists of simply constructing a single-program predictor for each benchmark. We chose to use the scheme proposed by İpek et al. [2], although we could have used any other related approach [3], [4], [5], [15]. Since this technique predicts the space of each individual program, we simply average the prediction of the individual models to compare with our scheme.

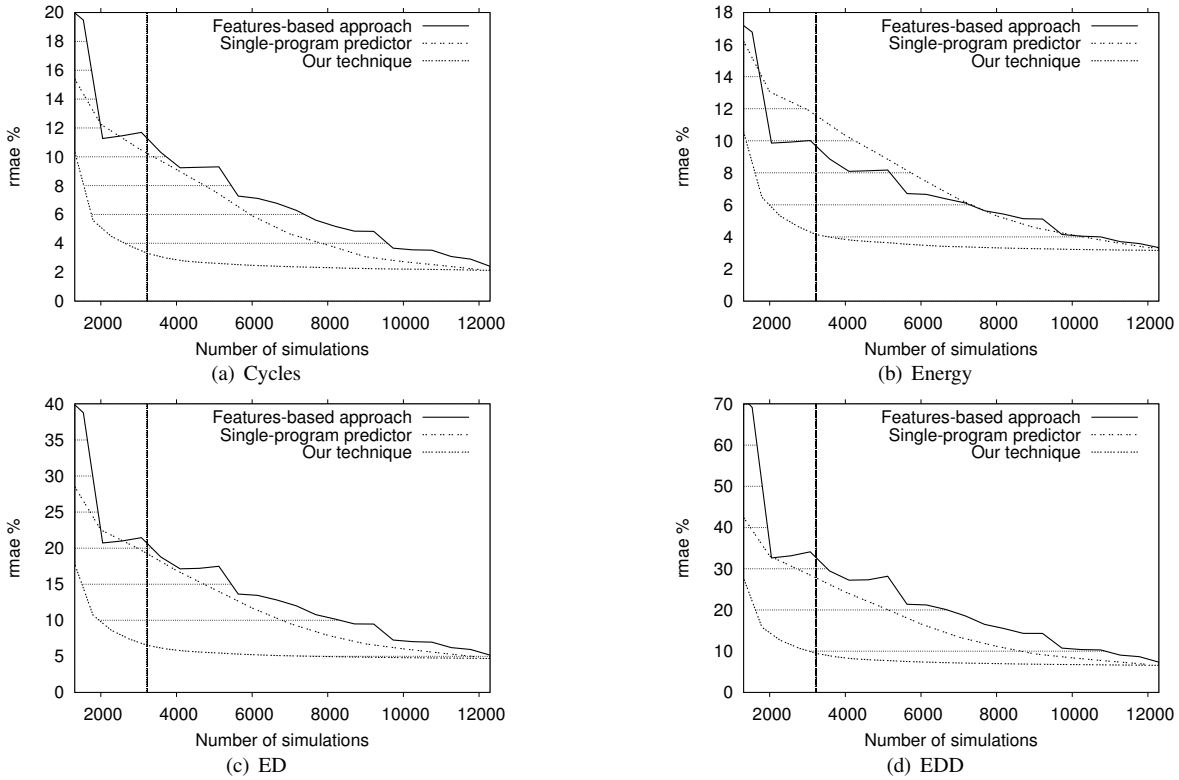


Fig. 5. Error as a function of the number of simulations to train both state-of-the-art schemes and our technique. Our approach achieves the same or better error as the state-of-the-art schemes with half the number of training simulations or fewer.

C. Training Costs

In order to simplify the comparison between different models, we define a training budget in terms of the number of simulations allowed for training. While existing approaches have provided an error estimation mechanism [2], we decided not to use it to simplify the comparison. Note that this error estimation mechanism is not specific to the single-program predictor technique we compare against and could be applied equally to our technique or to the features-based approach, since they are both built on top of single-program predictors.

In our model, all programs require 32 (R) simulations to select K representative programs and learn a linear mapping. The chosen K programs receive the remaining budget. In the features-based model, one simulation is required to extract features, then the remaining simulations are assigned to the chosen benchmarks. For the single-program predictor, the entire training budget is shared equally between all programs.

For our approach, for the 26 SPEC CPU 2000 benchmarks, the cost for our predictor, as discussed in section III-E, when we pick K=5 representative programs, is $26{\cdot}32{+}5{\cdot}(512{-}32) = 3232$ simulations. For the features-based approach, only one simulation is needed per benchmark to extract program features. The rest of the budget is then distributed evenly across the chosen benchmarks. Following our example, with a budget of 3232 simulations, 6 programs are kept ($3232/512 \cong 6$). For the single-program predictor, each benchmark receives exactly the same proportion of the simulation budget. So for 3232 training simulations, each single-program predictor is trained with 124 simulations.

Note that the selection of the training programs for each technique is independent of the number of SimPoints required for each benchmark. On average, programs with many SimPoints and those with few are equally likely to be selected. This was verified in our experiments. Therefore, the total simulation time for each technique is proportional to the total number of simulations.

D. Predicting The Whole SPEC Suite

This section shows how our technique compares to the state-of-the-art schemes when predicting the whole of the SPEC CPU 2000 benchmark suite across the configuration space. Figure 5 shows the error for each metric as we vary the total number of training simulations. As can be seen, our predictor achieves the same error as the features-based approach and the single-program predictor using many fewer simulations. For example, when predicting ED we achieve an error of 6% using 3232 simulations, whereas the two state-of-the-art approaches require 10,000.

Furthermore, for any simulation budget, our predictor always achieves a lower error than the other schemes. As already seen, given 3232 simulations (shown in figure 5 with a vertical bar) our predictor achieves an error of 6% for ED, whereas the two others approaches have an error of 18%; three times higher. The same conclusion can be drawn for the other metrics. It is interesting to note that the error rate of the other schemes are different from those previously reported [2], [4]. This is explained by the fact that our space is different and, more fundamentally, because the simulation budget is shared across all programs (each receiving a fraction of it).

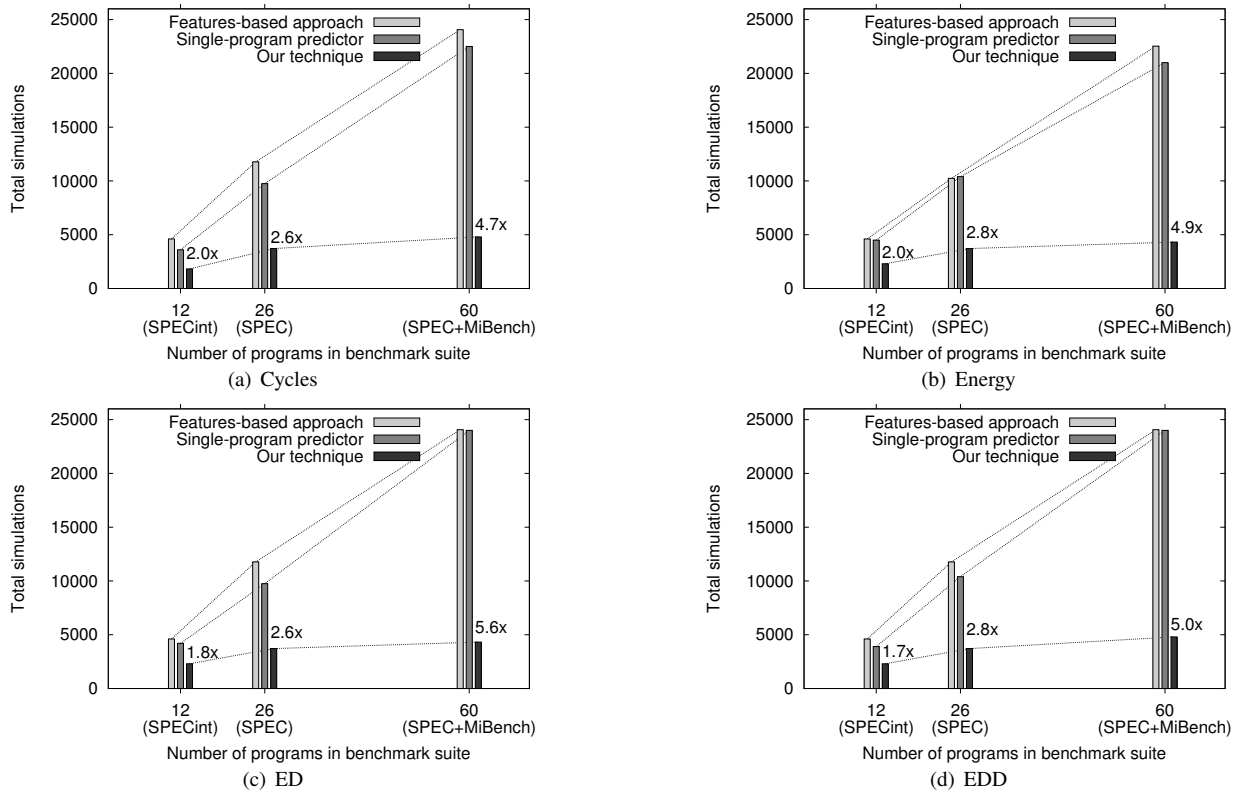


Fig. 6. Total simulations needed to train each model to reach a fixed error when different benchmark suites are considered. The lines represent the trend and the labels at the top of each bar for our scheme represent the savings compared to the best of the other approaches. As the number of programs in the benchmark suite increases, all schemes require a greater number of training simulations. However, our approach needs an order of magnitude fewer additional simulations than the other approaches as the benchmark suite size increases. Moving from 26 to 60 programs, we require fewer than 1000 new simulations whereas both other schemes require a further 10,000.

E. Larger Benchmark Suites

We now consider how the number of training simulations varies as we predict different benchmark suites, keeping a fixed error rate. We set the error rates at those achieved when we set K=5 for our model, as described in section V-C. That is 3232 simulations which corresponds to an error of 3% for cycles, 4% for energy, 6% for ED and 8% for EDD. It is straightforward to fix the budget and pick other error rates. We chose to predict for SPEC CPU 2000 Integer (12 programs), SPEC CPU 2000 (26 programs in total) and SPEC CPU 2000 combined with MiBench (60 programs in total). Figure 6 shows the results using our approach and the state-of-the-art predictors.

As can be seen, for small benchmark suites, such as SPECint, all three predictors require a similar number of training simulations to achieve the fixed error rate, although ours always requires fewer than the other two approaches. However, for larger benchmark suites, the benefits of our scheme become clear. When predicting cycles for the whole of SPEC, the single-program predictor and features-based approach need 9750 and 11,776 respectively, whereas our scheme requires just 3712, 2.6 times fewer. When predicting for combined SPEC and MiBench, we require 4800 simulations compared with 22,500 and 24,064, which is 4.7 times fewer. As the curves in these graphs demonstrate, when moving from 26 to 60 programs, both other schemes require a further 10,000 new simulations whereas our predictor requires fewer than 1000. This is asymptotically an order of magnitude fewer simulations than the state-of-the-art approaches as the benchmark suite size increases.

VI. Related Work

This paper has proposed a scheme to predict the mean of a suite of programs for a given metric. Recently there has been significant interest in predicting the performance of a *single* program across a microarchitectural design space. Schemes include linear regressors [3], artificial neural networks [2], [16], radial basis functions [15], [17] and spline functions [4], [5]. These models obtain similar accuracy to each other [6]. However, they require a significant number of training simulations when predicting a whole benchmark suite, since a new model must be built for every program.

Other work has focused on clustering benchmarks based on program features [1], [18], then using this offline knowledge to predict the best architecture for a new program [9]. We compare with this approach in this paper and find that we can achieve the same error rate but with five times fewer training simulations. Recently, researchers have proposed schemes that use reactions [8] or a signature [7] to characterise a new program based on previously-seen benchmarks. However, the offline training can result in a significant simulation cost and these schemes still only predict for a single program, rather than for the whole benchmark suite.

A common and widely accepted method of reducing simulation time is to use profile-based sampling such as SimPoint [10]. This picks only a small number of instructions from each program to simulate by grouping them into clusters that represent the entire program using a k-means clustering algorithm. As the number of instructions contained in the clusters is significantly fewer than the total number, simulation time can be reduced by several orders of magnitude. We used SimPoint to reduce the simulation time for our experiments. Statistical sampling techniques, such as SMARTS [19], also make savings by executing only a small fraction of the program using detailed simulation.

Statistical simulation [20], [21], [22], [23], [24] is similar to sampling since only a portion of the program is executed. However, instructions are executed only symbolically within a modified simulator using a statistical model. A similar idea consists in extracting micro-benchmarks from the real program and simulating these in order to reduce simulation time [25]. Whilst all these techniques offer reduction in simulation time, they require extraction of program characteristics. Since these characteristics depend on the microarchitecture, it makes it difficult to adapt to any major microarchitectural changes. In fact, new program features might need to be extracted, which again assumes prior knowledge of the architecture design space.

Finally, analytic models have been proposed as a way to reduce simulation cost whilst maintaining good accuracy [26], [27]. However these approaches require an in-depth knowledge of the architecture and need to be built by hand. Furthermore, when any major change is made to the microarchitecture, these models need to be adapted and tuned again [28]. On the other hand, our technique focuses on building architectural models automatically without any prior knowledge about the microarchitecture. As such it can easily adapt to future microarchitecture designs.

VII. Conclusions

This paper has proposed a novel approach to design space exploration by predicting the mean of a suite of programs. We have shown that we can reduce the number of simulations required by our model by training only on the most representative programs from the benchmark suite. We pick these using a greedy algorithm based on program behaviour for the metric we wish to predict. Using only 5 representative programs from SPEC CPU 2000, we are able to accurately model the average behaviour of the full suite.

Furthermore, we have compared our technique with two state-of-the-art predictors and shown that we can achieve the same error rate with five times fewer training simulations on the whole of the SPEC CPU 2000 and MiBench suites.

Acknowledgements

This work has been partially supported by Milepost, the Royal Academy of Engineering and EPSRC. It has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF) which is partially supported by the eDIKT initiative.

References

[1] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Workload design: Selecting representative program-input pairs," in *PACT*, 2002.
[2] E. İpek, S. A. McKee, R. Caruana, B. R. De Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *ASPLOS-XII*, 2006.
[3] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *HPCA-12*, February 2006.
[4] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS-XII*, 2006.
[5] B. C. Lee and D. Brooks, "Illustrative design space studies with microarchitectural regression models," in *HPCA-13*, 2007.
[6] B. C. Lee, D. M. Brooks, B. R. De Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *PPoPP-12*, 2007.
[7] C. Dubach, T. M. Jones, and M. F. P. O'Boyle, "Microarchitectural design space exploration using an architecture-centric approach," in *MICRO*, 2007.
[8] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, "Using predictive modeling for cross-program design space exploration in multicore systems," in *PACT*, 2007.
[9] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *PACT*, 2006.
[10] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS-X*, 2002.
[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *WWC-4 (MICRO-34)*, 2001.
[12] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *ISCA-27*, 2000.
[13] T. Austin, "The SimpleScalar Toolset," http://www.simplescalar.com.
[14] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "Cacti 4.0," HP Laboratories Palo Alto, Tech. Rep. HPL-2006-86, 2006.
[15] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictve performance model for superscalar processors," in *MICRO-39*, 2006.
[16] E. İpek, B. R. De Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *Euro-Par*, 2005.
[17] K. Vaswani, M. J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph, "Microarchitecture sensitive empirical models for compiler optimizations," in *CGO*, 2007.
[18] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *ISCA-34*, 2007.
[19] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *ISCA-30*, 2003.
[20] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John, "Control flow modeling in statistical simulation for accurate and efficient processor design studies," in *ISCA-31*, 2004.
[21] A. Joshi, J. Yi, R. Bell Jr., L. Eeckhout, L. John, and D. Lilja, "Evaluating the efficacy of statistical simulation for design space exploration," in *ISPASS*, 2006.
[22] M. Oskin, F. T. Chong, and M. Farrens, "HLS: Combining statistical and symbolic simulation to guide microprocessor designs," in *ISCA-27*, 2000.
[23] R. Rao, M. Oskin, and F. T. Chong, "HLSpower: Hybrid statistical modeling of the superscalar power-performance design space," in *HiPC*, 2002.
[24] S. Eyerman, L. Eeckhout, and K. De Bosschere, "Efficient design space exploration of high performance embedded out-of-order processors," in *DATE*, 2006.
[25] A. Joshi, L. Eeckhout, R. H. Bell, Jr., and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM TACO*, vol. 5, no. 2, 2008.
[26] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *ISCA-31*, 2004.
[27] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *MICRO-27*, 1994.
[28] D. Ofelt and J. L. Hennessy, "Efficient performance prediction for modern microprocessors," in *SIGMETRICS*, 2000.