Microprocessors and Microsystems 37 (2013) 41–51

Contents lists available at SciVerse ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Microarchitectural design space exploration made fast

Qi Guo [a,b,*], Tianshi Chen [a,c], Yunji Chen [a,c], Ling Li [a,c], Weiwu Hu [a,c]

[a] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[b] *Graduate University of Chinese Academy of Sciences, Beijing 100049, China*
[c] *Loongson Technologies Corporation Limited, Beijing 100190, China*

ARTICLE INFO

Article history:
Available online 10 August 2012

Keywords:
Design space exploration
Microarchitecture
Predictive models
Model tree

ABSTRACT

Predictive modeling is an emerging methodology for microarchitectural design space exploration. However, this method suffers from high costs to construct predictive models, especially when *unseen* programs are employed in performance evaluation. In this paper, we propose a fast predictive model-based approach for microarchitectural design space exploration. The key of our approach is utilizing inherent program characteristics as prior knowledge (in addition to microarchitectural configurations) to build a universal predictive model. Thus, no additional simulation is required for evaluating *new* programs on *new* configurations. Besides, due to employed *model tree* technique, we can provide insights of the design space for early design decisions. Experimental results demonstrate that our approach is comparable to previous approaches regarding their prediction accuracies of performance/energy. Meanwhile, the training time of our approach achieves 7.6–11.8× speedup over previous approaches for each workload. Moreover, the training costs of our approach can be further reduced via instrumentation technique.

1. Introduction

An important task in the architecture-level design of microprocessor is choosing the optimal configuration from the large number of potential candidate microarchitectural configurations for specific workloads. Currently, computer architects and researchers rely heavily on large-scale simulations to explore such huge design spaces. Unfortunately, as a consequence of the low simulation speed, it is intractable to exercise the entire exponentially large design space, which forces designers only simulate a small subset of design options to decide appropriate microarchitectural configuration parameters. To reduce the simulation costs to the full extent, many fast simulation approaches have been proposed, such as statistical sampling [1], statistical simulation [2–4], workloads subsetting [5,6] and so on. However, although these techniques play important roles in reducing the universal simulation time, given the exponentially large design space to explore, the simulation costs remain large. To avoid ineffective yet slow simulation, predictive modeling has received many attentions recently. In general, a predictive modeling approach consists of two phases, training phase and predicting phase. In the training phase, the mapping from design parameters to performance/energy of microprocessor was learned via machine learning techniques based on simulation results, and trained model is called *predictive model*. In the predicting phase, the performance/energy of microprocessor with respect to a given design configuration is predicted by the predictive model, and no simulation is required. In this way, compared with previous techniques, predictive modeling can reduce even more simulation time since simulations are only required in the training phase.

Traditionally, the predictive modeling approaches have to spend specific costly cycle-accurate simulations for *each* workload, so as to rebuild associated predictive models to maintain high accuracy [7–13] (program-based approaches), or attain some reactions (e.g., instruction per cycle) to provide so-called "signature" [14,15] (reaction-based approaches). These approaches treated the mapping from design parameters to performance/energy of microprocessor as a black-box, and no prior knowledge about workloads was directly utilized. Consequently, each workload has to consume specific simulation costs, thus the ever increasing number of complicated workloads would significantly increase the overall simulation costs of predictive modeling, which has been greatly overlooked by previous approaches. In practice, this drawback will be easily disclosed, since industrial general purpose microprocessors should adapt to a large number of workloads from different domains (e.g., high performance computing, media processing, etc.) [16]. During the design of Godson-3 [17], more than 200 workloads including SPEC CPU2000, Linpack and MiBench, etc. have been evaluated. To meet the requirement of practical evaluations, an efficient predictive modeling approach should adapt to different workloads by consuming least costs.

In this paper, we propose a novel approach that explicitly incorporates inherent program characteristics for training predictive

* Corresponding author.
E-mail address: guoqi@ict.ac.cn (Q. Guo).

0141-9331/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2012.07.006

models. To be specific, the program characteristics and architectural configurations constitute the whole training data, which can be viewed as two explicit and orthogonal knowledge sources to model the relation between the program–architecture interaction and its corresponding responses. The key intuition is that, given some configurations, when a new workload has similar characteristics as some representative workloads involved in the training data, the corresponding reactions are also similar to those of representative workloads. Since the specific cycle-accurate simulations for rebuilding models or attaining signature of a new program is not required by our approach, the overall simulation costs can be significantly reduced. Fig. 1 illustrates the basic idea of our approach, and we can clearly see that our approach can reduce the number of simulations from $M \times N$ to only $m \times n$. Consequently, our approach can efficiently adapt to the large amount of workloads involved in industrial evaluations. Besides, we employ *model tree* [18] to deal with the complicated interactive design parameters in our application [9]. In this way, we can obtain an interpretable predictive model to further facilitate design space exploration and optimization.

The main contributions of our work are summarized as follows. First, instead of treating the parameter-performance/energy mapping as a black-box, our approach explicitly incorporates inherent program characteristics (instead of reactions on typical architectures) when learning the mapping, thus the simulation costs spent by the training phase are significantly reduced. As a result, our approach is much faster than (and, as effective as) previous approaches, which is validated by our experiments. Second, the model tree technique employed by our approach can quantitatively provide priority and weight of individual design parameter to performance and energy, which can help architects focus available budgets on most important issues to achieve early design decisions.

The paper proceeds as follows. Section 2 describes the evaluation methodology for the following experiments in our work. Section 3 provides the detailed mechanism of our approach with preliminary experiments to demonstrate the effectiveness of learning engines and proposed program characteristics. Section 4 demonstrates the primary experiments that implement the cross-program prediction of performance and energy with low average error. Section 5 compares the training costs of our approach with previous approaches. Section 6 illustrates the effectiveness of our approach for early design decisions by giving insights of the design space. Section 7 details related work of simulation-based design space exploration. Finally, Section 8 concludes the whole paper.

2. Evaluation methodology

2.1. Tool, simulator and benchmarks

The efficiency of extracting program characteristics is decisive to the successful application of our approach. To reduce the training costs, we implement a fast characteristics extraction tool through modifying Simplescalar [19] to obtain program characteristics, which avoids detailed cycle-accurate simulations for each workload. Moreover, our approach can further reduce the training costs via instrumentation technique on a real machine adopting the same instruction set architecture (ISA), which will be detailed in Section 5.2.

To evaluate the performance and energy for each configuration, we employ Wattch [20], which is a tool providing a power and delay evaluating methodology within the Simplescalar framework, as our simulator. Meanwhile, we employ SPEC CPU2000 as the representative benchmarks to validate our approach, and they are compiled with -O3 option. Moreover, due to the costly simulation, we employ full *test* input set to obtain a validation set with reasonable size.

2.2. Metrics

We use two metrics to represent the processor responses of certain program and architecture design options. The first metric is most widely used in performance evaluation field as *instruction per cycle* (IPC). Moreover, except performance, energy is also a great concern today. Thus, the second metric we use is so-called *power consumption per cycle* (PPC) that is reasonable for evaluating the model of energy.

To evaluate the accuracy of our models (including performance and energy models), other two metrics are also employed. The first one is widely used *mean relative error* (MRE), i.e., $\left(\sum_i^N \left|\frac{p_i - a_i}{a_i}\right|\right)/N$, where p_i and a_i are the predicted and actual value of the response (IPC or PPC) for ith instance and N is the number of total instances. However, to better evaluate the fitness of learned regression model, we employ another metric, *correlation coefficient*, which measures the extension of linear relationship between predicted value and actual value. The correlation coefficient ranges from -1 to 1, where 1 corresponds to the ideal correlation, that is, predicted value perfectly models the shape of the real value.

3. Proposed mechanism

In this section, we first introduce the overall framework of our approach. Then, we introduce employed model tree technique to

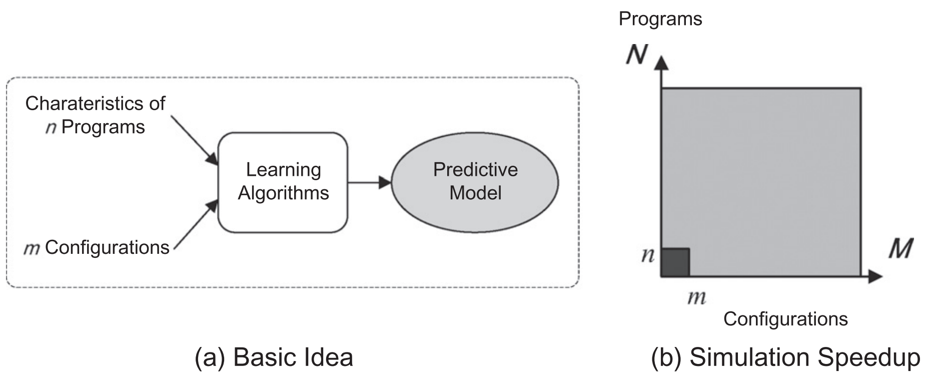


Fig. 1. (a) Illustrates the basic idea of our approach, where both sampled microarchitectural configurations and characteristics of representative programs are served as inputs for learning algorithms. Thus, we can obtain a universal model that can predict the responses of *new* programs running on *new* configurations. (b) Illustrates that necessary cycle-accurate simulations for exploring the design space containing M configurations when evaluating N programs, where the black regions indicate the simulation subspaces and the gray regions indicate the predictable subspaces. It can be clearly seen that our approach only needs $m \times n$ simulations, which significantly reduces the simulation costs, especially when the number of evaluated benchmarks is quite large.

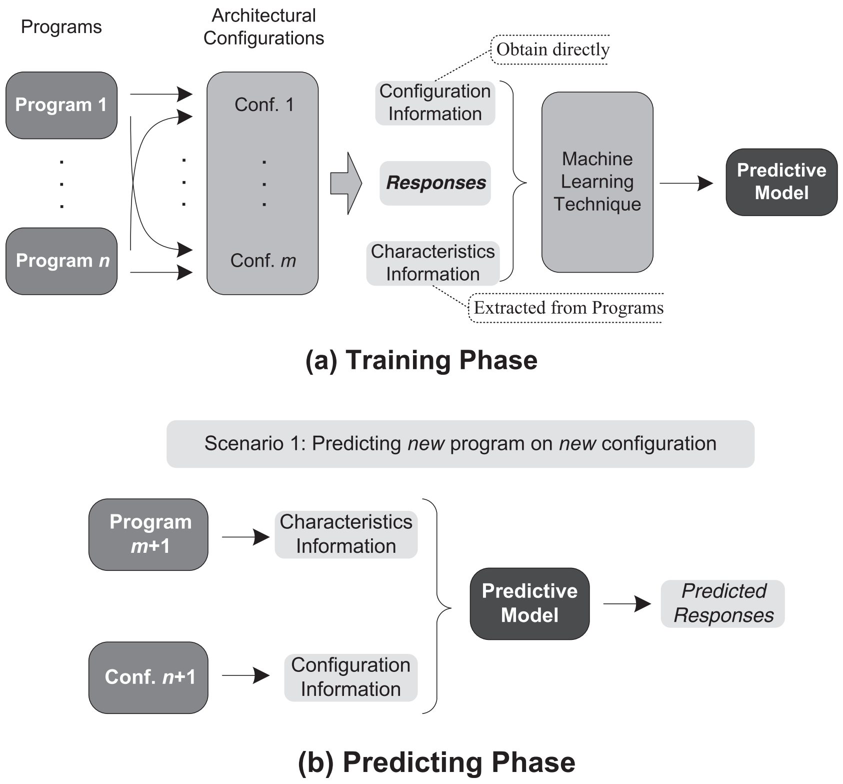

Programs
Architectural Configurations
Program 1
Program *n*
Conf. 1
Conf. *m*
Obtain directly
Configuration Information
Responses
Characteristics Information
Extracted from Programs
Machine Learning Technique
Predictive Model
(a) Training Phase
Scenario 1: Predicting *new* program on *new* configuration
Program *m*+1
Characteristics Information
Conf. *n*+1
Configuration Information
Predictive Model
Predicted Responses
(b) Predicting Phase

Fig. 2. Framework of our proposed methodology.

construct the predictive model. After that, we specify the program characteristics that potentially affect performance and energy, which are independent of concrete microarchitecture. Finally, we detail the training and predicting process of our predictive model.

3.1. Overview

Overall, our approach follows the conventional framework of predictive modeling technique. Nevertheless, the most important difference is that program characteristics are treated as prior knowledge to construct a more accurate and universal model as shown in Fig. 2. Specifically, the framework can be divided into two separate phases, *training* and *predicting* phase.

During the training phase, we first select m typical architectural configurations[1] and n representative programs as the original training set. Then, we have to simulate the execution of each program on each architectural configuration in the training set to collect the corresponding responses, that is, $m \times n$ simulations are necessary to obtain the training data. After that, involved program characteristics, architectural configurations and responses are served as inputs for machine learning engine to construct a predictive model. During the predicting phase, interested programs and architectural configurations are sent into the built predictive model to obtain predicted responses. Since both the program characteristics and sampled configurations are treated as the *independent variables* for learning, the correlation between them and the *target variables* (e.g., IPC) could be approximated by utilized learning algorithms. It is notable that our model can be applied to predict the responses of new programs on new architectures once the characteristics of such new programs are available, which is listed as Scenario 1 in Fig. 2. Therefore, we can rapidly obtain the responses for *new* programs on *new* architectural configurations without additional simulations for fast design space exploration.

[1] In our experiments, such architectural configurations are selected uniformly at random (UAR), as performed in [10]. Actually, some advanced sampling techniques, i.e., active learning [8], can be utilized to increase the diversity of sampled configurations.

3.2. Model tree based predictive modeling

3.2.1. Background

Although many machine learning techniques have been widely applied in the field of computer system, the sophisticated interaction between program and architecture still imposes a challenging requirement on the performance of existing learning algorithms. For instance, although the representation ability of artificial neural networks (ANNs) is rich enough to express any complex interactions among variables, choosing a suitable learning engine for evaluating each architectural option is still an open question. The first reason is that learning engines like ANNs always require abundant samples to train an accurate enough model, which may be unfeasible in practice due to the prohibitively slow simulation speed given such a large design space to explore. The second reason is that such intricate learning engines could achieve good performance while their models are hard to interpret that may not help designers focus on critical bottlenecks of the performance. Therefore, we seek help for learning algorithms that can quickly provide accurate and interpretable models.

Fortunately, model tree [18], which divides the input space into a tree structure and fits linear regression models on its leaves, can efficiently achieve better accuracy with appropriate training size. More importantly, it can provide an interpretable model to meet our requirements as stated. Besides, the most prominent feature of such a tree-based regression technique differs from other conventional competitors is that it splits the space into a number of different regions, through which the nature of the complicated diversity of the interaction between program and architecture is much more revealed.

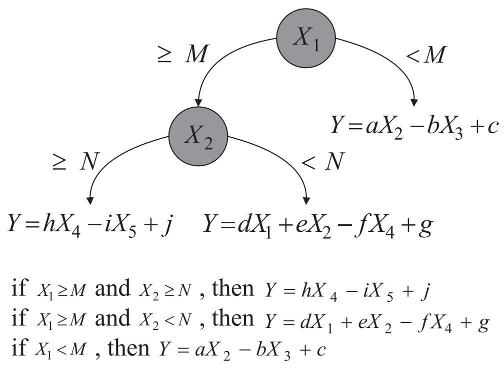

X_1
$\geq M$
$< M$
X_2
$Y = aX_2 - bX_3 + c$
$\geq N$
$< N$
$Y = hX_4 - iX_5 + j$
$Y = dX_1 + eX_2 - fX_4 + g$
if $X_1 \geq M$ and $X_2 \geq N$, then $Y = hX_4 - iX_5 + j$
if $X_1 \geq M$ and $X_2 < N$, then $Y = dX_1 + eX_2 - fX_4 + g$
if $X_1 < M$, then $Y = aX_2 - bX_3 + c$

Fig. 3. An illustrative example of model tree. *Y* is the processor responses (e.g., IPC or PPC) to predict and X_i is the independent variable (e.g., microarchitectural parameter and program characteristic) in our application.

Table 1
Microarchitectural design space we investigated with more than 70 M design options.

Abbreviation	Parameter	Value
WIDTH	Fetch/Issue/Commit width	2, 4, 6, 8
FUNIT	FPALU/FPMULT units	2, 4, 6, 8
IUINT	IALU/IMULT units	2, 4, 6, 8
L1IC	L1-ICache	8–256 KB: step 2×
L1DC	L1-DCache	8–256 KB: step 2×
L2UC	L2-UCache	256–4096 KB: step 2×
ROB	ROB size	16–256: step 16+
LSQ	LSQ size	8–128: step 8+
GSHARE	GShare size	1–32 K: step 2×
BTB	BTB size	512–4096: step 2×
Total no.		70,778,880

3.2.2. Tree construction

Model tree is a special category of *regression tree*. In contrast to classical regression trees, it incorporates linear regression functions into a conventional decision tree, which results in a compact regression tree model and better prediction accuracy. Actually, model tree is constructed by the divide-and-conquer method, which consists of three steps to obtain a generalizable model. The first step is to build the initial large tree via computing the standard deviation of the target value of cases in the training set. Then, the tree is pruned back to mitigate potential overfitting. Finally, a smoothing process is executed to compensate for the sharp discontinuities among adjacent linear models on the leaves of the pruned tree. Fig. 3 gives an illustrative example of the model learned by model tree algorithm, where a global non-linear relation is simulated via such a set of simple local linear models on the leaves.

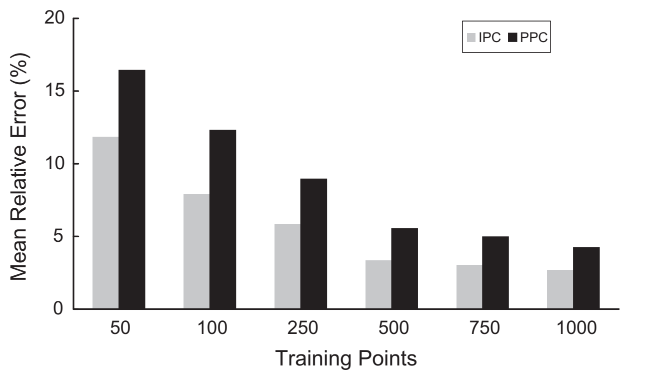
To barely evaluate the effectiveness of employed model tree for modeling the performance and energy, we conduct experiments as conventional program-based predictive modeling approaches always did as described in Section 1, that is, the model only characterizes the relation between design parameters and processor responses for each workload. Table 1 shows the design space we investigated in our study, where more than 70 M design options need to explore. Fig. 4 illustrates the training error and correlation coefficient along with the increasing size of sampled set of design points. We can see that after 500 design points are trained in investigated design space, the training error decreases much slowly than before. Therefore, by trading-off the simulation time and prediction accuracy, we decide to employ 500 design points as the training set in the following experiments.

3.3. Program characteristics

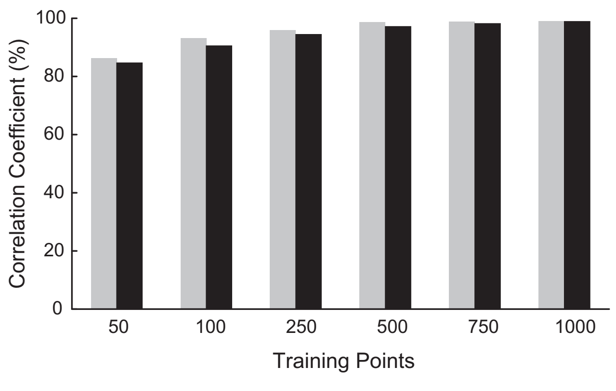
Hoste et al. have proposed several illustrative microarchitecture-independent characteristics in [21]. Inspired by their work, in this subsection, we pay more attentions on the inherent program characteristics that significantly affect performance and power consumption.

The common way to represent processor performance is to employ CPI (cycle per instruction) "stack" which breaks performance into a baseline CPI plus a number of individual miss events [22]. More precisely, the baseline performance of a program stems from the inherent parallelism (data dependency and hardware hazard). Meanwhile, individual miss events are composed of branch misprediction, data & instruction TLB miss, data & instruction L1 cache miss, and L2 cache miss. Therefore, we consider the program characteristics from stated aspects.

However, since power consumption is also a great concern in our work, we also try to delve the underlying relationship between program and the dynamic active power. By intuition, dynamic activity power is fundamentally a function of key system utilization statistics. According to Powell [23], at most nine architecture dependent predictors of run-time characteristics results are highly correlated to the activity factors. Such predictors are determined by the interaction between program and architecture, for example, IPC and SIPC (speculative-IPC), etc. However, in our approach, we only consider the microarchitecture-independent characteristics from the perspective of program. Therefore, considering the mentioned predictors, instruction mixture (such as load and store rate)



(a) Mean relative error of predicting IPC&PPC



(b) Correlation coefficient of predicting IPC&PPC

Fig. 4. Mean relative error and correlation coefficient of all SPEC CPU2000 programs along with the increasing training points. By trading-off the costly simulation time and prediction accuracy, we select 500 training points for the following experiments.

is the main program characteristic that may be related to the dynamic power consumption.

3.3.1. Inherent parallelism

The inherent parallelism determines the potential parallelism a machine can exploit. In modern typical superscalar microprocessors, hazards, which come from data hazard, structural hazard and control hazard, are the main factors to hamper the ideal performance from the speedup gained by pipelining. Therefore, the inherent instruction-level parallelism (ILP) of a program can be measured from three aspects as data dependency, instruction mix and branch behavior.

Data dependency. This category is concerned with the potential data hazard, and it is defined by the number of instructions between write and read of a register instance (Read-after-Write). Since the data dependency distance varies, we select metrics to express the distribution of the data dependency through dividing the percentage of different data dependency distance into eight categories: 1, 2, 4, 8, 16, 32, 64 and greater than 64.

Instruction mix. Another obstacle of fully exploiting potential ILP is the limited number and latency of functional units. Obviously, programs that are full of single-cycle instructions will be faster than those with many multi-cycle floating-point instructions. Moreover, instruction mix highly correlates to the dynamic activity of the microprocessor pipeline, so we consider it in a fine granularity by dividing the instruction mix into five categories, that is, integer computation rate, floating-point computation rate, load access rate, store access rate and branch instructions rate.

Branch behavior. Basic Block Size is the average distance between two consecutive branch instructions in the dynamic instruction stream of a program. And it is also decisive to the potential ILP a superscalar processor can exploit. *Fraction of Forward Branches* and *Fraction of Taken Branches* are also taken into consideration. Besides, considering the global history of branch behavior utilized by modern microprocessors, we also measure *Branch Transition Rate* as introduced in [24].

3.3.2. Locality behavior

As stated, important components of CPI stack are the miss events of caches and TLBs, both of which are based on the *principle of locality* (which includes temporal and spatial locality). Many researchers try to characterize locality behavior quantitatively and precisely [25,26] since it is intrinsic in the program structure but the exploitation by a specific machine can vary. According to Zhong et al. [27], the essential metric of locality is *reuse distance*, which is the number of distinct data elements accessed between the current and previous accesses to the same datum. Thus, we use this metric as the guideline to characterize the locality behavior of a program.

Data temporal locality. Reuse distance is the explicit measure of data temporal locality when each data element is a single memory address. However, such definition is too fine to characterize for tractability. Instead, we specify the data element as 16, 64, 256 and 4096 bytes since Ajay et al. illustrate that such window sizes can best characterize the temporal locality [28].

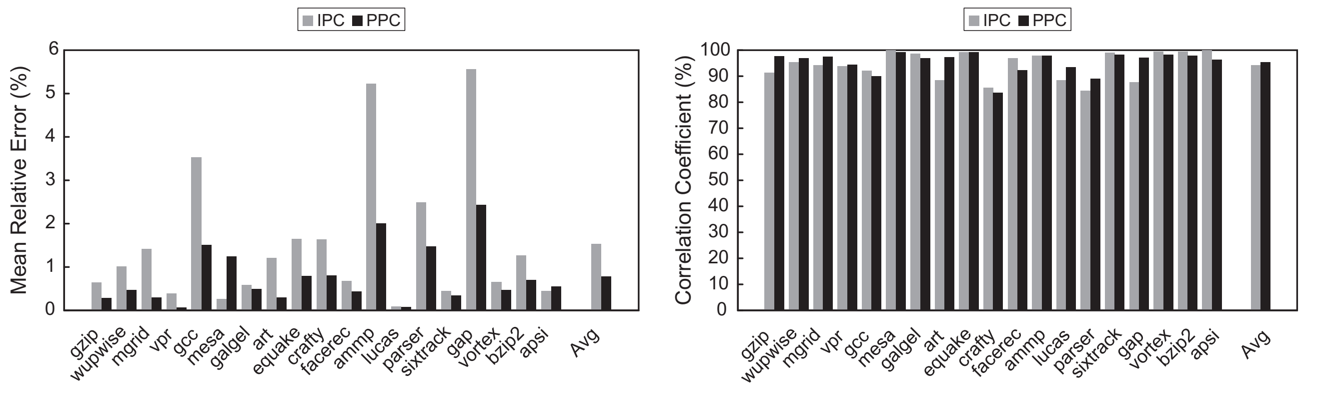
Data spatial locality. The spatial locality assumes that the data in the neighborhood will be accessed soon. Apparently, a program is full of consecutive memory accesses must have better spatial locality compared with a program that is full of random memory accesses. To quantify the comparison of data spatial locality, we measure the average distance between two consecutive data accesses. To reduce profiling costs, we collect the data spatial locality for window sizes of 16, 64, 256 and 4096 bytes.

Instruction temporal locality. Following the similar idea of reuse distance for data locality, instruction locality can also be specified as the distance between two consecutive accesses to the same instruction block. Similarly, we calculate the instruction temporal locality for window sizes of 16, 64, 256 and 4096 bytes.

Instruction spatial locality. Instruction spatial locality is also defined as the way of data spatial locality. Therefore, the average distances between two consecutive accesses to instruction block for window sizes of 16, 64, 256 and 4096 bytes are collected.

Table 2
Baseline architecture employed to evaluate the effectiveness of proposed program characteristics.

Pipeline dimensions	
Pipeline depth	5
Pipeline width	4
Processor core	
ROB/LSQ size	16/8
FALU/IALU	4/4
2-Level predictor	1 L1, 1024 L2, 8-bit history
BTB size	2048 entries, 4-way
Memory hierarchy	
L1 DCache size	16 KB, 4-way, 32B blocks, 1-cycle latency
L1 ICache size	64 KB, 4-way, 32B blocks, 1-cycle latency
L2 UCache size	256 KB, 4-way, 64B blocks, 6-cycle latency
ITLB	256 KB, 30-cycle miss penalty
DTLB	512 KB, 30-cycle miss penalty



(a) Mean relative error of predicting IPC&PPC (b) Correlation coefficient of predicting IPC&PPC

Fig. 5. Evaluation of proposed characteristics for predicting the IPC and PPC of all the phases *in* each program of SPEC CPU2000. Mean relative error and correlation coefficient of IPC&PPC are provided in (a) and (b), respectively.

3.3.3. Evaluation of program characteristics

To eliminate the effects of different architectures, we fix the architectural parameters as presented in Table 2 and only investigate the effectiveness of proposed program characteristics. Actually, many programs consist of as a series of phases, and each phase may be very different from the others. Thus, in order to detailedly investigate program characteristics with more training data other than the number of SPEC CPU2000, we split each program into phases with a granularity of 10 million instructions as performed by Sherwood et al. in [29].

However, since the number of instructions in some of the programs is less than 1 billion (mcf, applu, swim, fma3d, perlbmk and eon), i.e., less than 100 phases, we decide to validate the effectiveness of proposed program characteristics on the rest 19 programs in SPEC CPU2000 (the number of phases in these programs ranges from 120 to more than 1200). Besides, 10-cross validation, which can be used to improve the performance of built models and will be detailed in next section, is employed to minimize the training error.

Fig. 5a and b show the results of MREs and correlation coefficients for predicting IPC and PPC, respectively. From the left side, we can see that MREs of IPC vary from 0.085% (lucas) to 5.226% (ammp), and MREs of PPC vary from 0.08% (lucas) to 2.429% (gap). Moreover, the average MREs of IPC and PPC are 1.53% and 0.774%, respectively. From the right side, the average correlation coefficients of IPC and PPC are 94.28% and 95.4%, respectively, which indicates that the predicted value fits the actual value pretty well. Nevertheless, these encouraging results of preliminary experiment only validate the effectiveness of program characteristics for predicting performance and energy *in* program. To demonstrate the effectiveness of program characteristics *across* programs, we present the prediction results in Table 3 with more than 10,000 phases that are collected from all programs for training and validation.

3.3.4. Refined program characteristics

Apparently, potential correlations among proposed program characteristics are inevitable. To reduce such correlations and obtain a refined characteristics set, which is also beneficial to reduce the extraction costs, we employ so-called *feature selection* technique to process the raw program characteristics. To be specific, we utilize Correlation-based Feature Selection (CFS) to evaluate the merit of different subset of features, i.e., raw program characteristics, so as to identify an optimal subset. CFS selection algorithm takes effect by evaluating the worth of a subset of attributes via considering the individual predictive ability of each feature along with the degree of redundancy among them [30]. Then, subsets of features that are highly correlated with target variables (here IPC or PPC) and have low inter-correlation are preferred. Furthermore, since it is infeasible to exhaustively evaluate the whole exponential space of combinations of features, we use a simple genetic search [31] to get the optimal subset. Table 4 shows the refined program characteristics obtained by employed techniques.

Table 3
Effectiveness of program characteristics for performance modeling *across* programs.

	Mean relative error (%)	Correlation coefficient (%)
IPC	2.85	95.11
PPC	1.94	98.55

Table 4
Refined program characteristics via CFS + genetic search.

Characteristics	Definition
DATADEP4	% of data dependency distance = 8
DATADEP6	% of data dependency distance = 32
DATADEP8	% of data dependency distance > 64
BRATRAN	Branch transition rate
DATATL4	Data temporal locality with 4098B block
INSTTL1	Inst temporal locality with 16B block
DATASL3	Data spatial locality with 256B block

3.4. Model training and deployment

3.4.1. Data formulation

As an important process during the application of machine learning techniques, we also need to describe how to formulate the input data for creating and deploying models that try to approximate the behavior of a design. Intuitively, the input data for training primarily consists of three parts: program characteristics, architectural design configurations and their corresponding responses.

More specifically and formally, for each training program P_i (its inherent characteristics is $\boldsymbol{c_i}$ as detailed in last section), we can obtain its associated response r_j, e.g., IPC, of a certain architectural design configuration $\boldsymbol{a_j}$ via cycle-accurate simulation. Thus, the actual form of the training data tuple for model tree is $(\boldsymbol{c_i}, \boldsymbol{a_j}, r_j)$. Through training a group of the input data with above formulation, we can obtain aforementioned model tree as a combination of decision rules and linear regression models as f_{IPC}. As we obtain such a model, we can employ it to cope with a new program P_{i+1}, whose characteristics should be extracted first as $\boldsymbol{c_{i+1}}$ via trivial profilling. Thus, IPC of P_{i+1} on a new interested design options $\boldsymbol{a_{j+1}}$ can be estimated as $IPC = f_{IPC}(\boldsymbol{c_{i+1}}, \boldsymbol{a_{j+1}})$.

3.4.2. Composition of training data

Although we have specified the formulation of each input data for training and deployment, the size and composition of the training dataset still need to be addressed. The ideal training dataset should be as representative and large as possible. However, due to the time-consuming simulation, we hope to obtain a training set which is small while representative enough.

In essence, the representation problem of the training set in our approach can be considered from two orthogonal dimensions, i.e., architecture and program. To attain representative architectures, we *randomly* sample 500 design points as the training set by trading-off the simulation costs and prediction accuracy as shown in Section 3.2. However, sampling representative programs is not so intuitive since the number of programs to be trained is not comparable to the number of design options. Moreover, we hope to reduce the training costs as much as possible while controlling the prediction accuracy at hand. In our experiment, for simplicity, we use one program for validation and all the rest programs in SPEC CPU2000 as training data.

3.4.3. Cross-validation

Although employed model tree is claimed to achieve high genelizability, we still need to perform *cross-validation* to reduce the risk of overfitting. This technique divides the training set into multiple equal subsets or *folds*. In our case, we split the training set into 10 folds. We take folds 1–9 as training data to build the first model, and fold 10 is employed as testing set to evaluate the performance of this trained model. Then, the selection of folds as training data is rotated, i.e., folds 2–10 are trained for second model and fold 1 is treated as testing data, and so on. At last, the outputs of resulting 10 model trees are averaged for final prediction.

Consequently, cross-validation can reduce error variance and improve accuracy at the cost of training multiple models.

4. Primary experiments

Apparently, exploring the design space needs to traverse the 2-dimensional space that consists of workloads and design configurations. As an illustrative example and for fair comparison, the workload space is from SPEC CPU2000, which consists of 26 programs. And the architectural space to explore is listed in Table 1 with 70,778,880 design options.

Our technique focuses on predicting the performance of *new* programs on *new* architectural configurations. Thus, for each program in SPEC CPU2000, we first train **25** programs in the

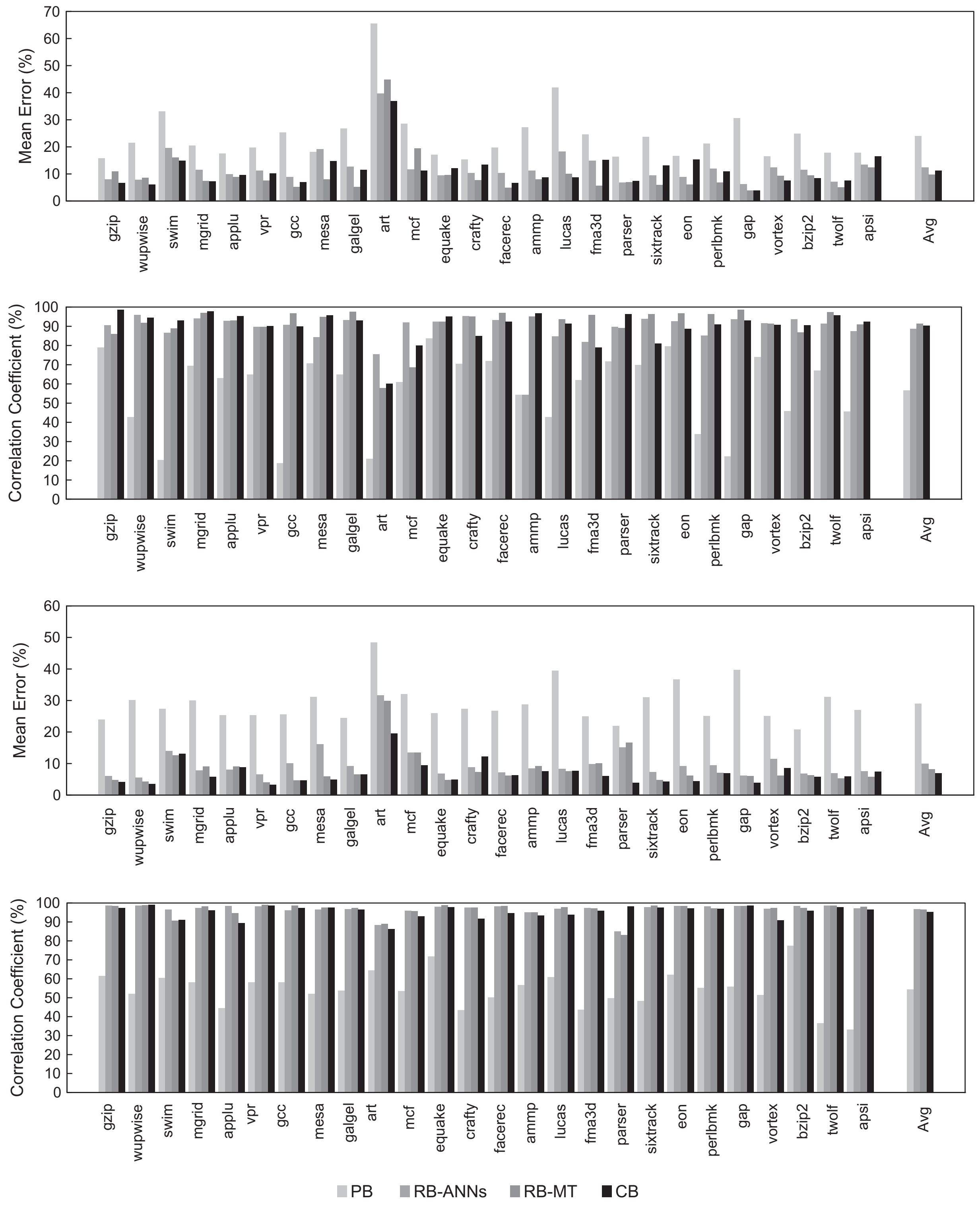


Fig. 6. Comparison of MREs and correlation coefficients of PB, RB-ANNs, RB-MT and CB. From top to bottom, it illustrates the MRE of instruction-per-cycle (IPC), correlation coefficients of IPC, MRE of power-per-cycle (PPC) and correlation coefficients of PPC, respectively.

benchmark suite with refined program characteristics, 500 architectural configurations and corresponding processor responses to attain two predictive models for performance and energy, respectively. Then, the built models are validated by *the rest* program with 500 *new* architectural configurations. To demonstrate the effectiveness of our approach (which is named as the Characteristic-Based approach, CB for short), we compare it with the previous Program-Based approach (PB) which builds predictive model for each program, and the Reaction-Based approach (RB) which employs 8 reactions as signature [14]. Moreover, since RB [14,15] always employed ANNs as the learning engines, we perform experiments on RB utilizing either ANNs or Model Tree (MT). As shown in Fig. 6, in comparison with PB, the average error has been reduced from 23.98% to 11.22% for performance and from 29.05% to 6.88% for energy, respectively. For RB, RB-MT is more accurate and reliable than RB-ANNs, which validates the effectiveness of employed model tree technique. Furthermore, the performance of our approach is comparable to that of RB-MT, where the average errors are 9.78%, 8.20% for performance and energy, respectively. Moreover, the correlation coefficients of CB are 90.26% and 95.4% for performance and energy, respectively. Besides, to evaluate the performance of our approach on some emerging benchmarks in addition to SPEC CPU2000, we also conduct experiments on two benchmarks, *bwaves* and *hmmer*, selected from SPEC CPU2006 benchmark suite. The performance and energy are predicted by the model trained by all 26 SPEC CPU2000 benchmarks. Regarding *bwaves*, the average errors for predicting performance and energy are 8.32% and 4.89%, respectively. Regarding *hmmer*, the average errors for predicting performance and energy are 7.28% and 3.85%, respectively.

To sum up, high prediction accuracy implies that our approach is very practical and effective for microarchitectural design space exploration. If more representative programs are selected (e.g., with Workload Subsetting techniques [5,6]) for training data, it is expected that the prediction accuracy can be further improved.

5. Notes on training costs

So far our experimental results have shown that the accuracy of our approach significantly outperforms PB, and is comparable to that of RB. In this section, we present the training costs of our approach, showing that it is more efficient than previous ones, which is one of the most notable merits of our approach.

5.1. Comparison with previous approaches

To demonstrate the efficiency of our approach, we compare the training costs of CB with PB&RB. Similarly, eight cycle-accurate simulation runs are employed to obtain the training data of PB&RB for new programs, as mentioned in the last section. However, for our approach, only one trivial profiling is sufficient to gain proposed refined program characteristics as shown in Table 4. Fig. 7 demonstrates the speedup gained by CB over PB&RB, where we can find that our approach can obtain 7.6–11.8× speedup over previous approaches for each program. And for all programs in SPEC CPU2000, the speedup is 8.4×. In more detail, for PB&RB, to collect the training data of all programs, we have to run about 524 machine hours on AMD Opteron processor. Nevertheless, our approach only requires 62 machine hours to obtain necessary program characteristics. Therefore, the training costs are dramatically reduced by our approach, which is very practical for exploring design space by many programs, and is helpful to achieve early design decisions.

5.2. Further reducing the training costs

So far we have seen that the training costs of CB are significantly smaller than those of previous approaches, especially when more programs are evaluated. A good news is that the training costs of our approach can be further reduced via instrumentation technique on a real machine. Specifically, since the inherent program

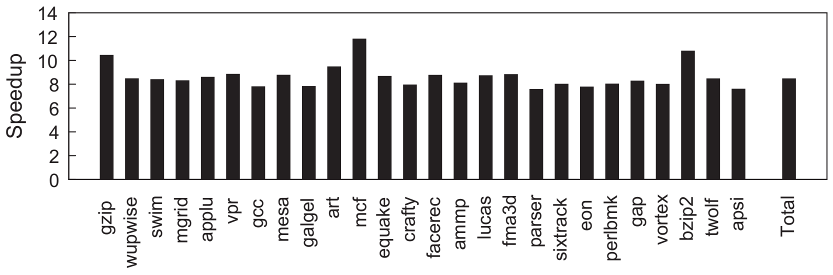


Fig. 7. Speedup of our approach over PB&RB when encountering new programs. For *mcf*, it gains at most 11.8× speedup. And the program with least speedup is *parser*, as 7.6× speedup. Precisely, for PB&RB, the training overhead for all 26 programs is 524 machine hours, and for our approach it is only 62 machine hours, achieving 8.4× speedup. Such encouraging result ensures efficient exploration of the universal design space.

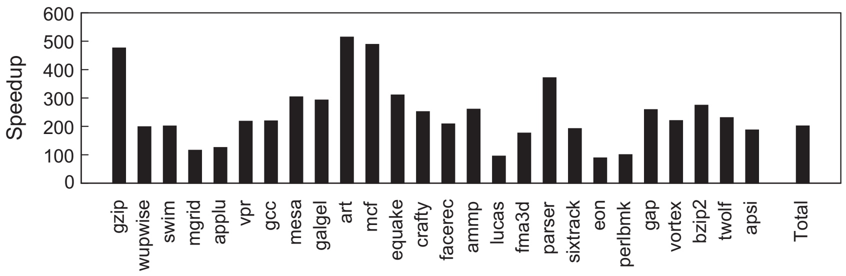


Fig. 8. Further reducing the training costs with instrumentation technique on a real machine compared with previous approaches. It is feasible based on the assumption that a real machine with compatible ISA is available. Results show that speedup ranges from 90× to 490× for each workload. In more detail, for all 26 programs in SPEC CPU2000, the total time is reduced from 524 machine hours to 2.6 machine hours, achieving 200× speedup.

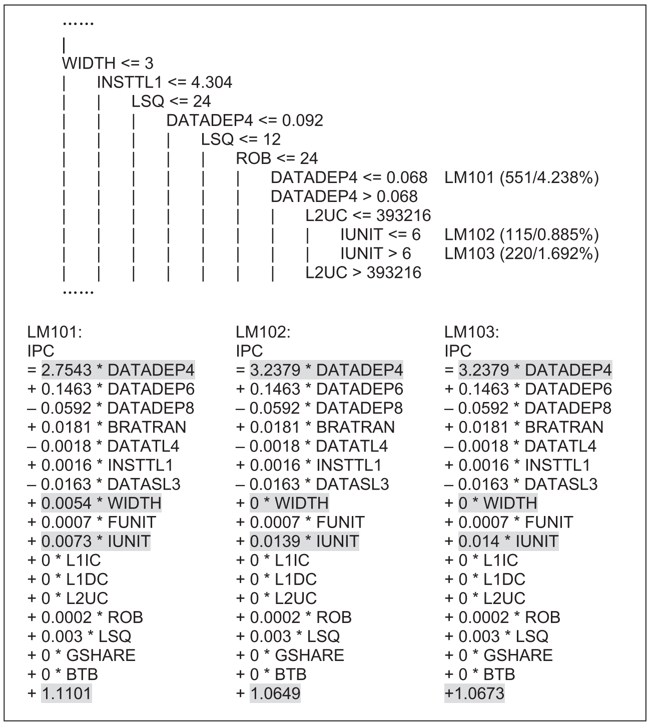
characteristics utilized by our approach are microarchitecture-independent, we can utilize real machines with the same instruction set architecture (ISA) to rapidly acquire the inherent program characteristics. This idea is quite practical in industry, because most successful chips are series of productions, and their instruction sets are compatible. In this case, exploiting the inherent program characteristics required by our approach would be very fast. For example, we utilize an instrumentation tool that is developed from PIN [32] on a X86 machine to extract the inherent program characteristics, which is similar to the methodology adopted by MICA [21]. As illustrated in Fig. 8, extracting program characteristics on a real machine can achieve 90–490× speedup for each program (200× speedup for the programs as a whole), compared with previous approaches which require specific cycle-accurate simulations for each program.

6. Insights of design space

In addition to the low training costs, another notable feature of our approach is that it provides interpretable models for architects. Precisely, owing to employing model tree based learning technique, we can specify the priority and weight of the design parameter according to *decision rules* which constitute the tree structure and *linear models* on the tree leaves, and then optimize the architecture accordingly.

Naturally, a large model tree with many decision rules and linear models is required to characterize the complicated interaction between program and architecture. Due to the page limit, we only present a small section of the overall tree structures of performance and energy in Fig. 9. From the subtree of the performance model in Fig. 9a, we can see that WIDTH (Fetch/Issue/Commit Width as shown in Table 1) is the most important parameter that influences the performance, which complies with the intuition that the width of the superscalar processor is crucial to exploit the inherent parallelism of the programs. Given the setting that WIDTH $\leqslant 3$, the next important parameter is INSTTL1, which is a program characteristic describing the instruction temporal locality with 16B block. When INSTTL1 $\leqslant 4.304$ (which is a threshold learned by the model tree algorithm), the distance between two accesses to the same instruction block is very short, which indicates great instruction temporal locality. In this case, we should specify the priority of LSQ over ROB [with DATADEP4 (percentage of data dependency distance = 8 as shown in Table 4) to characterize the inherent parallelism of the program]. It is interesting that IUNIT (IALU/IMUL Units) is considered as the last level of the performance tree, which implies that the influence of IUNIT is low when WIDTH, LSQ and ROB are small. A reasonable explanation is that integer function units are hard to be fully utilized when the instruction window is small, even when executing a program with great instruction locality.

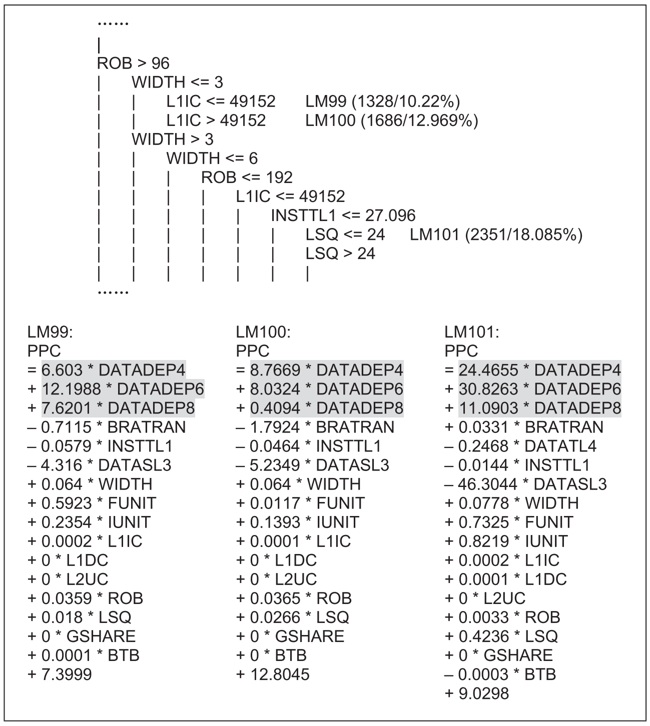
More specifically, the priority of design parameters can be validated by linear models on the leaves. By comparing linear model LM101, LM102 and LM103, we can conclude that LM103 is closer to LM102 than to LM101. According to the tree structure, LM102 and LM103 possess the same parent, which validates that design parameters near the root are more discriminative than ones near the leaves. Furthermore, once the ranges of most parameters are bounded, for example, arriving the leaves IUNIT $\leqslant 6$ with constraints on WIDTH, LSQ, ROB and L2UC (Level 2 Universal Cache), we can further tune the performance in a fine granularity according to the weight of each design parameter of the linear models on the leaves (here as LM102). In this case, tuning LSQ can achieve more performance improvement than ROB since the weight of LSQ (0.003) is greater than that of ROB (0.0002).



```
......
|
WIDTH <= 3
|   INSTTL1 <= 4.304
|   |   LSQ <= 24
|   |   |   DATADEP4 <= 0.092
|   |   |   |   LSQ <= 12
|   |   |   |   |   ROB <= 24
|   |   |   |   |   |   DATADEP4 <= 0.068   LM101 (551/4.238%)
|   |   |   |   |   |   DATADEP4 > 0.068
|   |   |   |   |   |   |   L2UC <= 393216
|   |   |   |   |   |   |   |   IUNIT <= 6   LM102 (115/0.885%)
|   |   |   |   |   |   |   |   IUNIT > 6    LM103 (220/1.692%)
|   |   |   |   |   |   |   L2UC > 393216
......

LM101:                    LM102:                    LM103:
IPC                       IPC                       IPC
= 2.7543 * DATADEP4       = 3.2379 * DATADEP4       = 3.2379 * DATADEP4
+ 0.1463 * DATADEP6       + 0.1463 * DATADEP6       + 0.1463 * DATADEP6
– 0.0592 * DATADEP8       – 0.0592 * DATADEP8       – 0.0592 * DATADEP8
+ 0.0181 * BRATRAN        + 0.0181 * BRATRAN        + 0.0181 * BRATRAN
– 0.0018 * DATATL4        – 0.0018 * DATATL4        – 0.0018 * DATATL4
+ 0.0016 * INSTTL1        + 0.0016 * INSTTL1        + 0.0016 * INSTTL1
– 0.0163 * DATASL3        – 0.0163 * DATASL3        – 0.0163 * DATASL3
+ 0.0054 * WIDTH          + 0 * WIDTH               + 0 * WIDTH
+ 0.0007 * FUNIT          + 0.0007 * FUNIT          + 0.0007 * FUNIT
+ 0.0073 * IUNIT          + 0.0139 * IUNIT          + 0.014 * IUNIT
+ 0 * L1IC                + 0 * L1IC                + 0 * L1IC
+ 0 * L1DC                + 0 * L1DC                + 0 * L1DC
+ 0 * L2UC                + 0 * L2UC                + 0 * L2UC
+ 0.0002 * ROB            + 0.0002 * ROB            + 0.0002 * ROB
+ 0.003 * LSQ             + 0.003 * LSQ             + 0.003 * LSQ
+ 0 * GSHARE              + 0 * GSHARE              + 0 * GSHARE
+ 0 * BTB                 + 0 * BTB                 + 0 * BTB
+ 1.1101                  + 1.0649                  +1.0673
```

(a) A section of the performance model tree



```
......
|
ROB > 96
|   WIDTH <= 3
|   |   L1IC <= 49152    LM99 (1328/10.22%)
|   |   L1IC > 49152     LM100 (1686/12.969%)
|   WIDTH > 3
|   |   WIDTH <= 6
|   |   |   ROB <= 192
|   |   |   |   L1IC <= 49152
|   |   |   |   |   INSTTL1 <= 27.096
|   |   |   |   |   |   LSQ <= 24   LM101 (2351/18.085%)
|   |   |   |   |   |   LSQ > 24
|   |   |   |   |   |   |
......

LM99:                     LM100:                    LM101:
PPC                       PPC                       PPC
= 6.603 * DATADEP4        = 8.7669 * DATADEP4       = 24.4655 * DATADEP4
+ 12.1988 * DATADEP6      + 8.0324 * DATADEP6       + 30.8263 * DATADEP6
+ 7.6201 * DATADEP8       + 0.4094 * DATADEP8       + 11.0903 * DATADEP8
– 0.7115 * BRATRAN        – 1.7924 * BRATRAN        + 0.0331 * BRATRAN
– 0.0579 * INSTTL1        – 0.0464 * INSTTL1        – 0.2468 * DATATL4
– 4.316 * DATASL3         – 5.2349 * DATASL3        – 0.0144 * INSTTL1
+ 0.064 * WIDTH           + 0.064 * WIDTH           – 46.3044 * DATASL3
+ 0.5923 * FUNIT          + 0.0117 * FUNIT          + 0.0778 * WIDTH
+ 0.2354 * IUNIT          + 0.1393 * IUNIT          + 0.7325 * FUNIT
+ 0.0002 * L1IC           + 0.0001 * L1IC           + 0.8219 * IUNIT
+ 0 * L1DC                + 0 * L1DC                + 0.0002 * L1IC
+ 0 * L2UC                + 0 * L2UC                + 0.0001 * L1DC
+ 0.0359 * ROB            + 0.0365 * ROB            + 0 * L2UC
+ 0.018 * LSQ             + 0.0266 * LSQ            + 0.0033 * ROB
+ 0 * GSHARE              + 0 * GSHARE              + 0.4236 * LSQ
+ 0.0001 * BTB            + 0 * BTB                 + 0 * GSHARE
+ 7.3999                  + 12.8045                 – 0.0003 * BTB
                                                    + 9.0298
```

(b) A section of the energy model tree

Fig. 9. Sections extracted from the whole performance (a) and energy model tree (b). The learned models consist of two parts, *decision rules* (e.g., WIDTH $\leqslant 3$) contribute to the tree structure and *linear models* (e.g., LM101 of performance model with 4.238% of the training data falling into this category) on the leaves. The discrimination of design parameters and program characteristics decreases as traversing the tree from the root to the leaves. Moreover, linear models on the leaves quantitatively demonstrate the impact of individual parameters on the performance/energy.

On the other hand, according to the energy model tree in Fig. 9b, ROB is the most discriminative design parameter for power consumption. Getting into the second level of this subtree, if WIDTH is very small ($\leqslant 3$), we only need to consider the impact of L1IC (Level 1 Instruction Cache) and can neglect the effects of other parameters, since most processor components would not be triggered frequently due to limited instruction-level parallelism. Otherwise, if WIDTH is medium ($3 < \text{WIDTH} \leqslant 6$), ROB, L1IC and LSQ should be taken into consideration in order. Additionally, if a small L1IC ($<48K$) is chosen, program characteristics such as INSTTL1 should be considered to decide the next design parameters to evaluate. Besides, from the point view of listed linear models, the impact of DATADEP (including DATADEP4, DATADEP6 and DATADEP8) on the energy in LM101 significantly increases since more parallelism can be exploited with larger WIDTH compared with LM99 and LM100.

Moreover, individual parameter may be proportional or inversely proportional to the performance/energy. For example, the program locality parameters (DATATL4, INSTTL1 and DATASL3) are inversely proportional to the PPC from presented subtree of energy model. Apparently, programs with poor locality may require more resources (such as memory access queue, memory bandwidth, etc.) in comparison with programs accessing a small and regular working set. An interesting phenomenon in this subtree is that BTB is inversely proportional to the resultant energy in despite of a small ratio (0.0003), which implies that architectures with larger BTB will gain less power consumption potentially due to the better branch prediction accuracy.

Overall, the benefits of utilizing inherent program characteristics and model tree for predictive modeling can be summarized as follows. First, according to the order of decision rules traversed from the root to the leaves, we can focus available budgets on issues that are most discriminative to performance/energy. Besides, once the ranges of most parameters are bounded, microarchitecture can be further tuned based on the linear models on the leaves.

7. Related work

As stated in Section 1, tremendous work elucidate the problem of efficient microarchitectural design space exploration. Here, we focus on the detailed taxonomy of simulation-based approaches since they are still the most practical way in industry.

At first, *statistical sampling* simulates some representative instruction phases rather than the whole program via periodic sampling [1]. Meanwhile, *statistical simulation* uses synthetic instructions from the statistical profiling of the original workloads. This technique evolves from simple models [2] to complex and detailed models [3], and also from uniprocessors to CMPs [4]. On the other hands, through considering the workload in a higher level, Eeckout et al. [5] proposed statistical *workload subsetting* to select a representative subset of the original benchmark and input pair set for simulation. And Yi [6] also proposed a workload subsetting approach through Plackett and Burman (P&B) design.

All those above techniques try to reduce the simulation time from the perspective of workloads, in other words, trying to use a reduced representative set of workloads (in program- or instruction-level) to evaluate the performance (thus we refer as *representative methods* in short). While predictive modeling, which employs regression/machine learning techniques to predict the performance, deals with this problem in a different way. As stated in Section 1, most of the previous predictive modeling techniques focus on program-based prediction while concrete learning technique differs, e.g., linear regression [7], ANNs [8], radial basis function [9], spline functions regression [10,11], genetic programming (GP) [12] and semi-supervised learning techniques [33]. Besides, Cho et al. proposed a predictive model for workload dynamics based on wavelet-based multiresolution decomposition and neural network regression [13]. Li et al. also proposed to employ regression tree-based models, such as multiple additive regression tree [35,36] and boosted regression tree [37], to predict the performance (or vulnerability). Nevertheless, such approaches still require to construct predictive models for each program. To address this problem, Khan et al. [14] and Dubach et al. [15] have separately extended the work so as to adapt to new programs via reaction-based approaches. The assumption behind these approaches is that programs with similar responses on typical configurations might have similar behavior on other configurations. However, this assumption may not hold given a large design space to explore. For instance, *parser* from SPECint and *sixtrack* from SPECfp may achieve equivalent IPC on a certain architecture with more floating-point units than integer units, while we cannot ensure that they will exhibit similar behavior on other configurations with more integer units. Unlike traditional approaches, we propose to explicitly incorporate inherent program characteristics for training predictive models, which can significantly reduce the simulation costs. Besides, our approach can cooperate with aforementioned representative methods to further speedup the simulation for training and evaluation.

8. Conclusion

In this paper, we propose a universal predictive modeling technique, which incorporates inherent program characteristics as a part of the training data, to effectively and efficiently obtain optimal microarchitectures according to the design specification, given the whole $M \times N$ space needs to traverse during design space exploration. Since specific cycle-accurate simulations for each program in previous approaches are not required by our approach, the overall simulation costs can be significantly reduced. Besides, due to employed model tree, our approach not only efficiently provides an accurate predictive model across programs but also quantitatively illustrates the priority and weight of individual parameter to performance/energy, which helps architects focus available budgets on most discriminative design issues.

Primary experimental results on cross-program prediction show that average errors for predicting performance and energy are 11.22% and 6.88%, respectively. Besides, the training time of our approach achieves 7.6–11.8$\times$ speedup over previous approaches for each workload. Furthermore, we can employ instrumentation technique to achieve 90–490$\times$ for each workload. In summary, experimental results have validated that our approach is effective and efficient for universal design space exploration.

In the future, we prepare to utilize proposed predictive modeling technique in the design of multi-core processors, where the simulation speed is several orders of magnitude slower (than that of the uni-core), and more workloads and interactive design parameters should be involved in evaluations. Besides, we also consider to apply this model to find the pareto front of a design space via either trivially comparing all the predicted responses or some mature multi-objective optimization algorithms, such as NSGA-II [34].

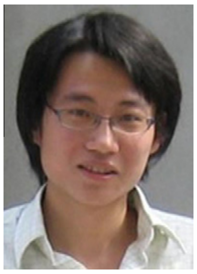
Acknowledgement

This work is partially supported by the National Natural Science Foundation of China (under Grants 61003064, 61100163, 61173006, 61133004, and 60921002), the National S&T Major Project of China (under Grants 2009ZX01028-002-003, 2009ZX01029-001-003, and 2010ZX01036-001-002), the National 863 Program of China (under Grant 2012AA012202), and the Strategic Priority Research Program of the Chinese Academy of Sciences (under Grant XDA06010401).

References

[1] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, J.C. Hoe, Smarts: accelerating microarchitecture simulation via rigorous statistical sampling, in: Proceedings of ISCA, 2003, pp. 84–97.
[2] S. Nussbaum, J.E. Smith, Modeling superscalar processor via statistical simulation, in: Proceedings of PACT, 2001, pp. 15–24.
[3] L. Eeckhout, R.H. Bell Jr., B. Stougie, K.D. Bosschere, L.K. John, Control flow modeling in statistical simulation for accurate and efficient processor design studies, in: Proceedings of ISCA, 2004, pp. 350–361.
[4] D. Genbrugge, L. Eeckhout, Chip multiprocessor design space exploration through statistical simulation, IEEE Trans. Comput. 58 (12) (2009) 1668–1681.
[5] L. Eeckhout, H. Vandierendonck, K.D. Bosschere, Workload design: selecting representative program-input pairs, in: Proceedings of PACT, 2002, pp. 83–94.
[6] J.J. Yi, D.J. Lilja, D.M. Hawkins, A statistically rigorous approach for improving simulation methodology, in: Proceedings of HPCA, 2003, pp. 281–291.
[7] P. Joseph, K. Vaswani, M. Thazhuthaveetil, Construction and use of linear regression models for processor performance analysis, in: Proceedings of HPCA, 2006, pp. 99–108.
[8] E. Ïpek, S.A. McKee, R. Caruana, B.R. de Supinski, M. Schulz, Efficiently exploring architectural design spaces via predictive modeling, in: Proceedings of ASPLOS, 2006, pp. 195–206.
[9] P.J. Joseph, K. Vaswani, M.J. Thazhuthaveetil, A predictive performance model for superscalar processors, in: Proceedings of MICRO, 2006, pp. 161–170.
[10] B.C. Lee, D.M. Brooks, Illustrative design space studies with microarchitectural regression models, in: Proceedings of HPCA, 2007, pp. 340–351.
[11] B.C. Lee, J. Collins, H. Wang, D. Brooks, Cpr: composable performance regression for scalable multiprocessor models, in: Proceedings of MICRO, 2008, pp. 270–281.
[12] H. Cook, K. Skadron, Predictive design space exploration using genetically programmed response surfaces, in: Proceedings of DAC, 2008, pp. 960–965.
[13] C.-B. Cho, W. Zhang, T. Li, Informed microarchitecture design space exploration using workload dynamics, in: Proceedings of MICRO, 2007, pp. 274–285.
[14] S. Khan, P. Xekalakis, J. Cavazos, M. Cintra, Using predictive modeling for cross-program design space exploration, in: Proceedings of PACT, 2007, pp. 327–338.
[15] C. Dubach, T. Jones, M. O'Boyle, Microarchitectural design space exploration using an architecture-centric approach, in: Proceedings of MICRO, 2007, pp. 262–271.
[16] S.Y. Borkar, P. Dubey, K.C. Kahn, D.J. Kuck, H. Mulder, E.R.M. Ramanathan, V. Thomas, I. Corporation, S.S. Pawlowski, Platform 2015: Intel processor and platform evolution for the next decade executive summary, Intel White Paper. 2005.
[17] W. Hu, J. Wang, X. Gao, Y. Chen, Q. Liu, G. Li, Godson-3: a scalable multicore RISC processor with X86 emulation, IEEE Micro 29 (2) (2009) 17–29.
[18] J.R. Quinlan, Learning with continuous classes, in: Proceedings of AJCAI, 1992, pp. 343–348.
[19] T. Austin, E. Larson, D. Ernst, Simplescalar: an infrastructure for computer system modeling, Computer 35 (2) (2002) 59–67.
[20] D. Brooks, V. Tiwari, M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, in: Proceedings of ISCA, 2000, pp. 83–94.
[21] K. Hoste, L. Eeckhout, Microarchitecture-independent workload characterization, IEEE Micro 27 (2007) 63–72.
[22] S. Eyerman, L. Eeckhout, T. Karkhanis, J.E. Smith, A performance counter architecture for computing accurate cpi components, in: Proceedings of ASPLOS, 2006, pp. 175–184.
[23] M.D. Powell, A. Biswas, J.S. Emer, S.S. Mukherjee, B.R. Sheikh, S.M. Yardi, Camp: a technique to estimate per-structure power at run-time using a few simple parameters, in: Proceedings of HPCA, 2009, pp. 289–300.
[24] M. Haungs, P. Sallee, M.K. Farrens, Branch transition rate: a new metric for improved branch classification analysis, in: Proceedings of HPCA, 2000, pp. 241–250.
[25] T. Rauber, G. Rünger, Program-based locality measures for scientific computing, in: Proceedings of IPDPS, 2003, pp. 164–172.
[26] X. Gu, I. Christopher, T. Bai, C. Zhang, C. Ding, A component model of spatial locality, in: Proceedings of ISMM, 2009, pp. 99–108.
[27] Y. Zhong, X. Shen, C. Ding, Program locality analysis using reuse distance, ACM Trans. Program. Lang. Syst. 31 (6) (2009) 1–39.
[28] J. Ajay, P. Aashish, E. Lieven, K.J. Lizy, Measuring benchmark similarity using iherent program characteristics, IEEE Trans. Comput. 55 (6) (2006) 769–782.
[29] T. Sherwood, S. Sair, B. Calder, Phase tracking and prediction, in: Proceedings of ISCA, 2003, pp. 336–349.
[30] M.A. Hall, Correlation-based feature selection for discrete and numeric class machine learning, in: Proceedings of ICML, 2000, pp. 359–366.
[31] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, 1989.
[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: Proceedings of PLDI, 2005, pp. 190–200.
[33] Q. Guo, T. Chen, Y. Chen, Z.-H. Zhou, W. Hu, Z. Xu, Effective and efficient microprocessor design space exploration using unlabeled design configurations, in: Proceedings of IJCAI, 2011, pp. 1671–1677.
[34] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evolution. Comput. 6 (2) (2002) 182–197.
[35] B. Li, L. Peng, B. Ramadass, Efficient MART-aided modeling for microarchitecture design space exploration and performance prediction, in: Proceedings of SIGMETRICS, 2008, pp. 439–440.
[36] B. Li, L. Peng, B. Ramadass, Accurate and efficient processor performance prediction via regression tree based modeling, J. Syst. Architect. 55 (2009) 457–467.
[37] B. Li, L. Duan, L. Peng, Efficient microarchitectural vulnerabilities prediction using boosted regression trees and patient rule inductions, IEEE Trans. Comput. 59 (5) (2010) 593–607.

Qi Guo received the B.S. degree in computer sciences from Department of Computer Science and Technology, Tongji University, Shanghai, China, in 2007, and the Ph.D. degree in computer sciences from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS) in 2012, Beijing, China. His research interests include computer architecture, performance evaluation, VLSI design and verification.


Tianshi Chen received the B.S. degree in mathematics from the Special Class for the Gifted Young, University of Science and Technology of China (USTC), Hefei, China, in 2005, and the Ph.D. degree in computer science from USTC in 2010. He is currently an assistant professor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include parallel computing and computational intelligence.

Yunji Chen graduated from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, China, in 2002. Then, he received the Ph.D. degree in computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2007. He is currently an associate professor at ICT. His research interests include parallel computing, microarchitecture, hardware verification, and computational intelligence. He has authored or coauthored 1 book and over 40 papers in these areas.

Ling Li received the B.S. degree in computer science from Wuhan University, Wuhan, China, in 2004, and the Ph.D. degree from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 2009. She is currently an assistant professor at ICT. Her research interests include image/video processing, signal processing, and high performance computing.