A Predictive Model for Dynamic Microarchitectural Adaptivity Control

Christophe Dubach, Timothy M. Jones
Members of HiPEAC
University of Edinburgh

Edwin V. Bonilla
NICTA &
Australian National University

Michael F. P. O'Boyle
Member of HiPEAC
University of Edinburgh

***Abstract*—Adaptive microarchitectures are a promising solution for designing high-performance, power-efficient microprocessors. They offer the ability to tailor computational resources to the specific requirements of different programs or program phases. They have the potential to adapt the hardware cost-effectively at runtime to any application's needs. However, one of the key challenges is how to dynamically determine the best architecture configuration at any given time, for any new workload.**

This paper proposes a novel control mechanism based on a predictive model for microarchitectural adaptivity control. This model is able to efficiently control adaptivity by monitoring the behaviour of an application's different phases at runtime. We show that using this model on SPEC 2000, we double the energy/performance efficiency of the processor when compared to the best static configuration tuned for the whole benchmark suite. This represents 74% of the improvement available if we knew the best microarchitecture for each program phase ahead of time. In addition, we show that the overheads associated with the implementation of our scheme have a negligible impact on performance and power.

I. Introduction

Adaptive superscalar microarchitectures are a promising solution to the challenge of designing high-performance, power-efficient microprocessors. They offer the ability to tailor computational resources to the specific requirements of an application, providing performance when the application needs it. At other times, hardware structures can be reorganised or scaled down for a significantly reduced energy cost. These architectures have the potential to cost-effectively adapt the hardware at runtime to any application's needs.

The amount of adaptation available directly determines the level of performance and power-savings achievable. With high adaptivity the processor is able to vary many different microarchitectural parameters. This maximises the degree of flexibility available to the hardware, allowing adaptation of the computational resources to best fit the varying structure of the running program. Although previous work has quantified the theoretical benefits of high adaptivity [1], predicting and delivering this adaptation is still an open and challenging problem. The key question is how to dynamically determine the right hardware configuration at any time, for any unseen program.

In order to achieve the potential efficiencies of high adaptivity we require an effective control mechanism that predicts the right hardware configuration in time. Simple feedback mechanisms that predict the future occupancy requirements of a resource based on the recent past [2], [3] will not scale to a large number of configurations. Other prior works have used statistical machine learning to construct models which estimate the performance and/or power as a function of the microarchitectural configuration [4], [5], [6], [7]. However, these approaches are not practical in a dynamic setting. We wish to predict the best microarchitectural parameter values rather than the performance of any given configuration. Prior work would require online searching and evaluation of the microarchitectural configuration space which is not realistic for anything other than trivial design spaces. What we require are light-weight, runtime control mechanisms.

This paper develops a runtime resource management scheme that predicts the best hardware configuration for any phase of a program to maximise energy efficiency. We use a soft-max machine learning model based on runtime hardware counters to predict the best level of resource adaptation. Our model is constructed empirically by identifying optimal designs on training data. Optima from off-line training quickly guide the model to runtime optima for each adaptive interval. We show that determining the right hardware counters is critical in accurately predicting the right hardware configuration. We also show that predicting the right configuration is an unusually difficult learning problem which explains the lack of progress in this area.

Whenever the program enters a new phase of execution, our technique profiles the application to gather a new type of temporal histogram hardware counter. These are fed into our model which dynamically predicts the best hardware configuration to use for that phase and enables us to double the average energy/performance efficiency over the best possible static design. This represents 74% of the improvement available from knowing the best microarchitecture for each program phase from our sample space ahead of time.

The rest of this paper is structured as follows. Section II motivates the use of machine learning for adaptivity. Section III then describes our approach to dynamic adaptation using a model explained in section IV. Section V presents the experimental setup and section VI evaluates our approach. Section VII investigates model accuracy and section VIII describes implementation details. Section IX describes related work and finally section X concludes.

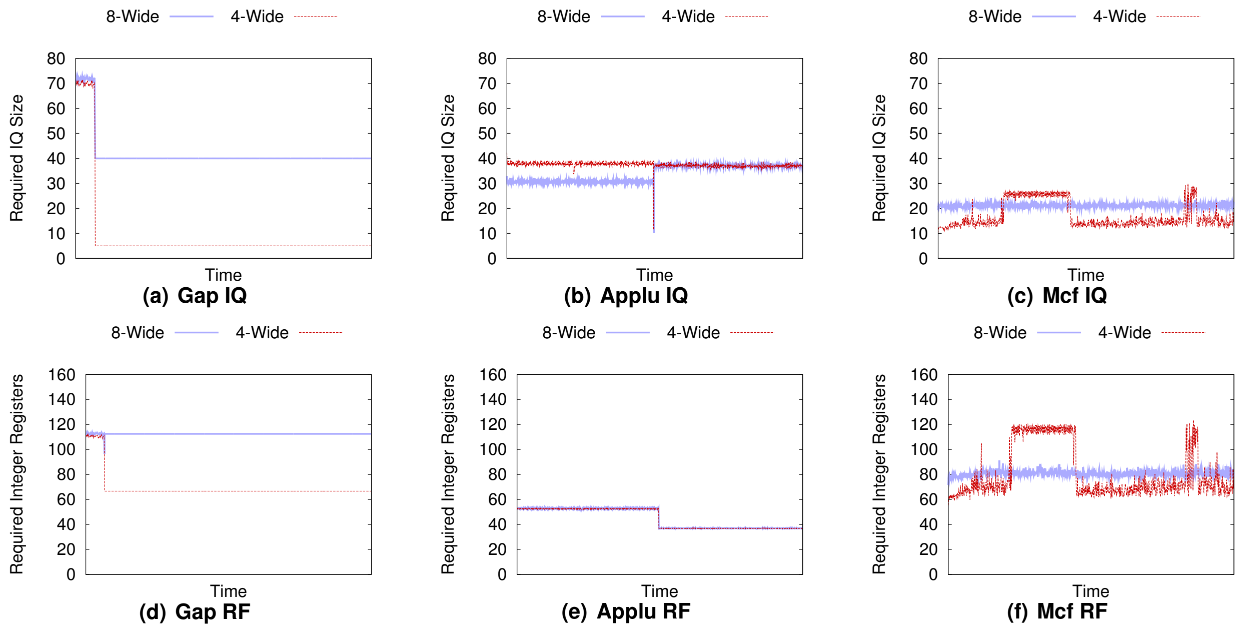


Figure 1. How the optimal size of two processor structures varies with time for pipeline widths 8 and 4 for three applications.

II. The Need for ML-Based Control

This paper proposes a novel technique for dynamic microprocessor adaptation that differs substantially from prior work. Existing schemes, described in section IX, have either focused on adapting only a few microarchitectural parameters at a time, or proposed techniques for efficient searching of the design space at runtime. However, these schemes are not suited for adapting an entire processor's resources due to the complex interactions that exist between hardware structures. Furthermore, runtime searching is undesirable since it would inevitably visit poorly-performing configurations, reducing overall efficiency. We require a control mechanism that can quickly identify the optimal global hardware configuration to minimise power consumption whilst maintaining high performance.

To illustrate this point, consider figure 1 where we show the changing requirements of two hardware structures for three applications over time, in order to maximise efficiency. The first line in each graph shows the size required for best efficiency when the pipeline width is 8 instructions. The second line shows the desired size when this is reduced to 4 instructions.

It is clear from this figure that the sizes of the issue queue and register file leading to the best efficiency vary over time. Furthermore, they are different when the width is fixed to 4 compared to a width of 8. For example, in *gap* the optimal register file size is initially 113 in both cases, but quickly needs to be adjusted to 67 when the width is 4. Conversely, for *applu* the desired size does not depend on the width. Furthermore, looking at the required issue queue size for each application is not enough to find the desired register file size. In other words, the structures' optimal sizes change over time and these changes are not necessarily correlated with one another.

This motivates the need for machine learning based control mechanisms to learn how to adapt each structure and determine the optimal configuration for the entire processor. The next section discusses our approach, then section IV gives a formal description of our model.

III. Machine Learning for Adaptivity Control

Our approach to microarchitectural adaptivity control uses a machine learning model to automatically determine the best hardware configuration for each phase of a program. Our model predicts the best parameters for the entire processor design space with only one attempt. To do this we gather hardware counters that can be used to characterise the phase and then provide them as an input to our model to guide its predictions. We first give an overview of how our scheme works, then describe the counters that we gather through dynamic profiling of each program phase.

A. Overview

Figure 2 shows an overview of how our technique works. In stage 1 the application is monitored so that we can detect when the program enters a new phase of execution. We then profile the application on a pre-defined *profiling configuration* in stage 2 to gather characteristics of the new phase. These are fed as an input into our machine learning model which gives us a prediction of the best configuration to use (stage 3). After the processor has been reconfigured we continue running the application until the next phase change is detected.

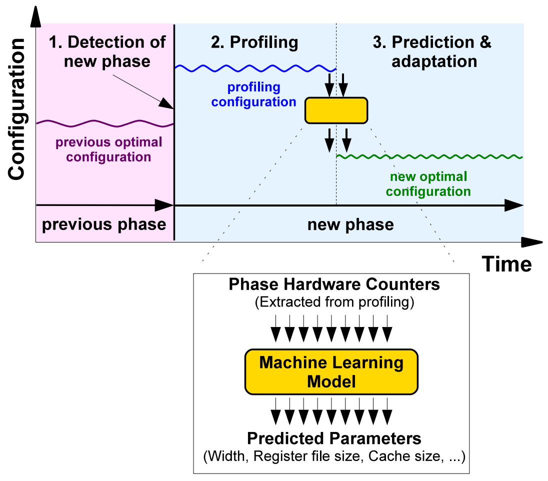

1. Detection of new phase
2. Profiling
3. Prediction & adaptation
Configuration
profiling configuration
previous optimal configuration
new optimal configuration
previous phase
new phase
Time
Phase Hardware Counters
(Extracted from profiling)
Machine Learning Model
Predicted Parameters
(Width, Register file size, Cache size, ...)

Figure 2. Overview of our technique. The hardware detects phase changes, then profiles the application on a pre-defined configuration to extract hardware counters. These are used as an input to our model that predicts the optimal microarchitectural parameters for the phase. The hardware is then reconfigured and execution continues.

Table I shows the configurable microarchitectural parameters that we have considered. It represents the design space of a high-performance out-of-order superscalar processor and is similar to spaces that other researchers have considered [1]. We vary fourteen different microarchitectural parameters across a range of values, giving a total design space of 627 billion points. The prior analysis column cites papers that have developed techniques to resize each of the structures we consider. We discuss this further in section VIII.

The main contribution of this work is a machine learning model that can accurately predict the best microarchitectural configuration to use for each program phase. We therefore focus solely on stages 2 and 3 from figure 2 in this paper. Section V describes the experimental methodology and execution environment in more detail.

B. Dynamic Profiling

To characterise each application phase we extract hardware counters from the running program. These are used as an input to our machine learning model to allow it to predict the best hardware configuration for the phase.

1) Profiling Configuration: One of the main problems with extracting hardware counters at runtime is the risk of the internal processor resources saturating: the resources can become full, causing bottlenecks in the processor. This, in turn, can hide the real resource requirements making it difficult to extract accurate information about the program's runtime behaviour. To overcome this problem we need to extract counters on a configuration that makes saturation unlikely. We therefore briefly use the microarchitectural configuration with the largest structures and the highest level of branch speculation (named the profiling configuration).

Table I
MICROARCHITECTURAL DESIGN PARAMETERS THAT WERE VARIED WITH THEIR RANGE, STEPS AND THE NUMBER OF DIFFERENT VALUES THEY CAN TAKE.

Parameter	Value Range	Num	Prior Analysis
Width	$2, 4, 6, 8$	4	[8], [9], [10]
ROB size	$32 \rightarrow 160 : 8+$	17	[11], [12]
IQ size	$8 \rightarrow 80 : 8+$	10	[11], [12], [13]
LSQ size	$8 \rightarrow 80 : 8+$	10	[3], [12]
RF sizes	$40 \rightarrow 160 : 8+$	16	[11], [12]
RF rd ports	$2 \rightarrow 16 : 2+$	8	
RF wr ports	$1 \rightarrow 8 : 1+$	8	
Gshare size	$1K \rightarrow 32K : 2*$	6	
BTB size	$1K, 2K, 4K$	3	
Branches allowed	$8, 16, 24, 32$	4	
L1 Icache size	$8K \rightarrow 128K : 2*$	5	[14], [15]
L1 Dcache size	$8K \rightarrow 128K : 2*$	5	[14], [15]
L2 Ucache size	$256K \rightarrow 4M : 2*$	5	[14], [15]
Depth (FO4 delay)	$9 \rightarrow 36 : 3+$	10	[16], [17], [18]
Total		627bn	

For each program phase we gather hardware counters on the profiling configuration. We then reconfigure to the configuration predicted by our model and run the application for that phase. Section VIII demonstrates that the cost of gathering these counters is negligible. The next section now describes the counters gathered during this profiling phase.

2) Hardware Counters: Table II gives a summary of the counters that we gather for each processor structure. They monitor the usage of each structure and the events that occur during the profile gathering phase and would therefore be simple to extract in a real implementation. We discuss their implementation in section VIII, showing that they can be gathered with low overhead.

One key aspect of our counters is the notion of a *temporal histogram*. This shows the distribution of events over time and is vital to capture the exact requirements of each structure. Each bin of the histogram stores the number of cycles that the structure has a particular usage (e.g., 100 cycles with 16 entries used, 200 cycles with 32 entries used, etc.).

Width: For the pipeline width we build a temporal histogram that keeps track of the usage frequency of each functional unit type. The histogram bins correspond directly to the number of units in use.

Queues: We use temporal histograms to collect the number of entries used in the queue on each cycle. In addition to this we add information about the average number of speculative instructions present in the queue and the number that were mis-speculated. Since our profiling configuration performs a high level of speculation, it is important to know how many of the instructions are really useful.

Register File: We use temporal histograms to summarise the number of the integer and floating point registers used. In addition, temporal histograms are used to store the usage of the read and write ports.

Table II
HARDWARE COUNTERS USED AS AN INPUT TO OUR MACHINE LEARNING MODEL.

Width
ALU usage (histogram)
Memory port usage (histogram)
Queues
Queue usage (histogram)
Speculative instructions (%)
Mis-speculated instructions (%)
Register File
Register usage (histogram)
Read port usage (histogram)
Write port usage (histogram)
Caches
Stack distance (histogram)
Block reuse distance (histogram)
Set reuse distance (histogram)
Reduced set reuse distance (histogram)
Branch predictor
BTB reuse distance (histogram)
Branch mis-prediction rate (%)
Pipeline depth
Cycles per instruction

Caches: We use temporal histograms representing stack distance [19], [20] and reuse distance. Each bin corresponds to a specific distance. Intuitively the stack distance is important since it characterises the capacity usage of the cache. We also estimate the potential conflicts that could arise if the cache size were smaller in the Reduced set reuse distance histogram. To do this we map the sets to those of the smallest cache size (as though "emulating" the smallest cache size available).

Branch Predictor: We use the access reuse distance within the BTB, which is similar to the block reuse distance in the caches. The second counter corresponds to the branch mis-prediction rate which is useful to control the degree of speculation within the processor.

Pipeline Depth: We only need the average number of instructions executed per cycle over the entire phase.

C. Example

This section gives an example of how the hardware counters are used to determine the size of the load/store queue that will lead to the best energy efficiency value. Figure 3 shows the efficiency values and counters extracted from phases within four different programs. For each figure, the top graph shows the relative efficiency of the processor when the load/store queue size is varied. By choosing the best configuration for this phase from our training data (described in section V-C), we can determine the optimal values for all other parameters. To obtain maximum efficiency, the size of the load/store queue for *mgrid* should be 32, *swim* 72, *parser* 16 and *vortex* 16. Underneath are the counters gathered. The queue usage histogram on the left has bins corresponding to queue sizes. On the right is the average number of speculative instructions in the queue and the fraction that were mis-speculated.

For *mgrid* and *swim* we see the best queue size directly corresponds to the observed usage during the profiling phase. For these applications there are few mis-speculated instructions (mis-spec) present in the queue during the phase.

Now consider *parser* and *vortex* which both have a significant number of mis-speculated instructions. This time the largest bin in the queue usage histogram counter is 8 which does not directly correspond to the size of the queue that maximises efficiency. Instead, the best size of the queue is 16 entries in both cases. Since these programs have similar counters and the same desired queue size, our model can "learn" this information. So, after training on *parser*, it can make the correct prediction when it sees the same counters again in *vortex*.

The next section shows how these counters can be used to build a model that makes a single prediction of the best hardware configuration to use for this phase.

IV. MODELLING GOOD MICROARCHITECTURAL CONFIGURATIONS ACROSS PROGRAM PHASES

In order to build a model that predicts good microarchitectural configurations across program phases we require examples of various microarchitectural configurations on different program phases and their corresponding performance metrics (e.g., their energy-efficiency values). Additionally, we require a program phase to be characterised by a set of hardware counters described in the previous section.

Let $\{X^{(j)}\}_{j=1}^{M}$ be the set of training program phases and $\{\mathbf{x}^{(j)}\}_{j=1}^{M}$ be their corresponding D-dimensional vector of counters. For each of these program phases we record the performance on a set of N distinct microarchitectural configurations $\{\mathbf{y}^{(i)}\}_{i=1}^{N}$. Each component of a microarchitectural configuration $\mathbf{y}$ is a single microarchitectural parameter y_a with $a = 1, \dots, A$, with A representing the number of architectural parameters (14 in this paper). Given a new program phase X^* described by a set of counters $\mathbf{x}^*$, we aim to predict a set of (good) microarchitectural parameters $\mathbf{y}^*$ that are expected to lead to the highest energy-efficiency.

A. The Model

Our goal is to build a model that correctly captures the relationship between program phases' hardware counters and good microarchitectural configurations. In other words, we aim to learn a mapping $f : \mathcal{X} \rightarrow \widetilde{Y}$ from the space of program phase counters $\mathcal{X}$ to the space of good microarchitectural configurations $\widetilde{Y}$.

In order to achieve this we model the conditional distribution $P(\tilde{\mathbf{y}}|\mathbf{x})$ of good microarchitectural configurations $\tilde{\mathbf{y}}$

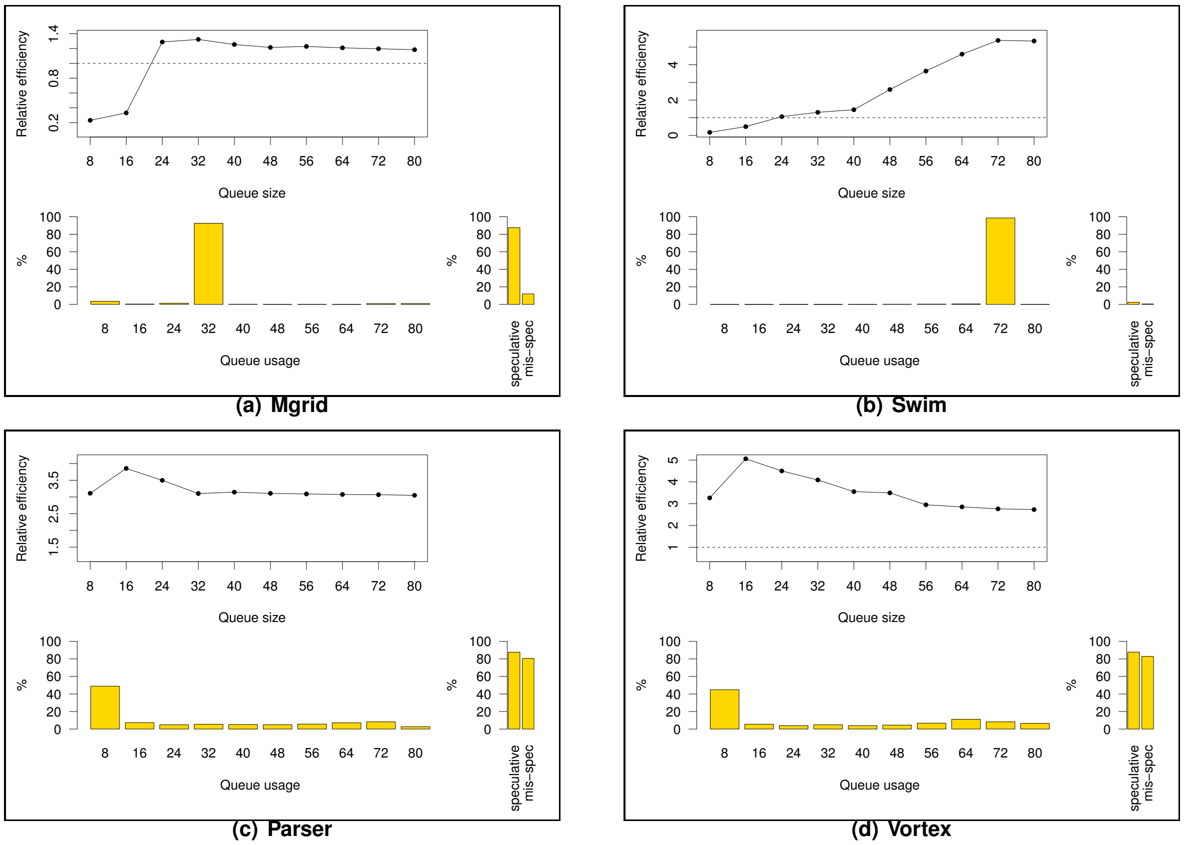


Figure 3. Load/store queue counters for four phases from different programs. We also show the relative efficiency achieved when varying the load/store queue parameters on the best configuration found (higher is better).

given a set program phase's counters $\mathbf{x}$. In our approach we consider each microarchitectural parameter to be conditionally independent given the counters:

$$P(\tilde{\mathbf{y}}|\mathbf{x}) = \prod_{a=1}^{A} P(\tilde{y}_a|\mathbf{x}). \tag{1}$$

It is important to note that there are dependencies between microarchitectural parameters. However, our model assumes that *good* parameters are *conditionally* independent given the program phase's counters, rather than assuming marginal independence between parameters.

B. Predictions

Given the learnt model, we can predict a set of expected good microarchitectural configurations $\mathbf{y}$ on a new program phase $\mathbf{x}^*$ by determining the most likely configuration under the learnt distribution:

$$\mathbf{y}^* = \underset{\tilde{\mathbf{y}}}{\operatorname{argmax}}\, P(\tilde{\mathbf{y}}|\mathbf{x}^*), \tag{2}$$

where we note that, due to conditional independence, this reduces to computing the value of each $\tilde{y}_a$ that maximises each single distribution $P(\tilde{y}_a|\mathbf{x})$.

C. Model Parametrisation

In our model the conditional distribution of each microarchitecture parameter $\tilde{y}$ (where we omit the subindex a for clarity) given a set of counters $\mathbf{x}$ is described by a soft-max function:

$$P(\tilde{y} = s_k|\mathbf{x}) = \sigma_k(\mathbf{x}, \mathbf{W}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\sum_{j=1}^{K} \exp\left(\mathbf{w}_j^T \mathbf{x}\right)}, \tag{3}$$

where $P(\tilde{y} = s_k|\mathbf{x})$ denotes the probability of microarchitectural parameter $\tilde{y}$ having the value s_k (out of K possible values) given the program phase's counters $\mathbf{x}$; and the $D \times K$ matrix of weights $\mathbf{W}$ are the model parameters where each column $\{\mathbf{w}_k\}_{k=1}^{K}$ corresponds to a set of weights one for each value $\tilde{y}$ can take on[1].

D. Model Learning

In order to learn the parameters of the model our approach is based upon likelihood maximisation. For clarity, we focus on a single microarchitectural parameter y which can take one out of K possible values as we can learn the model parameters for each architectural parameter independently.

[1]Other approaches were tried and we found that a soft-max model led to the best results.

The data likelihood is given by:

$$L(\mathbf{W}) = \prod_{n=1}^{\tilde{N}} \prod_{k=1}^{K} P(\tilde{y}^{(n)} = s_k | \mathbf{x}^{(n)})^{\delta(y^{(n)}=s_k)}, \quad (4)$$

where $\mathbf{x}^{(n)}$ is the vector of counters corresponding to architecture configuration $\tilde{y}^{(n)}$ and $\delta(y^{(n)} = s_k)$ is an indicator function that is 1 only when the particular architecture parameter on data-point n $(y^{(n)})$ takes on the value s_k and zero otherwise. Additionally, we have introduced a new symbol $\tilde{N}$ denoting the number of good architecture configurations. In our experiments we have selected the set of good configurations to be those that are within 5% of the best empirical performance.

By taking the logarithm of equation (4) and using equation (3) the expression for the data log-likelihood that we aim to maximise is:

$$\mathcal{L} = \sum_{n=1}^{\tilde{N}} \sum_{k=1}^{K} \delta(\tilde{y}^{(n)} = s_k) \log \sigma_k(\mathbf{x}^{(n)}, \mathbf{W}). \quad (5)$$

We note that a naïve maximum likelihood approach can lead to severe over-fitting. Hence we have considered a regularised version of the data log-likelihood by adding a term to penalise large weights, preventing over-fitting:

$$\mathcal{L}^{\text{POST}} = \mathcal{L} + \lambda \text{ tr } (\mathbf{W}^T\mathbf{W}), \quad (6)$$

where tr $(.)$ denotes the trace operator and λ is the regularisation parameter.

Thus, the optimal solution to the weight parameters is obtained with:

$$\mathbf{W}^{\text{Reg}} = \underset{\mathbf{W}}{\text{argmax}}(\mathcal{L}^{\text{POST}}) \quad (7)$$

Training our model means finding the solution for $\mathbf{W}^{\text{Reg}}$. This can be done by using conjugate gradient optimisation with a deterministic initialisation of all the weights to 1 and with $\lambda = 0.5$. See [21] for more information.

E. Prediction

To make predictions, only equations (2) and (3) need to be considered because the training is performed off-line. Let us assume that we are concerned with making predictions on single architecture parameter and that this parameter may take on one out of K possible values. Additionally, lets say that the corresponding model parameters are denoted by the $D \times K$ matrix $\mathbf{W}$. Hence, the computations involved for a new program phase characterised by the $D \times 1$ vector of counters $\mathbf{x}^*$ are:

$$\mathbf{b} = \mathbf{W}^T\mathbf{x}^* \quad (8)$$

$$y^* = \underset{k}{\text{argmax}}(b_1, \ldots, b_K), \quad (9)$$

where we have avoided the exponentiation in equation (3) by realising that, at prediction time, we can make a *hard* decision without computing the probabilities explicitly.

V. Experimental Methodology

This section presents the simulator and benchmarks used. We also describe how we gathered our training data and the methodology used to evaluate our technique.

A. Simulator and Benchmarks

Our cycle-accurate simulator is based on Wattch [22], an extension to SimpleScalar [23]. We altered Wattch's underlying Cacti [24] models to updated circuit parameters. We also removed the SimpleScalar RUU and added a reorder buffer, issue queue and register files. To make our simulations as realistic as possible we used Cacti to accurately model the latencies of the microarchitectural components as they varied in size. To avoid errors resulting from cold structures, we warmed the caches and branch predictor for 10 million instructions before performing each detailed simulation.

To evaluate our technique, we used all 26 SPEC CPU 2000 benchmarks [25] compiled with the highest optimisation level. We ran each benchmark using the *reference* input set. We extracted 10 phases per program using SimPoint with an interval size of 10 million instructions.

B. Performance Metric

We have evaluated the results of our predictor using energy efficiency as a metric, measured as $[ips^3/Watt]$ where $[ips]$ is the number of instructions executed per second and $[Watt]$ is the power consumption in Watts. This metric represents the trade-offs between power and performance, or the efficiency of each design point. It is widely used within the architecture community [26] to indicate how efficient a configuration is at converting energy into processing speed.

C. Gathering the Training Data

As seen in section IV, we need to gather data to train our model and find good solutions within our design space. To achieve this we first searched the design space by uniformly sampling 1000 random configurations. We found the best configuration for each phase, then randomly chose 200 local neighbour configurations. Finally, we repeated this by choosing the best out of the 1,200 for each phase and altered each parameter one at a time to each of its possible values. This totals 1,298 simulations per phase, or more than 300,000 in total. In addition, the results of the search were also used to approximate the best possible performance achievable per phase.

D. Evaluation Methodology

With this data we built our model and evaluated it using leave-one-out cross-validation. This is standard machine learning methodology that ensures that when we present results for a specific program, our model has never been trained with it.

To evaluate our technique, we proceed in three stages. We first characterise the current program phase by running part

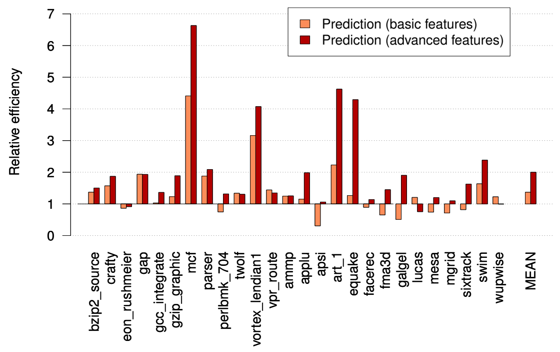


Figure 4. Energy-efficiency $[ips^3/Watt]$ achieved by our model compared to the best overall static configuration for SPEC CPU 2000 (higher is better). Two different sets of hardware counters were used with our model: the basic counters are made of the standard performance counters available on current processors while the advanced ones use the new temporal histogram counters.

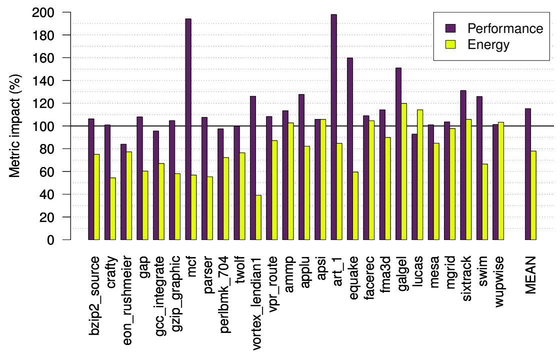


Figure 5. Performance and energy breakdown for our model when using the advanced features compared to the best overall static configuration. On average performance is improved by 15% and energy reduced by 21%.

of it on the profiling configuration in order to gather the hardware counters. We use our model to make a prediction and then continue execution of the current phase with the configuration supplied by our model. We repeat this process for all the program's phases extracted.

VI. Results

This section presents the results of our technique, compared against a baseline static processor configuration.

A. Baseline Configuration

In order to determine a suitable baseline, we examined all the architecture configurations in our sample space and selected the static configuration that led to the best energy-efficiency on average across the benchmarks. This represents the best achievable with a single fixed static hardware configuration and is an aggressive baseline. Table III shows its configuration.

B. Results with two Hardware Counter Sets

In this section we evaluate the gains achievable with our technique across the benchmark suite for two sets of hardware counters. The first is composed of standard performance counters available in current processors. This includes average queue occupancy, number of ALU operations, average register file usage, cache access and miss rates, branch predictor access and miss rates, and average number of instructions per cycle. The second set of counters corresponds to the more advanced features presented in section III-B2 that includes temporal histograms.

Figure 4 shows the energy-efficiency improvement achieved by our approach relative to the baseline configuration for the two counter sets. When compared to the best static hardware we achieve on average a factor 2x improvement in energy-efficiency with the advanced counter set. In some cases we achieve over 4x the performance of the best static hardware for *vortex*, *art*, *equake* and up to 6.5x for *mcf*. Only in two cases is the best static configuration slightly better than our approach: *eon* and *lucas*.

With the basic counter set, our model only achieves 1.3x average improvement over the best overall static configuration. For several benchmarks, the performance is significantly below that of the advanced counters. This shows that the more advanced set of counters is necessary in order to achieve good performance.

C. Breakdown in Performance and Energy

Having seen the results for the combined efficiency metric, we now look at the breakdown in terms of performance $[ips]$ and energy $[Joules]$. Figure 5 shows these two metrics individually compared to the best overall static configuration. On average we observe a 15% increase in performance and a 21% decrease in energy. For some benchmarks such as *crafty*, the model achieves a remarkable 48% cut in energy while maintaining the same performance as the baseline configuration. The model detects that the L2 cache and the register file are not being fully utilised and reduces their correspoding size to 256K and 64 respectively. In other cases, such as *art*, the model decreases the energy consumption by 15% while at the same time increasing performance by a factor 2. In this case, the model increases the issue width and the number of read/write ports to the register files and at the same time decreases the size of the instrution cache to achieve lower energy consumption. This clearly shows that our approach of driving adaptivity with a predictive model can offer large benefits to these applications. They would otherwise exhibit poor energy-efficiency had we use a fixed static configuration tuned for the average case.

VII. Analysis of the Accuracy of the Model

In this section we evaluate the accuracy of our approach in predicting the best configuration for each phase of the applications. We also present an analysis of the model performance at a phase level and show how architectural configurations vary with program phases.

Table III
THE CONFIGURATION OF OUR BASELINE ARCHITECTURE.

Width	ROB	IQ	LSQ	RF	RF rd	RF wr	Gshare	BTB	Branches	Icache	Dcache	Ucache	Depth
4	144	48	32	160	4	1	16K	1K	24	64K	32K	1M	12

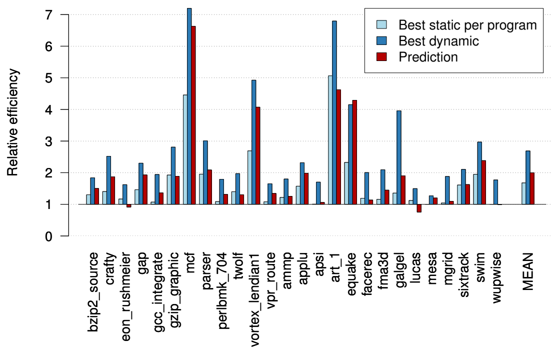


Figure 6. Energy-efficiency achieved by our model for all of SPEC CPU 2000 compared to the best static configuration tailored for each program and compared with the best dynamic configuration tailored for each program's phase. All the values are normalised by the best overall static configuration (higher is better).

A. *Comparison Against* **Specialised** *Static Configurations*

Although our approach clearly outperforms any fixed static configuration, having different specialised static configurations for each program may be considered an attractive alternative. This approach is used for domain specific processors such as DSPs and GPUs. Figure 6 shows the performance of our technique relative to the best specialised static configuration found in our sample space for that program. Clearly such an approach cannot be applied to "unseen" programs and is not viable for general-purpose computing. Nonetheless, it gives an important limit evaluation of our approach.

On average, a specialised static configuration gives a factor 1.5x improvement compared to the factor 2x of our approach. It is guaranteed never to perform worse than the best average static configuration so does not suffer performance loss in *lucas* and *eon*. Conversely, it is unable to exploit those cases where there is significant improvement available, e.g., *mcf* and *equake*, due to the large intra-program dynamic phase variation.

B. *Comparison Against* **Ideal** *Dynamic Configurations*

We now wish to determine how far our model is from the upper bound on efficiency. For this purpose we consider a scheme that has the ability to adapt the microarchitecture on a per-phase basis with full knowledge about how the application and architecture will perform. Therefore we selected, offline, the best configuration from the sample space for each phase of each program and then ran each phase with its corresponding ideal configuration (best dynamic) leading to maximum energy-efficiency.

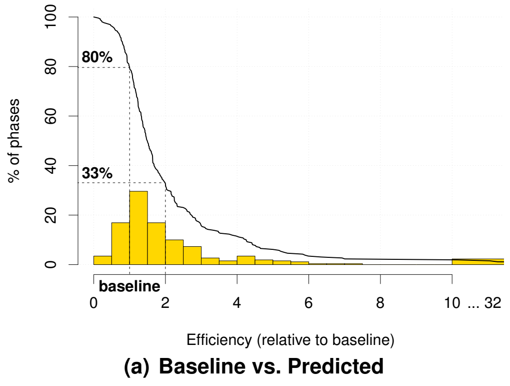
As can be seen in figure 6, on average this ideal setup gives an improvement of 2.7x over the best fixed static configuration. In some cases, like *mcf*, this improvement is more than 7x. Even in the worst case, *eon*, there is an improvement of 1.5x over the static baseline. As seen our technique gives an average improvement of 2x, thus achieving 74% of the available improvement. Generally the performance of our approach tracks the maximum available. In the case of *galgel*, however, there is a 4x improvement available, yet we achieve only a factor 2x, showing there is still room for improvement.

C. *Accuracy of Our Approach on a Phase Basis*

This section evaluates the accuracy of the predictive model on a per phase basis. Figure 7(a) shows two graphs overlaid. The first is a histogram representing the distribution of the efficiency values for the 260 phases. The x-axis shows the improvement achieved for a particular phase relative to the baseline. The y-axis represents the percentage of phases with a specific efficiency value. So, for example. the largest bin has an efficiency between 1x and 1.5x of the baseline and corresponds to approximately 30% of the phases. As in the previous section, the efficiency values are normalised according to the baseline (i.e., the best overall static configuration).

To determine how often we are better (or worse) than the baseline and by how much, we can look at the continuous line on the graph which is the ECDF (Estimated Cumulative Distribution Function). It shows how often our approach achieves at least a certain efficiency improvement. For example we see that our model predicts a configuration better than the baseline for 80% of the phases. We also notice that for approximately 33% of the phases the predicted configuration has an efficiency of at least two times that of the baseline. There are even a small number of phases that achieve improvement of 32 times the baseline.

Although it is important to evaluate our approach relative to the best static configuration, it is equally important to compare its accuracy against the best dynamic configurations found in the sample space for each phase as shown in figure 7(b). The best configuration has a value of 1. If the performance of the predicted configuration is lower than 1, it means that it is less efficient. A value greater than 1, although surprising at first, indicates that the prediction is actually better than the best found in the sample space. This can occur because the best was not established by using an exhaustive search of the entire space.



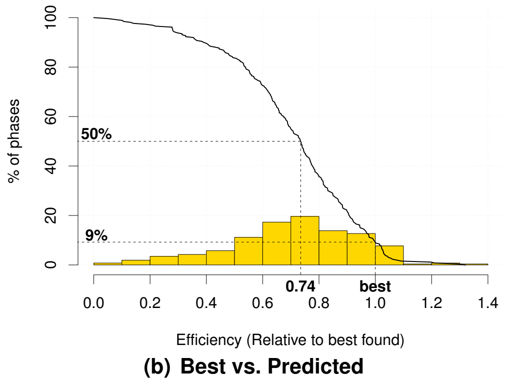


Figure 7. Histograms showing the distribution of energy-efficiency values for the 260 different phases extracted from SPEC 2000 when compared to the baseline (a) and the best (b). In addition the ECDF (estimated cumulative distribution function) is represented by the solid line. The values are accumulated from the right.

We notice that 50% of the phases achieved at least 74% of the efficiency of the best configuration. In other words, on average, we expect our model to achieve 74% of the maximum available (confirming earlier results). Interestingly, for about 9% of the phases, the predicted configuration actually performs better than the best found using a thousand samples. This provides evidence that our model can actually predict very efficient parameters.

D. Architecture Configuration Variation

We now want to show how architectural configurations affect the efficiency of the overall processor design. Due to space considerations we only present results for three out of the fourteen microarchitectural parameters.

Figure 8 shows the distribution of efficiency values for our 260 phases as violin diagrams for the width, instruction queue, and instruction cache. These graphs show what happens when the considered parameter is fixed to a specific value and all others are allowed to vary in order to find the highest-efficiency configuration for each phase. This best efficiency value is recorded on the graph for each phase and the distribution of these values represented by the violin (the thicker the violin, the more phases are concentrated around that value). The % value on top, shows the percentage of phases for which that fixed hardware parameter is best. For instance, in the case of processor width (figure 8(a)), a width of 2 is best in 22% of cases, while a width of 4 is best in 32% of cases.

By observing these graphs it is clear that there is no single parameter value that is good for all phases. Considering the issue queue for instance (figure 8(b)), we see that a size of 72 is only optimal for 34% of the phases. However, for 25% of the phases, those below the quantile black line, this value would mean that the best achievable would be 0.6 that of the optimal (i.e., 40% less efficient). In addition we see that the efficiency of some phases can drop to 0.3, the extreme lower point of the violin's distribution.

Looking at the instruction cache in figure 8(c) we see that a small size (64 sets) is optimal for 28% of the phases. It is also the value that gives the highest median (white dot) at about 0.9 from the optimal. So if a designer was to choose a static architecture, this could be a good candidate. However, the smallest size is also the one that corresponds to the lowest efficiency for some phases. We conclude that there isn't a one-fits-all approach and shows the challenges in building predictors for microarchitectural adaptivity.

Table IV
NUMBER OF SETS SAMPLED FOR EACH CACHE PER FEATURE TYPE.

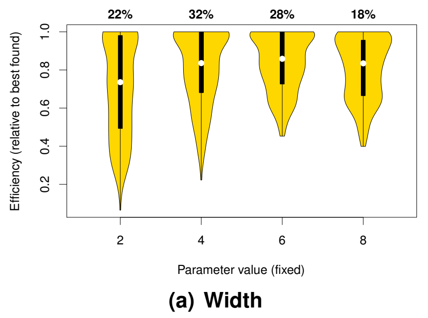
Feature Type	Insn. cache	Data cache	L2 cache
Set reuse distance	256	4	16
Blk reuse distance	16	128	32

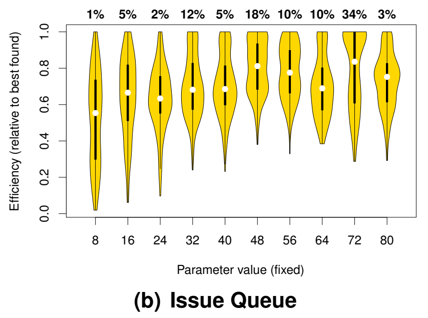
VIII. IMPLEMENTATION ANALYSIS

This section describes how our technique could be implemented in an actual processor design. We have evaluated the costs of gathering our hardware counters and performing reconfiguration to demonstrate that our approach can be implemented at low cost and with few overheads.

Gathering Hardware Counters: The construction of our temporal histograms is the main overhead when gathering our hardware counters. However, an efficient implementation is feasible. Since the caches contain the most complex histograms and consume the largest fraction of total processor power, they represent an upper bound on the overheads necessary to characterise program behaviour. The block and set reuse histograms are the most costly to gather. For each block the former requires two timestamps (to record the time the block was brought into the cache and the last hit), and a hit counter. The latter requires a hit counter per set.

We have used dynamic set sampling [27] to reduce the number of sets and blocks that need monitoring in order to build these histograms. We ran the profiling configuration on all program phases and determined the optimum number of sets that need to be sampled to maintain high prediction accuracy. The results are shown in table IV. For example,





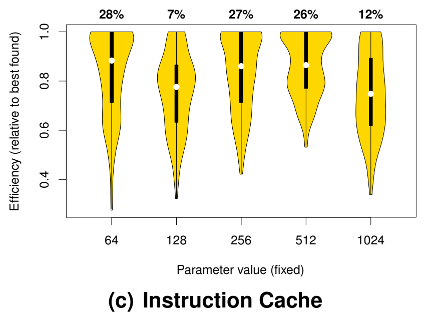


Figure 8. Distribution of the highest energy-efficiency achievable for the 260 phases when the value of one parameter is fixed and the rest of the parameters are allowed to vary. For each parameter's value the white central dot represent the median efficiency value achievable in the phases and the black rectangle shows the two quartiles, where 50% of the data lies. The % value on top, shows the percentage of phases for which that fixed hardware parameter is best.

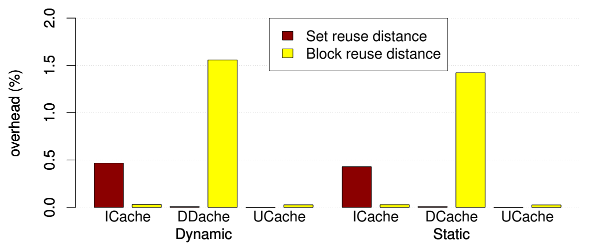


Figure 9. Energy overheads of extracting the set and block reuse distance for each cache. The maximum overhead for the dynamic energy is 1.55% in the case of the data cache when extracting the block reuse distance. For the static energy, a maximum overhead of 1.4% is reached.

to gather the data cache's set reuse distance histogram we only need to sample four sets.

Figure 9 shows how this translates into energy overheads. The maximum dynamic energy overhead is 1.6% when extracting the block reuse histogram from the data cache, which also incurs a leakage energy overhead of 1.4%. However, these overheads are only required when running the profiling configuration. We have verified experimentally that reconfiguration occurs once every 10 intervals, on average. Therefore, the overall overheads of gathering these counters become almost insignificant. These results show that gathering our hardware counters is cost-effective considering the efficiency savings that our model achieves.

Resource Reconfiguration: Adaptation can be achieved through the use of simple bitline segmentation of processor structures [12], [13]. This allows partitions to be turned off in isolation. We have modelled this within our simulator, allowing a 200ns delay to power up 1.2 million transistors [28]. In addition, we have accurately modelled the delays required to flush caches and stall the pipeline when resources need reconfiguration. Table V shows the results.

We see that the branch predictor is the quickest to reconfigure at 154 cycles whereas the L2 cache takes the longest at almost 20,000 cycles. However, the majority of this time is hidden as transistors can be powered up and down whilst the resource is still being used. Our results shows that the overall

Table V
OVERHEADS OF RECONFIGURING EACH STRUCTURE IN CYCLES.

Processor structure	Cycle overhead
Width	443
RF	487
Bpred	154
ROB	255
IQ / LSQ	234 / 275
ICache / DCache	478 / 620
UCache	18322

performance penalty when reconfiguration occurs is just 3% for one interval and the energy overheads are also 3%. However, since reconfiguration only occurs once every 10 intervals, the overheads for the whole phase are significantly reduced. This shows that reconfiguring processor resources can be achieved with very few overheads that are amortized over the execution of the whole phase.

Model: Work by Jiménez and Lin [29] has shown how to build a perceptron-based neural branch predictor. At prediction time, our technique can be seen as a multi-class generalisation of the perceptron. We can therefore use a low-overhead version of their proposed circuit-level implementation, since our approach does not need to be trained online. This can be achieved, for example, by using 8bit signed integers for the weights ($\mathbf{W}$). Since we have approximately 2000 of these, this would require 2KB of storage. Given that the model is only employed once every 10 intervals, on average, we estimate the runtime overheads to be insignificant.

IX. Prior Work on Microarchitectural Adaptivity

Recently, Lee and Brooks [1] showed that it is possible to significantly increase processor energy efficiency by adapting it as a program is running. Our work takes this a step further and shows that it is possible to build a model that can automatically drive the adaptation process.

Adaptive Processor Structures: Many researchers have examined how processor structures can be made adaptive.

The last column of table I summarises this information. In particular the issue queue [2], [3], [11], [12], [13], [30], reorder buffer [11], [12], register files [11], [12], pipeline [16], [8] and caches [14], [30] have been studied.

Dhodapkar and Smith [31] focused on control mechanisms by assessing the use of working set signatures to detect changes in behaviour of the program. Liang, et. al. and Tiwari, et. al. separately proposed variable latency architectures where additional stages can be added to the pipeline to combat process variations [17], [18].

However, these studies considered only a limited adaptivity scope and looked at each of the components of the processor in isolation using control mechanisms based on simple heuristics. More recently a table-driven technique [32] was proposed to reduce peak power in an adaptive processor. In comparison, our work considers varying all these parameters together and uses a machine learning model to control the adaptation process.

Multicore Adaptivity: For multicore processors, Mai et. al. illustrated an adaptive memory substrate and its flexibility when implementing very different architectures named "Smart Memories" [15]. Later Sankaralingam et. al. proposed the TRIPS architecture [33], Ipek et. al. "Core Fusion" [9] and Tarjan et al. "Core Federation" [10]. These last two approaches merge simple cores together in order to create a wide superscalar processor.

Software-Controlled Adaptivity: Several researchers have looked at adaptivity control from the software side. Hughes et. al. [8] looked at multimedia applications characterised by repeated frame processing. Hsu and Kremer [34] implemented a compiler algorithm that adapts the voltage and frequency based on the characteristics of the code. Later Wu et. al. [35] looked at adapting the voltage within the context of a dynamic compilation framework which can monitor and transform the program as it is running. Huang et. al. [36] proposed using subroutines as a natural way to decide when to reconfigure the processor. Finally Isci et. al. [37] developed a real system framework that predicts program phases on the fly to guide dynamic voltage and frequency scaling.

Heuristic-Driven Schemes: Prior work [3] has also considered controlling adaptivity by looking at hardware counters extracted at runtime. However, they make use of a heuristic for search at runtime whereas we directly predict the best configuration using a machine learning model. Furthermore, they only focus on three processor queues whereas we consider 14 parameters at once.

Runtime Exploration: Other researchers looked at learning the space at runtime [38], [39]. In our context it is undesirable to perform any sort of runtime exploration since this would inevitably result in visiting poorly-performing configurations and reduce the overall efficiency.

Predictive Models: Recently, Ipek et. al. [5], Lee and Brooks [7] and Joseph et al. [6] proposed predictive modelling (i.e., machine learning) for architectural design space exploration. These models predict the design space of a whole program for various architecture configurations, thus enabling the efficient exploration of large design spaces. However, these are limited to whole program modelling and must first be trained for each application needing prediction. Furthermore, they are not directly usable within the context of dynamic adaptation since they would require a search of the design space at runtime.

Phase Detection: Phase detection techniques are at the core of any dynamic adaptive system and have been extensively studied previously. The work from Dhodapkar and Smith [40] offers a good comparison between many proposed techniques. There are a number of examples of online phase detection techniques in the literature that rely on basic block vectors [41], instruction working sets [31] or conditional branch counts [14], for example. Wavelet analysis has also gained some attention [42], [43].

X. Conclusion and Future Directions

This paper has proposed a novel technique for dynamic microprocessor adaptation that differs substantially from prior work. We built a machine-learning model to predict the best configuration that uses hardware counters collected at runtime. We have introduced the notion of a temporal histogram and shown that our model is able to perform much better using these than conventional performance counters. By using our model to drive adaptivity we were able to double the energy-efficiency over the best overall static configuration. This represents 74% of the best that achievable within our sampled space.

In this work we have assumed a fixed profiling period and that all resources are adapted at the same time. Given a hardware substrate capable of reconfiguring itself at different frequencies for each resource, the challenge will be to find the degree of adaptation suitable for each hardware structure.

Finally, this paper has targeted a uniprocessor design. However, the technique presented can be directly applicable in the context of a multicore processor. If each of the cores could implement our scheme and dynamic adapt to their own workloads, this would lead to true heterogeneity; the key to high energy-efficiency. In this scenario a possible extension to this work could be to look at the implications of resource sharing when driving adaptivity.

Acknowledgements

This work was supported by the Royal Academy of Engineering and EPSRC. It has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF). (http://www.ecdf.ed.ac.uk/). The ECDF is partially supported by the eDIKT initiative (http://www.edikt.org.uk). NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] B. Lee and D. Brooks, "Efficiency trends and limits from comprehensive microarchitectural adaptivity," in *ASPLOS*, 2008.

[2] D. Folegnani and A. Gonzalez, "Energy-effective issue logic," in *ISCA*, 2001.

[3] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources," in *MICRO*, 2001.

[4] C. Dubach, T. Jones, and M. O'Boyle, "Microarchitectural design space exploration using an architecture-centric approach," in *MICRO*, 2007.

[5] E.Ipek, S.A.McKee, B. de Supinski, M. Schulz, and R. Caruana, "Efficiently exploring architectural design spaces via predictive modeling," in *ASPLOS*, 2006.

[6] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *MICRO*, 2006.

[7] B. Lee and D. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS*, 2006.

[8] C. Hughes, J. Srinivasan, and S. Adve, "Saving energy with architectural and frequency adaptations for multimedia applications," in *MICRO*, 2001.

[9] E. Ipek, M. Kirman, N. Kirman, and J. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *ISCA*, 2007.

[10] D. Tarjan, M. Boyer, and K. Skadron, "Federation: Repurposing scalar cores for out-of-order instruction issue," in *DAC*, 2008.

[11] J. Abella and A. González, "On reducing register pressure and energy in multiple-banked register files," in *ICCD*, 2003.

[12] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semerano, G. Magklis, and M. L. Scott, "Integrating adaptive on-chip storage structures for reduced dynamic power," University of Rochester, Tech. Rep., 2002.

[13] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook, "A circuit level implementation of an adaptive issue queue for power-aware microprocessors," in *GLSVLSI*, 2001.

[14] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *MICRO*, 2000.

[15] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *ISCA*, 2000.

[16] A. Efthymiou and J. Garside, "Adaptive pipeline structures for speculation control," in *ASYNC*, May 2003.

[17] X. Liang, G.-Y. Wei, and D. Brooks, "Revival: Variation tolerant architecture using voltage interpolation and variable latency," in *ISCA*, 2008.

[18] A. Tiwari, S. Sarangi, and J. Torrellas, "Recycle: Pipeline adaptation to tolerate process variation," in *ISCA*, 2007.

[19] K. Beyls and E. H. D'Hollander, "Reuse distance as a metric for cache behavior," in *PDCS*, 2001.

[20] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *PLDI*, 2003.

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

[22] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *ISCA*, 2000.

[23] D. Burger and T. Austin, "The simplescalar tool set, version 2.0." University of Wisconsin, Tech. Rep. TR-1342, 1997.

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

[25] J. Henning, "Spec cpu2000: Measuring cpu performance in the new millenium," *IEEE Computer*, 2000.

[26] A. Hartstein and T. R. Puzak, "Optimum power/performance pipeline depth," in *MICRO*, 2003.

[27] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *ISCA*, 2006.

[28] P. Royannez, H. Mair, F. Dahan, M. Wagner, M. Streeter, L. Bouetel, J. Blasquez, H. Clasen, G. Semino, J. Dong, D. Scott, B. Pitts, C. Raibaut, and U. Ko, "90nm low leakage soc design techniques for wireless applications," in *ISSCC*, 2005.

[29] D. A. Jiménez and C. Lin, "Neural methods for dynamic branch prediction," *ACM Trans. on Computer Systems*, vol. 20, 2002.

[30] D. Albonesi, "Dynamic ipc/clock rate optimization," in *ISCA*, 1998.

[31] A. Dhodapkar and J. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *ISCA*, 2002.

[32] V. Kontorinis, A. Shayan, D. M. Tullsen, and R. Kumar, "Reducing peak power with a table-driven adaptive processor core," in *Micro*, 2009.

[33] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *ISCA*, 2003.

[34] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction," in *PLDI*, 2003.

[35] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks, "A dynamic compilation framework for controlling microprocessor energy and performance," in *MICRO*, 2005.

[36] M. Huang, J. Renau, and J. Torrellas, "Positional adaptation of processors: Application to energy reduction," in *ISCA*, 2003.

[37] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *MICRO*, 2006.

[38] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO*, 2008.

[39] S. Choi and D. Yeung, "Learning-based smt processor resource distribution via hill-climbing," in *ISCA*, 2006.

[40] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *MICRO*, 2003.

[41] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *ISCA*, 2003.

[42] C.-B. Cho, W. Zhang, and T. Li, "Informed microarchitecture design space exploration using workload dynamics," in *MICRO*, 2007.

[43] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *ASPLOS*, 2004.