# Dynamic Microarchitectural Adaptation Using Machine Learning

CHRISTOPHE DUBACH, University of Edinburgh, UK (Member of HiPEAC)
TIMOTHY M. JONES, University of Cambridge, UK (Member of HiPEAC)
EDWIN V. BONILLA, NICTA & Australian National University, Australia

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 article 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 by 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 know the best microarchitecture for each program phase ahead of time. In addition, we present an extended analysis of the best configurations found and show that the overheads associated with the implementation of our scheme have a negligible impact on performance and power.

Categories and Subject Descriptors: B.8.2 [**Performance Analysis and Design Aids**]; I.6.5 [**Model Development**]; C.4 [**Performance of Systems**]; C.1.3 [**Other Architecture Styles**]: Adaptable Architectures

General Terms: Performance

Additional Key Words and Phrases: Machine learning, processor adaptation, performance analysis

**ACM Reference Format:**
Dubach, C., Jones, T. M., and Bonilla, E. V. 2013. Dynamic microarchitectural adaptation using machine learning. ACM Trans. Architec. Code Optim. 10, 4, Article 31 (December 2013), 28 pages.
DOI: http://dx.doi.org/10.1145/2541228.2541238

## 1. 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 achievable power savings. With high adaptivity, the processor is able to vary many

Author's address: Christophe Dubach; email: christophe.dubach@ed.ac.uk; Timothy M. Jones; email: timothy.jones@cl.cam.ac.uk; Edwin V. Bonilla; email: edwin.bonilla@nicta.com.au.

DOI: http://dx.doi.org/10.1145/2541228.2541238

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 [Lee and Brooks 2008], 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 [Folegnani and Gonzalez 2001; Ponomarev et al. 2001] will not scale to a large number of configurations. Other prior works have used statistical machine learning to construct models that estimate the performance and/or power as a function of the microarchitectural configuration [Dubach et al. 2007; Ipek et al. 2006; Joseph et al. 2006; Lee and Brooks 2006]. 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 lightweight, runtime control mechanisms.

This article 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 offline 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 that 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 article is structured as follows. Section 2 motivates the use of machine learning for adaptivity. Section 3 then describes our approach to dynamic adaptation using a model explained in Section 4. Section 5 presents the experimental setup, and Section 6 evaluates our approach. Section 7 investigates model accuracy, and Section 8 discusses the properties of the best configurations found. Section 9 describes implementation details, and Section 10 offers a phase granularity study. Finally, Section 11 describes related work and then Section 12 concludes.

## 2. THE NEED FOR MACHINE LEARNING–BASED CONTROL

This article proposes a novel technique for dynamic microprocessor adaptation that differs substantially from prior work. Existing schemes, described in Section 11, 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

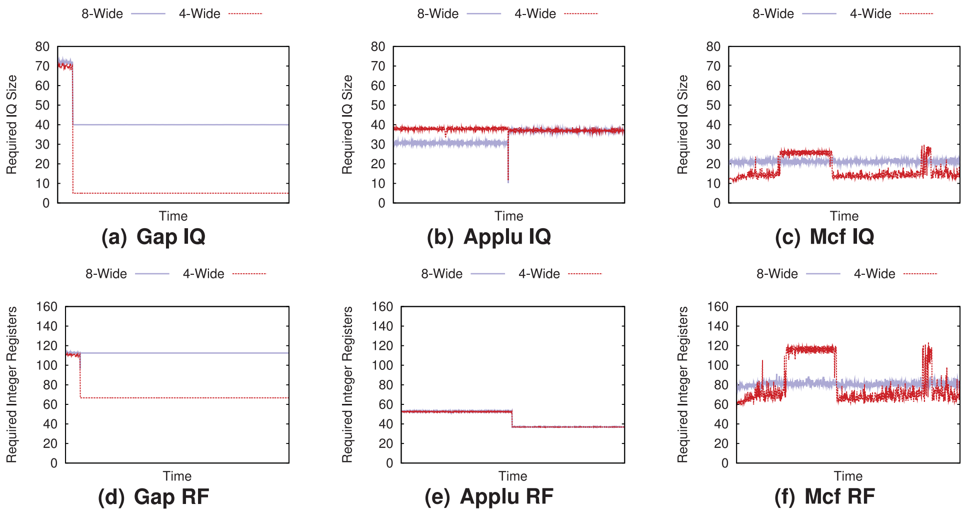


Fig. 1. How the optimal size of two processor structures varies with time for pipeline widths 8 and 4 for three applications.

interactions that exist between hardware structures. Furthermore, runtime searching is undesirable because 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 4 gives a formal description of our model.

## 3. 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

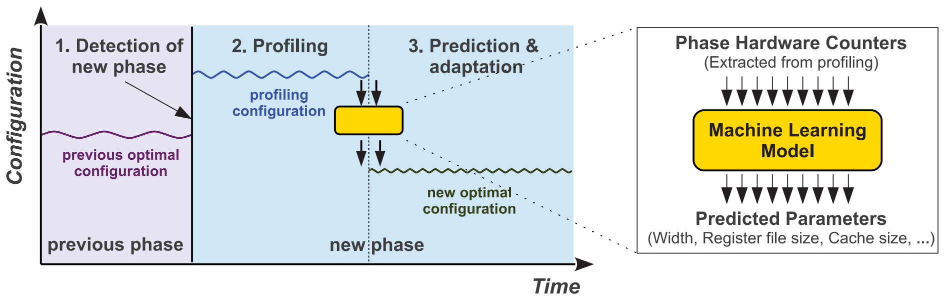

Configuration
1. Detection of new phase
2. Profiling
3. Prediction & adaptation
profiling configuration
previous optimal configuration
new optimal configuration
previous phase
new phase
Time
Phase Hardware Counters
(Extracted from profiling)
Machine Learning Model
Predicted Parameters
(Width, Register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

Fig. 2. Overview of our technique. The hardware detects phase changes, then profiles the application on a predefined 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. Microarchitectural Design Parameters That Were Varied with Their Range, Steps, and the Number of Different Values They Can Take

| Parameter | Value Range | Number | Prior Analysis |
|---|---|---|---|
| Width | 2, 4, 6, 8 | 4 | [Hughes et al. 2001; Ipek et al. 2007; Tarjan et al. 2008] |
| ROB size | $32 \rightarrow 160 : 8+$ | 17 | [Abella and González 2003; Dropsho et al. 2002] |
| IQ size | $8 \rightarrow 80 : 8+$ | 10 | [Abella and González 2003; Dropsho et al. 2002; Buyuktosunoglu et al. 2001] |
| LSQ size | $8 \rightarrow 80 : 8+$ | 10 | [Ponomarev et al. 2001; Dropsho et al. 2002] |
| RF sizes | $40 \rightarrow 160 : 8+$ | 16 | [Abella and González 2003; Dropsho et al. 2002] |
| 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 | [Balasubramonian et al. 2000; Mai et al. 2000] |
| L1 Dcache size | $8K \rightarrow 128K : 2*$ | 5 | [Balasubramonian et al. 2000; Mai et al. 2000] |
| L2 Ucache size | $256K \rightarrow 4M : 2*$ | 5 | [Balasubramonian et al. 2000; Mai et al. 2000] |
| Depth (FO4 delay) | $9 \rightarrow 36 : 3+$ | 10 | [Efthymiou and Garside 2003; Liang et al. 2008; Tiwari et al. 2007] |
| Total | | 627bn | |

first give an overview of how our scheme works, then describe the counters that we gather through dynamic profiling of each program phase.

## 3.1. 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 predefined *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.

Table I shows the configurable microarchitectural parameters that we have considered. It represents the design space of a high-performance out-of-order superscalar

Table II. Hardware Counters Used as an Input to our Machine Learning Model

| Width | Caches |
|---|---|
| ALU usage (histogram) | Stack distance (histogram) |
| Memory port usage (histogram) | Block reuse distance (histogram) |
| Queues | Set reuse distance (histogram) |
| Queue usage (histogram) | Reduced set reuse distance (histogram) |
| Speculative instructions (%) | Branch Predictor |
| Misspeculated instructions (%) | BTB reuse distance (histogram) |
| Register File | Branch misprediction rate (%) |
| Register usage (histogram) | Pipeline Depth |
| Read port usage (histogram) | Cycles per instruction |
| Write port usage (histogram) | |

processor and is similar to spaces that other researchers have considered [Lee and Brooks 2008]. We vary 14 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 9.

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 5 describes the experimental methodology and execution environment in more detail.

## 3.2. 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.

*3.2.1. 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).

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 9 demonstrates that the cost of gathering these counters is negligible. The next section describes the counters gathered during this profiling phase.

*3.2.2. 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 9, 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 for 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 misspeculated. 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.

*Caches.* We use temporal histograms representing stack distance [Beyls and D'Hollander 2001; Ding and Zhong 2003] and reuse distance. Each bin corresponds to a specific distance. Intuitively, the stack distance is important because 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 misprediction 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.

## 3.3. 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 5.3), 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 misspeculated.

For *mgrid* and *swim*, we see that the best queue size directly corresponds to the observed usage during the profiling phase. For these applications, there are few misspeculated instructions (misspec) present in the queue during the phase.

Now consider *parser* and *vortex*, which both have a significant number of misspeculated 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. One reason for this is that a larger load/store queue allows more in-flight loads and, although a significant number are squashed, their speculative execution has a prefetching effect, bringing data into the L1 in advance of it being required. Since these programs have similar

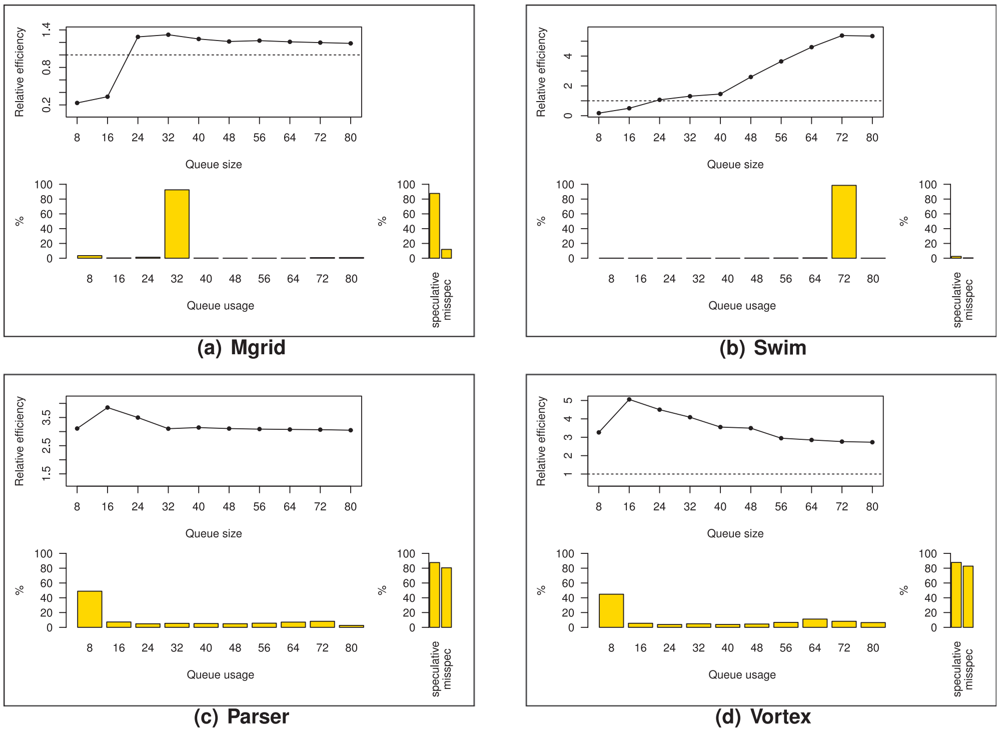


Fig. 3. Load/store queue counters for four phases from different programs. We also show the relative efficiency achieved when varying the load/store queue parameters on the best configuration found (higher is better).

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.

## 4. 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 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 $a = 1, \ldots, A$, 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.

### 4.1. The Model

Our goal is to build a model that correctly captures the relationship between program phases' hardware counters and good microarchitectural configurations (i.e., those with good energy efficiency). In other words, we aim to learn a mapping $\mathcal{X} \to \widetilde{Y}$ 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 $\tilde{\mathbf{y}}$ 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. While the model makes separate predictions for each parameter, the phase's features do contain information about the other parameters. Therefore, the model is able to learn even in the presence of dependencies.

### 4.2. 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 $P(\tilde{y}_a|\mathbf{x})$.

### 4.3. 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:

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

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, 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 that one for each value $\tilde{y}$ can take on.[1]

### 4.4. Model Learning

The task of training the model consists of finding the matrix of weights $\mathbf{W}$, since these are the only parameters in our model. Readers not interested in the mathematical details are invited to jump to the next section, which discusses how the model is used to make predictions.

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

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

can take one out of $K$ possible values as we can learn the model parameters for each architectural parameter independently. The data likelihood is given by:

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

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:

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

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

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

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

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

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmax}}(\mathcal{L}^{\text{POST}}). \tag{7}$$

*Learning* $\mathbf{W}$ *via Gradient-Based Optimisation.* Training our model means finding the solution for $\widehat{\mathbf{W}}$. To this end, we carry out a gradient-based optimisation using the following gradient information:

$$\nabla_{\mathbf{w}_k} \mathcal{L}^{\text{POST}} = \sum_{n=1}^{\tilde{N}} \Big(\delta\big(\tilde{y}^{(n)} = s_k\big) - \sigma_k\big(\mathbf{x}^{(n)}, \mathbf{W}\big)\Big)\, \mathbf{x}^{(n)} + 2\lambda \mathbf{w}_k, \tag{8}$$

where $\nabla_{\mathbf{w}_k} \mathcal{L}^{\text{POST}}$ is the gradient of the penalised data log likelihood (Equation (6)) with respect to the parameter vector corresponding to the $k$th state of each microarchitetural parameter. In our experiments, we use conjugate gradient optimisation with a deterministic initialisation of all the weights to 1 and with $\lambda = 0.5$. See, for example, Bishop [2006] for details of other approaches to parameter learning in these types of models.

### 4.5. Using the Model

To make predictions, only Equations (2) and (3) need to be considered because the training is performed offline. 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. For instance, in the case of the pipeline width we have $K = 4$ possible values: $\{2, 4, 6, 8\}$. Let's say that the corresponding model parameters (learnt during training) are denoted by the $D \times K$ matrix $\mathbf{W}$ ($D$ corresponds to the number of features). Hence, the computations involved for a new program phase characterised by the $D \times 1$ vector of counters $\mathbf{x}^*$ collected during the profiling phase are:

$$\mathbf{b} = \mathbf{W}^T \mathbf{x}^* \tag{9}$$

$$y^* = \underset{k}{\operatorname{argmax}}(b_1, \ldots, b_K), \tag{10}$$

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. Thus, using the model to make a prediction simply consists of multiplying the vector of features with the matrix of weights learnt during training and choosing the parameter value corresponding to the largest element in the resulting vector. So, if the third element happens to be the maximum, we select the thirds parameter value as the prediction. In our example, this would corresponds to a pipeline width of 6.

## 5. 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.

### 5.1. Simulator and Benchmarks

Our cycle-accurate simulator is based on Wattch [Brooks et al. 2000], an extension to SimpleScalar [Burger and Austin 1997]. We altered Wattch's underlying Cacti [Tarjan et al. 2006] 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 [Henning 2000] 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.

### 5.2. 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 [Hartstein and Puzak 2003] to indicate how efficient a configuration is at converting energy into processing speed.

### 5.3. Gathering the Training Data

As seen in Section 4, 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 1,000 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.

Table III. The Configuration of Our Baseline Architecture

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| Issue width | 4 | ROB | 144 |
| IQ, LSQ | 48, 32 | RF, read ports, write ports | 160, 4, 1 |
| Gshare, BTB | 16K, 1K | Branches allowed | 24 |
| Icache, Dcache, Ucache | 64K, 32K, 1M | Depth (FO4 delay) | 12 |

### 5.4. Evaluation Methodology

With this data, we built our model and evaluated it using leave-one-out cross-validation. This is standard machine learning methodology, which 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 of it on the profiling configuration in order to gather the hardware counters. We then use our model to make a prediction and continue execution of the current phase with the configuration supplied by our model. The phases for each application are then combined using the weights provided by SimPoint to get overall results for each program.

We have evaluated our model offline because this article focuses on building a predictive model for microarchitectural adaptation. We have therefore made several assumptions about the environment in which the model works, which are detailed next.

*Phase Detection.* We have assumed that our processor contains the ability to dynamically predict phases. SimPoint is an offline phase detection algorithm, so it could not be used in practise. However, there are a number of examples of online phase detection techniques in the literature that rely on basic block vectors [Sherwood et al. 2003], instruction working sets [Dhodapkar and Smith 2002], or conditional branch counts [Balasubramonian et al. 2000]. All of these could work at runtime. Our work is orthogonal to the choice of phase detection algorithm and is not tied to a particular implementation.

*Dynamic Reconfiguration.* The ability to reconfigure a processor on-the-fly requires logic and circuits to enable and disable parts of each structure on demand. We have assumed that our processor follows the resizing principles from Buyuktosunoglu et al. [2001] and Dropsho et al. [2002].

*Model Implementation and Overheads.* We have assumed that our model can be implemented efficiently in hardware and that in this environment there are no overheads in terms of performance or energy to detect, profile, and adapt to new phases. Whilst these are unrealistic assumptions for a practical implementation of our approach, they allow us to focus solely on the evaluation of our model without the additional complexity inherent in a full design. Nevertheless, we have conducted an initial evaluation of the implementation costs and overheads in Section 9.

### 5.5. Baseline Configuration

In order to determine a suitable baseline, we examined all of 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.

## 6. RESULTS

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

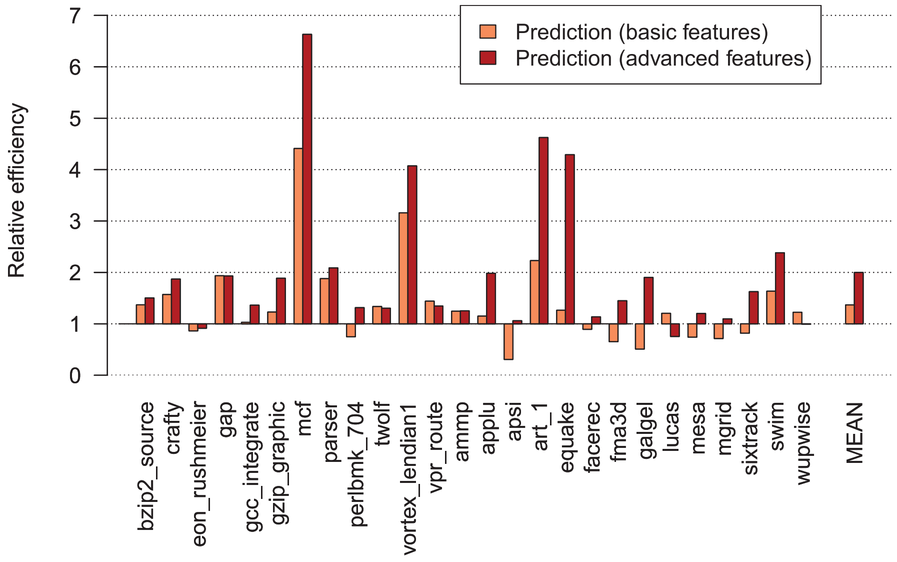


Fig. 4. Energy efficiency [$ips^3/Watt$] achieved by our model compared to the best overall static configuration for SPEC CPU 2000 (higher is better). Two different sets of hardware counters were used with our model: the basic counters are made of the standard performance counters available on current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

### 6.1. 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 3.2.2 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*, and *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.

### 6.2. 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 corresponding size to 256K and 64, respectively. In other cases, such as *art*, the model

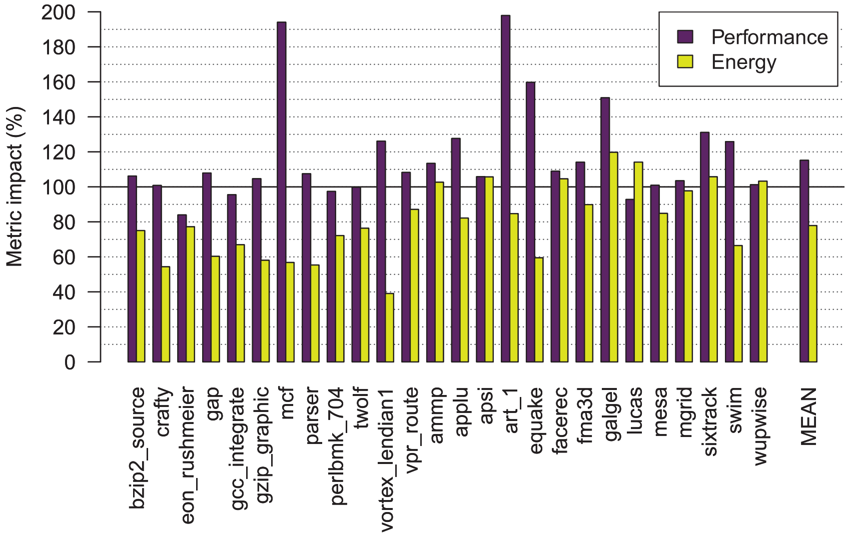


Fig. 5. Performance and energy breakdown for our model when using the advanced features compared to the best overall static configuration. On average, performance is improved by 15% and energy reduced by 21%.

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 instruction 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 used a fixed static configuration tuned for the average case.

## 7. 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.

### 7.1. 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 it does not suffer performance loss in *lucas* and *eon*. Conversely, it is unable to exploit those cases where there is significant improvement available, for example, *mcf* and *equake*, due to the large intraprogram dynamic phase variation.

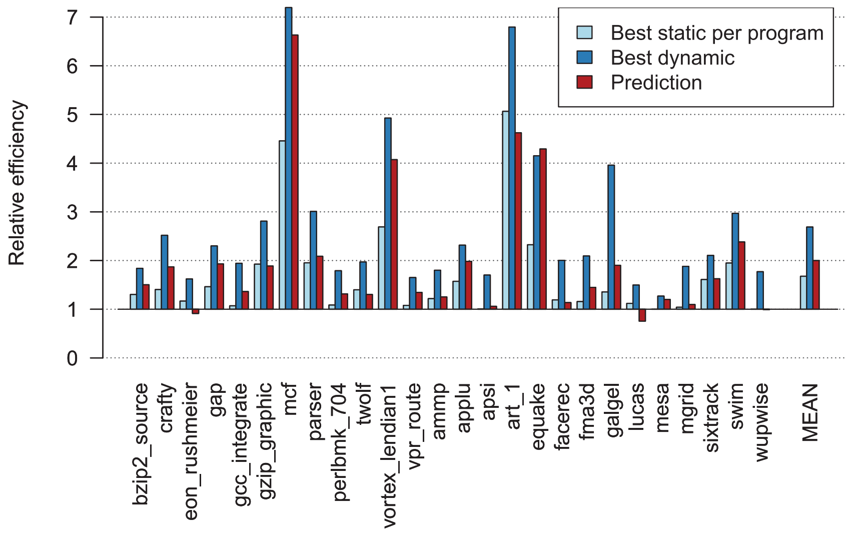


Fig. 6. Energy efficiency achieved by our model for all of SPEC CPU 2000 compared to the best static configuration tailored for each program and compared with the best dynamic configuration tailored for each program's phase. All of the values are normalised by the best overall static configuration (higher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

### 7.2. 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 that there is still room for improvement.

### 7.3. 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 Estimated Cumulative Distribution Function (ECDF). It shows how often our approach achieves at least a certain efficiency improvement. For example, we see that our model predicts a configuration

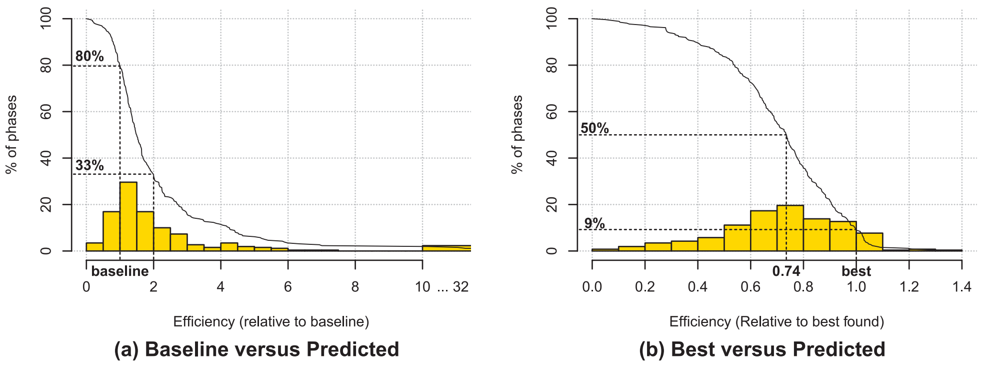


Fig. 7. Histograms showing the distribution of energy efficiency values for the 260 different phases extracted from SPEC 2000 when compared to the baseline (a) and the best (b). In addition the ECDF (estimated cumulative distribution function) is represented by the solid line. The values are accumulated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

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 2x that of the baseline. There are even a small number of phases that achieve improvement of 32x the baseline.

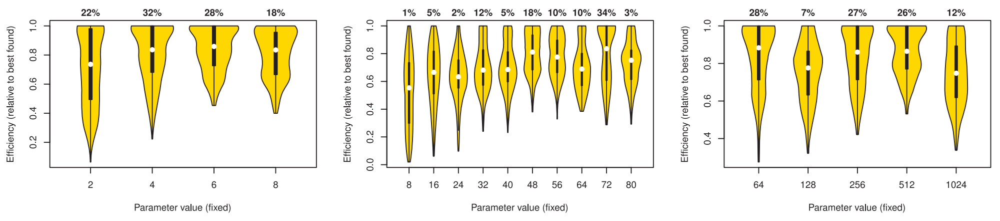
Although it is important to evaluate our approach relative to the best static configuration, it is equally important to compare its accuracy against the best dynamic configurations found in the sample space for each phase, as shown in Figure 7(b). The best configuration has a value of 1. If the performance 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.

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 1,000 samples. This provides evidence that our model can actually predict very efficient parameters.

## 7.4. 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 3 of the 14 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, whereas a width of 4 is best in 32% of cases.



**(a) Width** **(b) Issue Queue** **(c) Instruction Cache**

Fig. 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 represents 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.

By observing these graphs, it is clear that there is no single parameter value 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. Therefore, 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 is not a one-size-fits-all approach, which shows the challenges in building predictors for microarchitectural adaptivity.

## 8. ANALYSIS OF THE BEST CONFIGURATIONS

### 8.1. Best Parameter Values

Having seen how our model can accurately predict energy-efficient configurations for each phase, we now turn our attention to the characteristics of these good configurations. Figure 9 shows a segment diagram of the best parameters found for each program phase. Each parameter is represented by a segment. The bigger the segment, the larger the value that the corresponding parameter takes. For instance, in the case of the first phase of *bzip2_source*, the best configuration found has a small width, a large ROB, a small LSQ and IQ, a medium-sized register file with a medium number of read/write ports, a large gshare cache, a tiny BTB, a high number of speculative branches, tiny data and instruction caches, and a large unified cache. These diagrams show two things. First that the best parameter values vary across programs and second that each program phase requires a different set of parameter values in order to achieve good energy efficiency.

### 8.2. Distributions

In order to better analyse this data, Figure 10 offers a summarised view by showing the key statistical characteristics of the distribution for each parameter, such as the quantile, median, and maximum and minimum values. As can be seen, most parameter values span the whole design space with the notable exception of the fo4 delay (the frequency), which predominantly has the value 9 (the highest frequency). The main

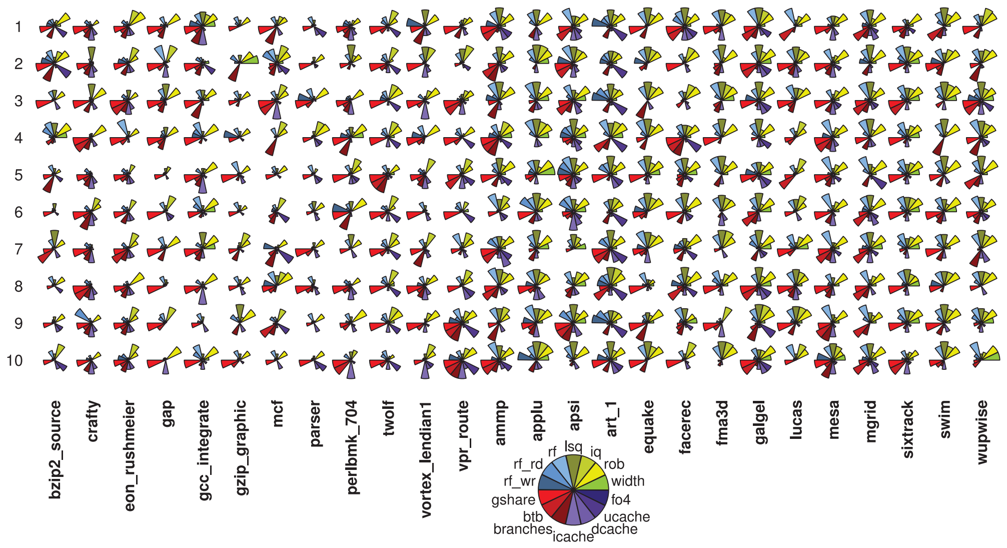


Fig. 9. Segment diagrams representing the best configuration parameters found for each phase. The size of the segment determines the parameter's value, varying between the minimum and maximum values considered for each parameter as shown earlier in Table I.

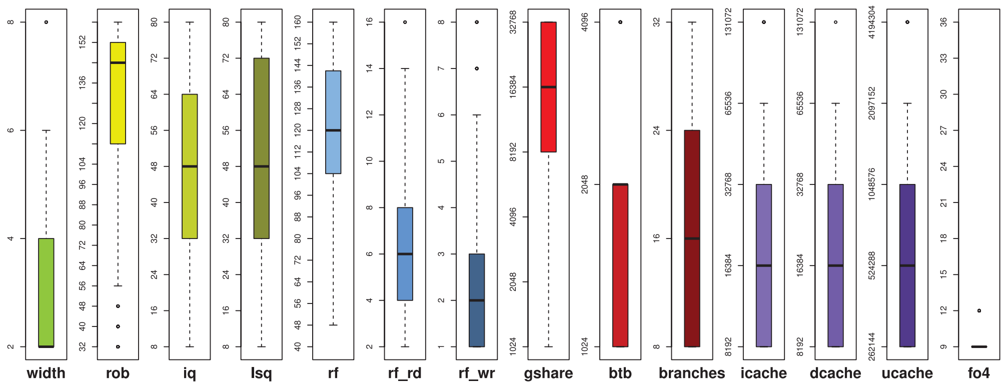


Fig. 10. Main statistical characteristics of the distribution over all program phases of each parameter. The lower and upper limit of each rectangle represent the 25% and 75% quantile, respectively. This means that at least 50% of the parameter values are contained in the rectangle. The median is represented by a thick horizontal bar within each rectangle. The maximum and minimum values are represented by the horizontal lines outside each rectangle, whereas the circles represent the outliers.

explanation for this is that we have used [$ips^3/Watt$] for efficiency, which favours performance over energy. As a result, slowing the core frequency would cause more of a performance penalty than can be gained back through energy savings.

Another interesting result is the fact that the optimal pipeline width is rarely over 6, with values typically between 2 and 4 for 75% of the phases. This is because most applications exhibit a low IPC ratio, meaning too much energy would be wasted with a wider pipeline. However, given our dynamic approach, the pipeline width is still sometimes set to 6 or even 8 when the phase has a potential for high IPC. Following this

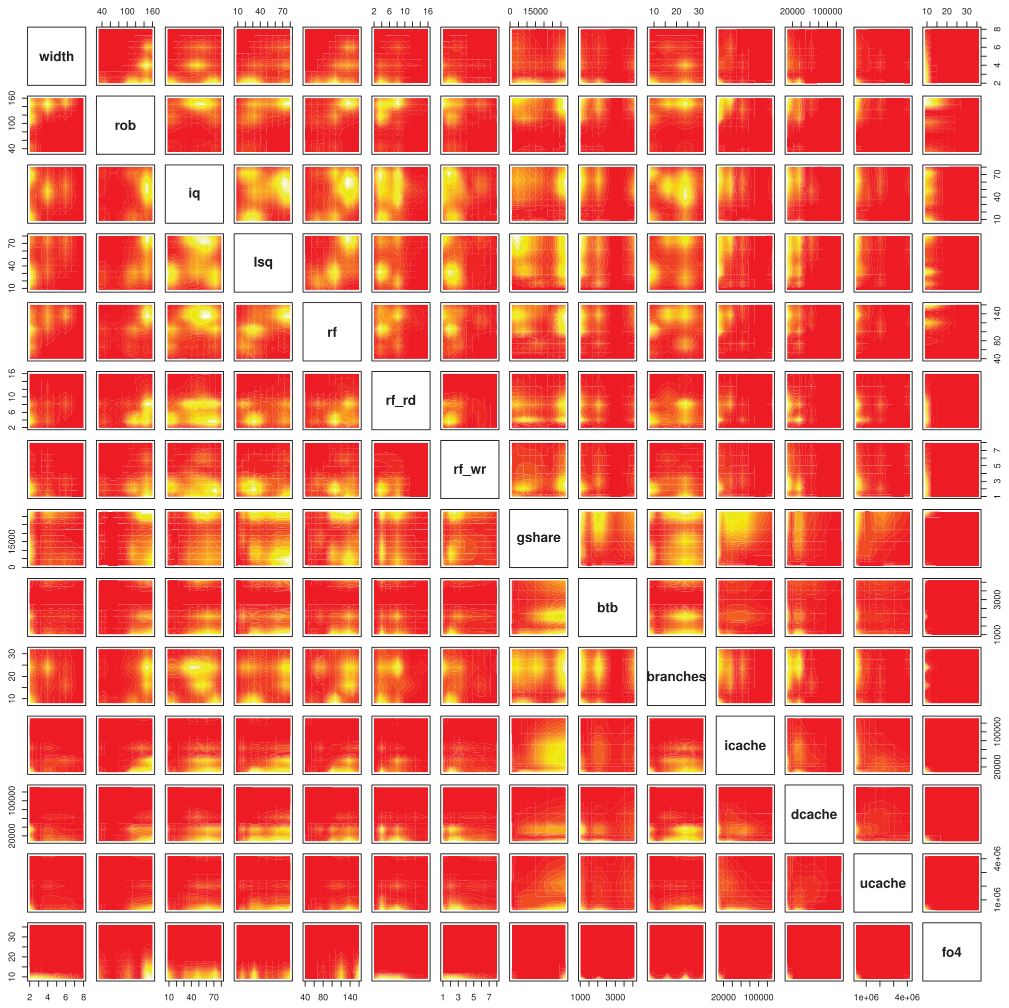


Fig. 11. Scatter plot matrix of the best parameter values per phase. We have used a kernel density estimation method for visualisation purposes. Light areas represents dense regions, whereas dark (red) areas mean that the corresponding pair of parameter values does not appear in the best configurations.

reasoning, it is also natural to observe that for most phases, the number of read/write ports to the register file is small given the small pipeline width. Finally, we observe that the cache size is almost never set to its maximal values but remains relatively small. This, again, makes perfect sense because we want to tune the microarchitecture for energy efficiency, and big caches are known to be very power hungry. We conclude that for most phases, the sacrifice in performance due to the small cache is worth the energy savings when it comes to dynamic adaptation.

### 8.3. Parameter Interactions

Having analysed the parameters in isolation, we now want to study the possible interactions between them. Figure 11 shows a scatter plot matrix for each pair of parameters, corresponding to the best configurations per phase (Figure 9). In this figure, we have used a kernel density estimation technique for visualisation purposes. The light yellow areas correspond to dense regions, that is, those where the pair of parameter's values occurs most frequently among the best configuration for each phase.

The first thing we notice is that there is not a single pair of parameters that correlates well with each other. This either means that most of the parameters are independent from one another or—as is more likely to be the case—the interactions between the parameters are too complex (i.e., $n$-dimensional interactions where $n > 2$) to be visualised or quantified easily.

Nonetheless, there are some observations that can be made from Figure 11. As the *width* of the pipeline gets bigger, so does the reorder buffer (*rob*) and the register file (*rf*). This is expected, since a wider pipeline means more in-flight instructions, which puts more pressure on the reorder buffer and register file. Another interesting observation is that the configurations exhibiting a large load-store queue (*lsq*) tend to have a large register file (*rf*). This is most likely attributed to the fact that a large load-store queue can prevent pipeline stalls, which in turn increases the number of in-flight instructions and puts pressure on the register file. When it comes to the branch predictor, we observe that configurations that allow a large number of speculative *branches* tend to have a larger reorder buffer (*rob*). Finally, when we consider the caches, we observe that there are no configurations exhibiting a large instruction cache (*icache*) and a large data cache (*dcache*) at the same time. This is probably because such configurations would be too power hungry and so are not selected due to our optimisation for power efficiency.

### 8.4. Phase Clustering

We now want to study the similarity of program phases. Instead of using features extracted from the performance counters, we decided to directly assess phase similarities using the variation in the energy efficiency value when running the same program phase on different configurations. This offers the best way of characterising phases, since we are ultimately interested in the efficiency of each configuration rather the value of the performance counters. Following this approach, two phases are similar if they respond in the same way when running them on different configurations—that is, configurations that are good for one phase are good for the other one, and conversely, configurations that are bad for one phase are bad for the other one.

In order to quantify the distance between phases, we normalise the $[ips^3/Watt]$ values and then use the Euclidean distance. The normalisation process consists of scaling the efficiency values from the 1,000 sample configurations so that the mean is 0 and the variance 1. We then applied the K-means clustering algorithm to find clusters of similar configurations. Once the clusters have been identified, we can assign the same best configuration to all phases in the cluster.

Figure 12 shows the relative efficiency when using the K-means clustering technique with different numbers of clusters. The first data point, which shows the highest efficiency, corresponds to 260 clusters, that is, each phase forms its own cluster, with its own best configuration. This, therefore, corresponds to the maximum efficiency achievable. At the other end, when we have only a single cluster, all of the phases use the same configuration, and this leads to the average best efficiency. In fact, this corresponds to our baseline configuration, which is fixed for all phases.

As can be seen in Figure 12, the average efficiency starts to drop significantly when we have fewer than 40 clusters. This means that, as far as the average efficiency achieved across all phases is concerned, more than 40 different types of phase exist. However, if we look at the worst-case scenario, we see that the efficiency starts dropping very early on, even when using more than 200 clusters. This result shows that program phases are in fact very different from one another and that it is difficult to cluster or classify them into different classes.

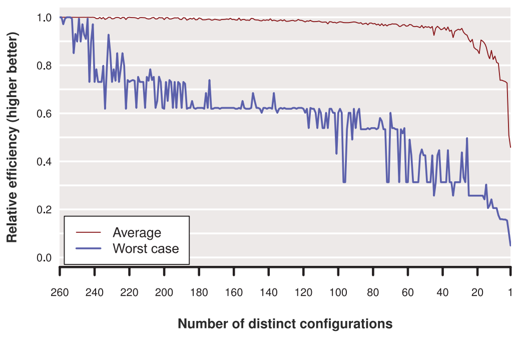


Fig. 12. Clustering program phases using the K-means algorithm on the output metric and its impact on efficiency. On the left, we see the best possible efficiency value when each phase is assigned to its own unique cluster. On the far right, we see the effect of assigning all phases into the same, single cluster (same best configuration for all phases). We show the average efficiency across all phases and the worst degradation on a per-phase basis.

## 9. 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.

### 9.1. Gathering Hardware Counters

The construction of our temporal histograms is the main overhead when gathering our counters. However, an efficient implementation is feasible and is detailed for each type of hardware structure as follows.

*Caches*. 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 now explore how dynamic set sampling [Qureshi et al. 2006] can 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 with differing numbers of sampled sets. Figure 13 shows the results obtained for the three caches and the two metrics we collected: the set and block reuse distances. The dashed and dotted lines show the overheads of collecting the features in terms of dynamic and static energy, respectively. As can be seen, gathering the block reuse distances incurs a higher overhead; however, it drops significantly as we sample fewer sets. The correlation coefficient of the features extracted, with respect to the baseline that uses the maximum number of sets, is also shown.[2] The block reuse distance features are the most affected by the sampling. We can still achieve high accuracy by sampling 128 sets for both the instruction cache and data cache, for example, and reduce the overheads to less than 2% in both cases.

However, these overheads are only required when running the profiling configuration. Through analysis of the SimPoint classification of each application interval, we have observed that reconfiguration occurs only once every 10 intervals of 10 million

[2]This is the lowest value achieved for 95% of the phases (i.e, 95% quantile).

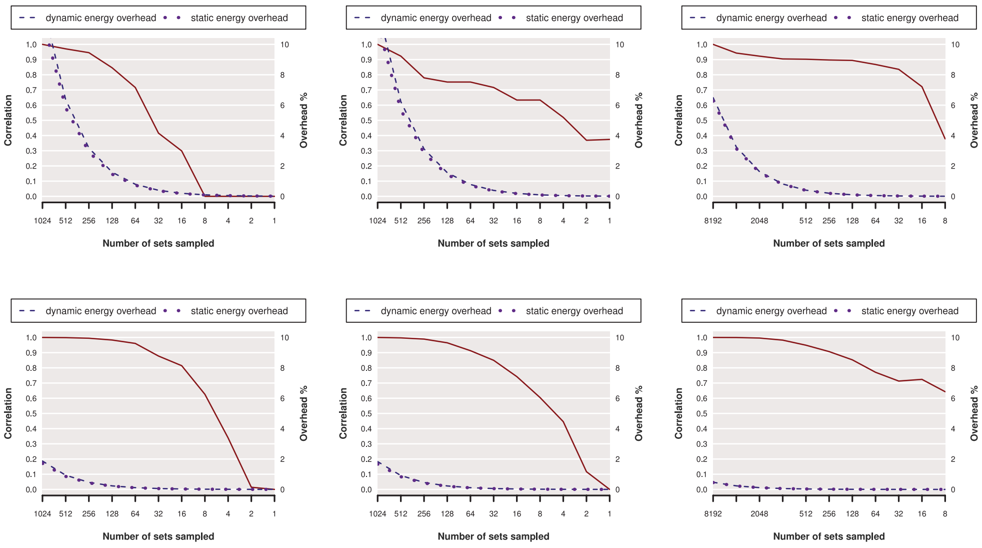


Fig. 13. Set sampling and its effect on overheads and feature accuracy measured with the minimum coefficient of correlation (for the 95% quantile).

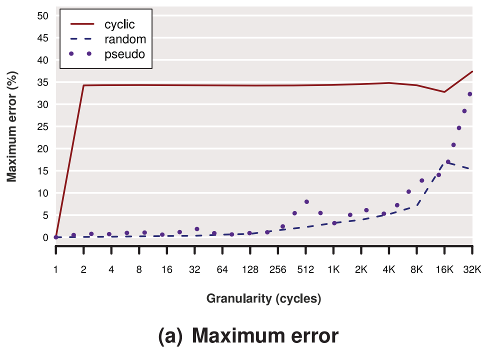
instructions, on average. Therefore, the overall overheads of gathering these counters during the profiling phase become almost insignificant. These results show that gathering our hardware counters is cost-effective considering the efficiency savings that our model achieves.

*Other Structures.* To characterise the pipeline width, we require temporal histograms for the ALU type and memory port usage. This totals 5 histograms with a maximum number of 9 counters in each. The issue queue and reorder buffer each require a single temporal histogram with 7 counters for the former and 17 for the latter, since this can be reconfigured to a larger number of sizes. For the register files, we require 6 temporal histograms (3 for the integer register file, 3 for the floating point counterpart), totalling 34 counters per register file. Finally, the branch predictor can be accurately characterised using 1 temporal histogram containing 8 counters.

*Summary.* In total, we require 48 temporal histograms to characterise each application phase and the optimum size of each processor structure. Gathering them results in an increase of 0.17% in overall processor dynamic energy and 0.28% in static energy.

### 9.2. Temporal Feature Sampling

Should the overheads of gathering features be too high, it is possible to reduce the frequency of gathering. Figure 14 shows the effects of temporal sampling on accuracy when building our histograms. Figure 14(a) shows the maximum error (lower is better), and Figure 14(b) shows the minimum correlation (higher is better) ever achieved across all phases of all programs and parameters. The $x$-axis shows the granularity of sampling in terms of cycles. The baseline case shows an error of 0% or a correlation of 1, corresponding to statistics collection every cycle. The three lines labelled *cyclic*, *random*, and *pseudo* correspond to different sampling strategies: cyclic simply registers the statistics every $x$ cycles, where $x$ is the granularity; random collects statistics using a random number generator, with an average period of $x$; and pseudo uses a simple pseudo-random strategy that splits execution into intervals of $x$ cycles, collecting



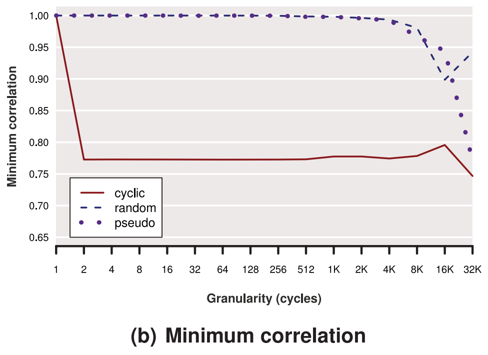


Fig. 14. Effects of temporal sampling on accuracy when gathering the temporal histograms for three different sampling strategies. The $x$-axis shows the sampling granularity or period. The cyclic approach collects statistics at regular interval $x$, random collects statistics at random interval with an average period of $x$, and pseudo collects statistics using a simple pseudo-randomly strategy with a period of $x$.

Table IV. Overheads of Reconfiguring Each Structure in Cycles

| Processor Structure | Cycle Overhead |
|---|---|
| Width | 443 |
| RF | 487 |
| Bpred | 154 |
| ROB / IQ / LSQ | 255 / 234 / 275 |
| I/D/U Cache | 478 / 620 / 18322 |

statistics one cycle later in each interval and in two consecutive cycles where the collection cycle is the last in the interval. Each point on the lines at a particular granularity corresponds to the same number of samples over the whole execution.

As can be seen, the cyclic strategy performs the worst. For some program phases, there exists a cyclic behaviour within the features; thus, when we collect the statistics in a cyclic manner, we risk missing important events. While the random strategy exhibits the best performance, pseudo is very close and can be used as a good substitute because it does not rely on a random number generator. The pseudo strategy with a granularity of 256 is a good compromise between accuracy and overhead reduction, with a maximum error of 2% and a correlation of nearly 1. This reduces the dynamic energy overheads of gathering the temporal histograms to 0.03% (from 0.17%, see Section 9.1). Static energy overheads remain the same because we cannot turn the temporal histograms off during temporal sampling.

## 9.3. Resource Reconfiguration

Adaptation can be achieved through the use of simple bitline segmentation of processor structures [Dropsho et al. 2002; Buyuktosunoglu et al. 2001]. 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 [Royannez et al. 2005]. In addition, we have accurately modelled the delays required to flush caches and stall the pipeline when resources need reconfiguration (see Table IV).

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 performance penalty when reconfiguration occurs is just 3% for one interval and that the energy overheads are also 3%. However,

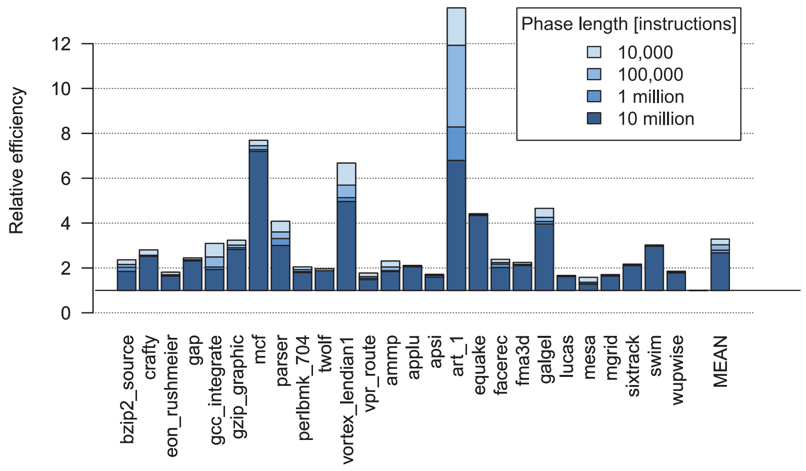


Fig. 15. Energy efficiency achieved when varying the phase length from 10 million instructions down to 10,000 instructions.

since reconfiguration only occurs once every 10 intervals, the overheads for the whole phase are significantly reduced. Therefore, reconfiguring processor resources can be achieved with very few overheads that are amortized over the execution of the whole phase.

### 9.4. Model

Work by Jiménez and Lin [2002] has shown how to build a perceptron-based neural branch predictor. At prediction time, our technique can be seen as a multiclass 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 (**W**). Since we have approximately 2,000 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.

## 10. PHASE GRANULARITY STUDY

Throughout this article, we have assumed a phase length of 10 million instructions. This section analyses how varying the phase granularity affects energy efficiency. We have varied the phase length from the original 10 million instructions to 10 thousand instructions, as shown in Figure 15. As expected under our ideal setup, the energy efficiency increases as the phases get shorter. However, the added benefit of having smaller phases is negligible compared to the original 10 million phase length. On average, energy efficiency increases from 2.7x to 3.2x, a mere 15% improvement.

With the exception of *art_1*, most benchmarks exhibit very small improvements when the phase length is reduced. In the case of *art_1*, the energy efficiency is doubled when using a phase length of 10,000 instructions, indicating that the program phases tend to be much smaller for this application. However, considering that reconfiguring the L2 cache costs around 20,000 cycles (see Section 9.3), it seems that such a phase length would result in excessive overheads, bringing the energy efficiency down. As such, we conclude that having coarser phases is best for energy efficiency and that the choice of 10 million instructions phase used in this work is justified.

## 11. PRIOR WORK ON MICROARCHITECTURAL ADAPTIVITY

This article is an extension of our previously published conference paper [Dubach et al. 2010]. Compared to the original version, we have extended the analysis section by discussing the characteristics of the best configurations found for each individual phase. In addition, we have proposed new schemes to further reduce the overhead of the feature collection, such as temporal sampling. Furthermore, we have studied the impact of varying the phase granularity and demonstrated that program phases of 10 million cycles offer a good trade-off for microarchitectural adaptation.

*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 [Folegnani and Gonzalez 2001; Ponomarev et al. 2001; Abella and González 2003; Dropsho et al. 2002; Buyuktosunoglu et al. 2001; Albonesi 1998], reorder buffer [Abella and González 2003; Dropsho et al. 2002], register files [Abella and González 2003; Dropsho et al. 2002], pipeline [Efthymiou and Garside 2003; Hughes et al. 2001] and caches [Balasubramonian et al. 2000; Albonesi 1998] have been studied.

Dhodapkar and Smith [2002] focused on control mechanisms by assessing the use of working set signatures to detect changes in behaviour of the program. Liang et al. [2008] and Tiwari et al. [2007] separately proposed variable latency architectures where additional stages can be added to the pipeline to combat process variations.

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 [Kontorinis et al. 2009] was proposed to reduce peak power in an adaptive processor. In comparison, our work considers varying all of these parameters together and uses a machine learning model to control the adaptation process.

*Multicore Adaptivity.* For multicore processors, Mai et al. [2000] illustrated an adaptive memory substrate and its flexibility when implementing very different architectures named “Smart Memories.” Mai et al. [2000]. Later, Sankaralingam et al. [2003] proposed the TRIPS architecture Sankaralingam et al. [2003], Ipek et al. [2007] “Core Fusion” Ipek et al. [2007], and Tarjan et al. [2008] “Core Federation” Tarjan et al. [2008]. 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. [2001] looked at multimedia applications characterised by repeated frame processing. Hsu and Kremer [2003] implemented a compiler algorithm that adapts the voltage and frequency based on the characteristics of the code. Later, Wu et al. [2005] looked at adapting the voltage within the context of a dynamic compilation framework that can monitor and transform the program as it is running. Huang et al. [2003] proposed using subroutines as a natural way to decide when to reconfigure the processor. Finally, Isci et al. [2006] developed a real system framework that predicts program phases on the fly to guide dynamic voltage and frequency scaling.

*Runtime Exploration.* Other researchers looked at learning or searching the space at runtime [Bitirgen et al. 2008; Choi and Yeung 2006; Ponomarev et al. 2001]. In our context, it is undesirable to perform any sort of runtime exploration, because this would inevitably result in visiting poorly performing configurations and reducing the overall efficiency.

There are three reasons why this type of approach is not suitable for our design space. Firstly, in order to explore the space at runtime, we would have to evaluate many different configurations during the same phase. As we have seen, a change of configuration means caches and other structures may need to be flushed. These small overheads would accumulate over time during an online search, leading to nonnegligible overall overheads.

Secondly, there are challenges in measuring performance and power. In a runtime learning scheme, after each reconfiguration, we would have to run for a certain number of cycles before being able to assess the performance of the new configuration and alleviate problems such as cold caches, adding to the total overheads. Further, with our approach, we do not directly measure energy efficiency, but only gather performance counters. However, with an online learning approach, we would have to measure energy consumption at a very fine granularity, which would be more challenging than assessing the performance of each configuration.

Finally, the number of parameters in the space means that hill-climbing approaches, such as Choi and Yeung [2006], become extremely complex, since this increases exponentially with the dimensions of the space (when interactions are taken into account). In our case, we have 14 parameters, which means that finding the gradient would require a large number of samples.

In Bitirgen et al. [2008], the authors tried to alleviate some of these issues by training application-specific models to estimate performance. However, these models, based on Artificial Neural Networks, need to be initially trained and adapted over time, which again brings us back to the same three main issues discussed previously.

*Predictive Models.* Recently, Ipek et al. [2006], Lee and Brooks [2006], and Joseph et al. [2006] 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, because they would require a search of the design space at runtime.

Lee and Brooks [2008] 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.

*Phase Detection.* Phase detection techniques are at the core of any dynamic adaptive system and have been extensively studied. The work from Dhodapkar and Smith [2003] 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 [Sherwood et al. 2003], instruction working sets [Dhodapkar and Smith 2002], or conditional branch counts [Balasubramonian et al. 2000]. Wavelet analysis has also gained some attention [Cho et al. 2007; Shen et al. 2004].

## 12. CONCLUSION AND FUTURE DIRECTIONS

This article 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 rather 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 were 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 structure.

Our processor is only capable of running a single application at a time. We believe that there would only need to be minor changes made to support a core with SMT capabilities. In this scenario, the features sampled would correspond to multiple threads running concurrently. However, we do not expect that this would change the overall structure of the model or the predictions that it would make, because it is independent of the workload running on the core. We would simply need to train it using examples of threads that could be scheduled together.

A second scenario that presents additional opportunities for energy saving would come within a CMP design. Here, we may be able to reduce the cost of reconfiguring by swapping two threads instead of reconfiguring two cores. To achieve this, we envision an additional model that monitors thread features across all (or a subset) or cores and makes decisions about thread swapping, whereas each core still has the local model presented in this work to reconfigure for the thread that is going to be run on the corresponding core.

Finally, this article 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 dynamically adapt to their own workloads, this would lead to true heterogeneity, which is 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.

## ACKNOWLEDGMENTS

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[3]). The ECDF is partially supported by the eDIKT[4] initiative. 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

Abella, J. and González, A. 2003. On reducing register pressure and energy in multiple-banked register files. In *Proceedings of the 21st International Conference on Computer Design (ICCD'03)*. 14.

Albonesi, D. 1998. Dynamic IPC/clock rate optimization. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. 282–292.

Balasubramonian, R., Albonesi, D., Buyuktosunoglu, A., and Dwarkadas, S. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'00)*. 245–257.

Beyls, K. and D'Hollander, E. H. 2001. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems.* 617–662

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

Bitirgen, R., Ipek, E., and Martinez, J. F. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture.* 318–329

[3]http://www.ecdf.ed.ac.uk.
[4]http://www.edikt.org.uk.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture.* 83–94

BURGER, D. AND AUSTIN, T. 1997. The simplescalar tool set, version 2.0. Tech. rep. TR-1342. University of Wisconsin.

BUYUKTOSUNOGLU, A., ALBONESI, D., SCHUSTER, S., BROOKS, D., BOSE, P., AND COOK, P. 2001. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes Symposium on VLSI (GLSVLSI'01)*. 73–78.

CHO, C.-B., ZHANG, W., AND LI, T. 2007. Informed microarchitecture design space exploration using workload dynamics. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 274–285.

CHOI, S. AND YEUNG, D. 2006. Learning-based SMT processor resource distribution via hill-climbing. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. 239–251.

DHODAPKAR, A. AND SMITH, J. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 233–244.

DHODAPKAR, A. S. AND SMITH, J. E. 2003. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*. 217.

DING, C. AND ZHONG, Y. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. 245–257.

DROPSHO, S., BUYUKTOSUNOGLU, A., BALASUBRAMONIAN, R., ALBONESI, D. H., DWARKADAS, S., SEMERANO, G., MAGKLIS, G., AND SCOTT, M. L. 2002. Integrating adaptive on-chip storage structures for reduced dynamic power. Tech. rep. University of Rochester.

DUBACH, C., JONES, T., AND O'BOYLE, M. 2007. Microarchitectural design space exploration using an architecture-centric approach. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 262–271.

DUBACH, C., JONES, T. M., BONILLA, E. V., AND O'BOYLE, M. F. P. 2010. A predictive model for dynamic microarchitectural adaptivity control. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 485–496.

EFTHYMIOU, A. AND GARSIDE, J. 2003. Adaptive pipeline structures for speculation control. In *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*. 46–55.

FOLEGNANI, D. AND GONZALEZ, A. 2001. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. 230–239.

HARTSTEIN, A. AND PUZAK, T. R. 2003. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. 117.

HENNING, J. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Comp. 33*, 7, 28–35.

HSU, C.-H. AND KREMER, U. 2003. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*. 38–48.

HUANG, M., RENAU, J., AND TORRELLAS, J. 2003. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. 157–168.

HUGHES, C., SRINIVASAN, J., AND ADVE, S. 2001. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34)*. 250–261.

IPEK, E., MCKEE, S. A., CARUANA, R., DE SUPINSKI, B. R., AND SCHULZ, M.2006. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. 195–206.

IPEK, E., KIRMAN, M., KIRMAN, N., AND MARTINEZ, J. 2007. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. 186–197.

ISCI, C., CONTRERAS, G., AND MARTONOSI, M. 2006. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. 359–370.

JIMÉNEZ, D. A. AND LIN, C. 2002. Neural methods for dynamic branch prediction. *ACM T. Comput. Syst. 20,* 4, 369–397.

JOSEPH, P., VASWANI, K., AND THAZHUTHAVEETIL, M. J. 2006. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 161–170.

KONTORINIS, V., SHAYAN, A., TULLSEN, D. M., AND KUMAR, R. 2009. Reducing peak power with a table-driven adaptive processor core. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 189–200.

LEE, B. AND BROOKS, D. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. 185–194.

LEE, B. AND BROOKS, D. 2008. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. 36–47.

LIANG, X., WEI, G.-Y., AND BROOKS, D. 2008. ReVIVaL: A variation-tolerant architecture using voltage interpolation and variable latency. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. 191–202.

MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W., AND HOROWITZ, M. 2000. Smart Memories: A modular reconfigurable architecture. In *Proceedings of the 27th International Symposium on Computer Architecture*. 161–171.

PONOMAREV, D., KUCUK, G., AND GHOSE, K. 2001. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34)*. 90–101.

QURESHI, M. K., LYNCH, D. N., MUTLU, O., AND PATT, Y. N. 2006. A case for MLP-aware cache replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. 167–178.

ROYANNEZ, P., MAIR, H., DAHAN, F., WAGNER, M., STREETER, M., BOUETEL, L., BLASQUEZ, J., CLASEN, H., SEMINO, G., DONG, J., SCOTT, D., PITTS, B., RAIBAUT, C., AND KO, U. 2005. 90nm low leakage SoC design techniques for wireless applications. In *Proceedings of the IEEE International Solid-State Circuits Conference*. 138–589.

SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S. W., AND MOORE, C. R. 2003. Exploiting ILP, TLP, and DLP with the polymorphous trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. 422–433.

SHEN, X., ZHONG, Y., AND DING, C. 2004. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. 165–176.

SHERWOOD, T., SAIR, S., AND CALDER, B. 2003. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. 336–349.

TARJAN, D., BOYER, M., AND SKADRON, K. 2008. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the 45th ACM/IEEE Design Automation Conference (DAC'08)*. 772–775.

TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. P. 2006. Cacti 4.0. Tech. rep. HPL-2006-86. HP Laboratories, Palo Alto, CA.

TIWARI, A., SARANGI, S., AND TORRELLAS, J. 2007. ReCycle: Pipeline adaptation to tolerate process variation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. 323–334.

WU, Q., MARTONOSI, M., CLARK, D. W., REDDI, V. J., CONNORS, D., WU, Y., LEE, J., AND BROOKS, D. 2005. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. 271–282.

Received February 2013; revised May 2013; accepted July 2013