AMERICAN UNIVERSITY OF BEIRUT

A UNIFIED APPROACH TO MAXIMIZING RUN-TIME PERFORMANCE ON POWER CONSTRAINED ADAPTIVE PROCESSORS USING SUPPORT VECTOR REGRESSION

by

MELISSA STOCKMAN

A dissertation
submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
to the Department of Electrical and Computer Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
January 2014
A UNIFIED APPROACH TO MAXIMIZING RUN-TIME PERFORMANCE ON POWER CONSTRAINED ADAPTIVE PROCESSORS USING SUPPORT VECTOR REGRESSION

by

MELISSA STOCKMAN

Approved by:

Dr. Karim Kabalan, Professor
Department of Electrical and Computer Engineering

Dr. Haitham Akkary, Associate Professor
Department of Electrical and Computer Engineering

Dr. Mariette Awad, Assistant Professor
Department of Electrical and Computer Engineering

Dr. Mazen Saghir, Associate Professor
Texas A&M University at Qatar

Dr. Roman Neruda, Researcher
Institute of Computer Science, Academy of Sciences of the Czech Republic

Date of thesis/dissertation defense: January 3, 2014
I, Melissa Stockman

[ ] authorize the American University of Beirut to supply copies of my thesis/dissertation/project to libraries or individuals upon request.

or

do not authorize the American University of Beirut to supply copies of my thesis/dissertation/project to libraries or individuals for a period of two years starting with the date of the thesis/dissertation/deposit.

[Signature]

Jan 16, 2014

Date
ACKNOWLEDGMENTS

I would first and foremost like to thank my advisors Dr. Haitham Akkary and Dr. Mariette Awad who have helped me through this long journey. Without their guidance, support and encouragement I would not have made it. I would also like to thank my committee chairman Dr. Karim Kabalan for his always helpful and positive attitude. I thank my committee members Dr. Roman Neruda, and Dr. Mazen Saghir for their input and feedback.

Loving thanks to my father, Gerald Stockman, who by his example has taught me all the most important things in life, and to his wife Kathy who explicitly motivated me with an email that mentally saved me at a point in my life where I did not think I would be able to continue.

I thank my kids, Ryan, Karina and Shirine for being the source of my biggest happiness.

And finally I thank my two Philippina friends Juliet Panglinan and Benadette Dela Cruz, who each lived with me over some part of these last years. I truly could not have done it without their kindness and help.
AN ABSTRACT OF THE DISSERTATION OF

Melissa Stockman for Doctor of Philosophy
Major: Electrical and Computer Engineering

Title: A Unified Approach to Maximizing Run-Time Performance on Power Constrained Adaptive Processors using Support Vector Regression

Processor resource needs vary across applications as well as within individual applications. Some programs may be memory intensive; others may have a high amount of branching, while others may need many floating point computations. Maximal configurations waste power when an application’s performance would not be degraded running on a lower power configuration. By adjusting a processor’s configuration during application runtime, it is possible to save power without a performance cost.

But maximizing performance and minimizing power are two conflicting objectives. Knowing what configuration will be the best at a given time for a running program is not easily determined. The majority of previous works have attempted to adjust a small number of microarchitectural parameters at a time. This detracts from the overall performance/power achievements that can be attained by adjusting a large number of parameters together. Additionally, none of these previous works have taken into consideration the fact that processor variability affects the model’s effectiveness. Our framework allows for building separate loadable models for chips with different leakage percentages.

In this dissertation we propose using a machine learning technique to determine maximal performance configurations during program execution. Specifically we use ‘support vector regression’ (SVR) to determine the best configurations for a given power level which maximizes performance in terms of IPC while constraining power to some user defined level. This research aims to reduce power in a global manner by determining maximal performance configurations for more than 17 variable microarchitectural parameters simultaneously in order to benefit from any interaction these units may have on each other. This number of variables makes it impossible to try every combination of all parameters in a trial and error method; therefore a machine learning approach was taken.

We provide a framework for determining unified maximal performance configurations for power constrained adaptive processors at runtime. This framework has the advantage of being adaptable post silicon, compensating for the inevitable
variability in leakage and transistor characteristics. We develop three different models to select from depending on user preference: 1) for highest accuracy which uses SVR technique 2) for least cost prediction in which we modify the SVR technique by reducing the number of support vectors and 3) for least over prediction power in which we modify the SVR technique to asymmetrically predict configurations using an upper bound (AUB-SVR). We empirically demonstrate the advantages of adaptive hardware as compared to DVFS techniques, showing a 17.8% decrease in power and a 17.1% increase in IPC. We also show an average 5.38% increase in IPC when compared to static configurations for 20 different power levels. We determine a reduced set of around 40 maximal performance configurations from the 486 originally presented and we find the smallest set of hardware counters required to give best prediction results.
CONTENTS

ACKNOWLEDGMENTS ........................................................................... v

ABSTRACT .......................................................................................... vi

FIGURES ............................................................................................. xi

TABLES ............................................................................................... xiii

1. INTRODUCTION ................................................................. 1

   1.1 Motivation .............................................................................. 1
   1.2 Thesis Statement ................................................................... 3
   1.3 Contribution ........................................................................... 3
   1.4 Dissertation Organization .................................................... 5

2. BACKGROUND AND RELATED WORK ................................. 6

   2.1 Adaptive Microprocessors .................................................... 6
   2.2 Program Phase Detection ...................................................... 10
   2.3 Our Related Work ................................................................. 12

3. SUPPORT VECTOR REGRESSION ...................................... 14

   3.1 Machine Learning ................................................................. 14
   3.2 Support Vector Regression (SVR) Overview ......................... 15
   3.3 Training and Validation Techniques ...................................... 19
   3.4 Structural Risk Minimization ............................................... 20
   3.5 Asymmetrical and Upper Bound SVR (AUB-SVR) ................. 21
   3.6 AUB-SVR Error Bounds ....................................................... 24
   3.7 AUB-SVR Related Work ....................................................... 25
   3.8 Feature Extraction ............................................................... 26
       3.8.1 Principal Component Analysis ...................................... 27
       3.8.2 Entropy and Information Gain ...................................... 28
       3.8.3 T-Test ........................................................................... 28
   3.9 SVR Example ....................................................................... 29

4. MODEL SETUP AND EVALUATION METHODOLOGY. 32
FIGURES

Figure 1. Typical adaptive parameters in an adaptive processor ...........................................2

Figure 2. Program phase behaviour [105] ................................................................................11

Figure 3. SVR with $\varepsilon$-insensitive tube ............................................................................16

Figure 4. $\varepsilon$-insensitive loss function ..................................................................................17

Figure 5. Structural Risk Minimization [75] ............................................................................20

Figure 6. AUB-SVR with $\varepsilon$-insensitive tube ....................................................................23

Figure 7. AUB-SVR $\varepsilon$-insensitive loss function ..................................................................24

Figure 8. PCA transformation ....................................................................................................27

Figure 9. An output vector for each SimPoint run on each configuration .................................29

Figure 10. Original training dataset is broken up into power levels and individual SVR models are trained ................................................................................................................30

Figure 11. Stages of program execution and SVR prediction .....................................................31

Figure 12. SimPoints chosen from the centers of program phases [106] .................................36

Figure 13. The 20 power level ranges with min and max power for 1 million instruction interval ........................................................................................................................................36

Figure 14. Best Configuration Algorithm ....................................................................................44

Figure 15. Dynamic Best configurations for each power level ....................................................45

Figure 16. Best static configuration per power level ....................................................................46

Figure 17. Adaptive architecture framework ...............................................................................47

Figure 18. Dynamic Adaptation Algorithm ....................................................................................48

Figure 19. Power improvements with fixed IPC compared to DVFS 20% ....................................50

Figure 20. IPC improvements with fixed power compared to DVFS 20% ....................................51

Figure 21. % IPC Improvement for fixed power. Note static improvement is too small to show up on the graph ....................................................................................................................................51

Figure 22. % Power reduction for fixed IPC. Note static reduction is too small to show up on the graph ....................................................................................................................................51
Figure 23. Algorithm for reducing support vectors ................................................................. 61

Figure 24. % IPC improvement per SPEC CPU2006 compared to best single configuration per power level using SVR model .................................................................................. 64

Figure 25. % IPC improvement per MiBench benchmark compared to best single configuration using SVR model ........................................................................................................... 65

Figure 26. Average IPC % Improvement PL1-8 comparison ................................................................... 66

Figure 27. Comparison % IPC improvement using the correct model and wrong model, with 30% leakage data .......................................................................................................................... 67

Figure 28. % IPC improvement per SPEC CPU2006 benchmark compared to best single configuration using AUB-SVR model .................................................................................................. 68

Figure 29. % IPC improvement per MiBench benchmark compared to best single configuration using AUB-SVR model ........................................................................................................... 69

Figure 30. Power per SimPoint per Power Level -- configuration prediction using SVR ............................................ 70

Figure 31. Power per SimPoint per Power Level – configuration prediction using ASVR ........................................ 70

Figure 32. Comparison of energy consumption (mJ) for the two types of prediction cost and for running 1 million instructions in the maxim configuration, excluding the cost of incrementing the hw counters ........................................................................................................ 74

Figure 33. Comparison of energy consumption (mJ) for the two types of prediction cost and for running 1 million instructions in the maxim configuration, including the cost of incrementing the counters ........................................................................................................ 75
TABLES

Table 1. Static portion of microarchitectural parameter configuration remains the same in all configurations........................33
Table 2. Dynamic portion of microarchitectural parameter configurations and their different values.........................................................33
Table 3. Output counters from SimpleScalar ......................................................38
Table 4. SPEC CPU2006 Benchmarks with SimPoint offset weights and inputs ........41
Table 5. MiBench Benchmarks with SimPoint offset, weight and input .............41
Table 6. Information Gain per hardware counter .............................................54
Table 7. T-test results for pairs of hardware counters ......................................56
Table 8. Inputs to SVR ..................................................................................57
Table 9. Reduced SVs with corresponding reduction in total accuracy ..............61
Table 10. Performance comparison for different SVR models showing the hardware counter used ..........................................................62
Table 11. Comparison of SVR to AUB-SVR ....................................................67
Table 12. Reduced SV model compared to regular SVR model .......................71
Table 13. Notation for Prediction Calculations ..............................................71
Table 14. Storage requirements for SVR model .............................................75
1.1 Motivation

Reducing power has become a major design concern in producing today’s microprocessors. The increasing density and speed of chips has led to power dissipation issues not only for the processor itself but also for large data centers in terms of cooling costs. Additionally, the proliferation of mobile devices with limited battery life makes power reduction even more critical. Any means of squeezing out more performance for a given amount of power, contributes positively to the environment as well as the bottom line.

General purpose processors are normally designed for best fit among a wide range of applications. Microarchitectural parameters such as cache size, issue window, TLB size, branch prediction method, etc. are selected based on their overall performance across typical user programs. This broad spectrum approach provides good performance on average to most programs. However, since power has become a forefront design issue, architectures having multi-configuration hardware have been proposed which can adapt, at run-time, to a particular application or section of an application, in an attempt to minimize power while maintaining performance [1-10]. Fig. 1 shows typical adaptive parameters.
These previous approaches in general concentrate individually on a small number of reconfigurable components at a time, for instance only cache reconfiguration [1-4], branch prediction [5], instruction fetch/issue size [6-8,26], or memory hierarchy configuration [10]. Periodically, as the application changes its resource requirements in what is known as a ‘phase’ [24], the system is ‘tuned’ and an optimal arrangement for the current phase is chosen by cycling through some small set of configurations while observing power and performance. The number of different configurations must be limited since the tuning time needed must be kept to a minimum.

Most importantly, as chip sizes decrease, there is an ever increasing amount of variability in the performance of transistors. These discrepancies can arise from either the fabrication process or even the day to day runtime activity and compute load [97].

The variability in chip performance and power as been pointed out in many previous works [100-104]. It is not possible to provide one design-time model that works across the board for all processors given this variability. Instead of providing a
‘worst case’ model, for instance with high leakage, which would not perform well on chips with low leakage, or a highly optimistic model which may cause power constraint issues on some chips, we provide a framework which adapts itself to the variations in each processor.

Our framework acknowledges chip power and performance variability and allows for training individual loadable models which can be selected based on the variations in the chip set. This leads to better performance/power gains across all chips instead of only a subset that would need to be determined at design time if our framework was not adaptable post-silicon.

1.2 Thesis Statement

Performance can be maximized in power constrained adaptive computing architectures using the machine learning approach modified Support Vector Regression to determine the best global configurations in terms of power and performance, post-silicon dynamically during application run-time.

1.3 Contribution

The following is a summary of our contributions in this work:

- A framework for determining unified maximal performance configurations for 17 different parameters of a power constrained adaptive processor at runtime using an updated support vector regression model which is modifiable post processor fabrication time
- A modified version of the support vector technique which reduces the number of support vectors for least cost computation during prediction
- AUB-SVR, our modified version of support vector regression to inhibit over estimation
- The determination of the least number of hardware counters necessary for best predictions
- The determination of a best subset of configurations for each power level
- We quantitatively demonstrate the benefits of an adaptive computing model compared to a DVFS20% model
In order to assess the feasibility of the proposed approach and to establish its benefits, experiments using a software CPU simulator were run to attain the necessary training data. This involved running a large number of simulations using 20 different benchmark programs (taken from SPEC CPU2006 [60] and MiBench [61] benchmark suites) with 486 different combinations of configuration parameters. The simulated power and performance results were measured in terms of IPC and watts and are compared to a model using Dynamic Voltage and Frequency Scaling (DVFS) and to a model with ‘ideal’ configurations as exhaustively determined from offline simulations.

We improve the scalability and find the lowest cost solution by minimizing the number of computations necessary for prediction. We devise a simple approach to reducing support vectors in the prediction computation. Finally, we determine the smallest set of hardware counters required to give best prediction results.

The need for detecting program behavior transitions (phases) was not necessary since our approach used interval sampling. Current methods for determining when to change to a new configuration rely on detecting a program ‘phase’ change [1-9]. Input to our model uses the same observable counters typically used to detect phases. Periodic sampling of these counters at set intervals was sufficient. We did not need to compare to the last set of counters to determine if a new phase had occurred. If a running application is in the same ‘phase’ at the end of the interval sampling, the hardware counters are consistently similar and the input to our model returns the same configuration as determined in the previous interval and therefore no reconfiguration would take place. Additionally, our approach allows for a more fine grained adaptation strategy since we sample every million instructions.
1.4 Dissertation Organization

This dissertation is organized as follows: Chapter 2 presents prior work and background regarding adaptive processors. We explain program phase detection which is a concept used in all these prior works and we review the relation of our previous work. Chapter 3 begins with an introduction to machine learning and the support vector regression technique. It discusses training and validation techniques for the models and the structural risk minimization. Our AUB-SVR is then explained along with its error bounds and related works. We then give an overview of the three feature extraction techniques we used in our work. Finally, we give an example of how the SVR model fits into the entire framework. Chapter 4 discusses our setup as far as configurations and cross validation. We explain SimPoints and the tools that we used and modified to run our simulations. We look at different leakage models and compare them. We discuss the different power levels and how we determine “best” configurations. We give an overview of the complete adaptive model and our dynamic adaptation algorithm. In Chapter 5 we explain our baseline models that we use for comparison. We first compare our regular SVR model to DVFS. We then adapt the SVR to fit a number of different constraints. We compare a model with reduced support vectors and a model which inhibits over predictions. We then look at the overhead involved with the adaptation framework. In Chapter 6 we present our conclusion and suggestions for future work directions.
2.1 Adaptive Microprocessors

Improving microprocessor energy efficiency by dynamically configuring resources while a program is running can lead to considerable savings in terms of power and performance. It has been shown that applications differ in their hardware demands not only from each other but also from themselves at different phases in their execution [31, 34]. The usage pattern and size requirements of major parts of the chip such as caches, instruction fetch logic and issue queues can be drastically diverse. Different program execution phases require different amounts of resources.

Typically, a standard static configuration is selected due to its average optimal performance for a large variety of applications. But these types of one-size-fits-all configurations are not necessarily the best for all programs. At certain times, hardware resources may go unused by applications. For instance a processor with a pipeline width of 6 can execute up to 6 instructions per cycle, but if characteristics of a program do not support large instruction-level parallelism (ILP) then this resource is wasted. This holds as well for programs that perhaps have relatively predictable branch prediction and hence do not need a large sophisticated branch predictor.

Tuning certain hardware parameters specifically for an application’s current needs can increase performance and save energy. Reconfigurable architectures allow for this adaptation and optimization at the microarchitectural level [42]. Significant power savings and performance gains have been shown in numerous studies [42, 50,51,52,56,57,58].
Many previous works have made run-time reconfiguration optimizations and have achieved considerable power savings. [1-10,20-22,24,26] The majority of these have addressed a small number of parameters for reconfiguration at a time such as cache size [1-3,38,47,54], TLB size [4,46] or instruction queue size [6,26]. The authors in [45] analyze the simultaneous reconfiguration of the issue-width and instruction cache for a VLIW processor. In [33] the pipeline width is reconfigured based on a threshold of the percentage of used functional units or when the IPC is below a certain level. In [37] the i-cache size is modified based on miss rates. The reorder buffer, instruction queue, load/store queue and register files are reconfigured in [39] based on increases in ROB, LQ and SQ occupancies during cache miss periods. The authors in [49] modify the issue rate based on IPC values. Few previous works take into consideration the consequences of optimizing a large number of reconfiguration parameters at that same time that may affect each other. In [36], the theoretical advantages of such approaches are examined. Our approach maximizes performance for 17 microarchitectural parameters at once, benefitting from the coordinated effect this may have on performance/power improvements.

Most earlier works use simple heuristics to determine when to change configurations. A simple search algorithm is used in [43] to find optimal cache configurations among 148 configurations by comparing energy consumption between configurations. In [55], the issue width is determined using program execution profiles. The cache configuration in [48] is selected based on periodic sampling of occupancy rates of the Issue queue, reorder buffer and load/store queue and more recently a table-driven approach was used [35]. Our framework support vector regression, a robust
function approximation technique based on statistical learning theory, to predict the optimal configuration.

Some works use machine learning on minimal resource reconfigurations. In [40], ML is used to determine which prefetch configuration to choose from 16 different patterns. In [41] decision trees are used to monitor the behavior of each cache and dynamically reconfigure it in response to changing application requirements.

The author in [89] uses machine learning techniques to deal with the complexities of microprocessor architecture/compiler co-design. They specifically use support vector machines to predict the performance of an optimizing compiler on different microarchitectural configurations for a given application.

Predictive modeling and machine learning are used in [32] to determine optimal configurations for adaptive hardware. The authors used a soft-max artificial neural network (ANN) model to predict optimal configurations. They use temporal histogram hardware counters as the input to the ANN to dynamically predict the configuration for a running program’s current phase. This work most closely relates to our work but we differ in the following manner: 1) our framework can be used on processors with different leakage percentages simply by loading the corresponding model. 2) we introduce user selectable power levels and determine configurations based on them, 3) we do not need to determine a program ‘phase’ as in their approach 4) their approach gathers counters from a ‘profiling configuration’ which has been selected as the largest configuration in order not to overflow any counters, 5) our approach uses only standard performance counters typically available in most processors while their approach uses additional non-standard counters such as temporal histograms, 6) their
approach predicts each individual parameter separately whereas our approach predicts the entire configuration at once.

Considering a large number of reconfigurable resources together is difficult due the exponentially expanding timeframe needed to explore the different combinations. However, the potential for larger energy savings based on a larger view of all available reconfigurable parameters makes this a problem of interest to be tackled.

In [21], determining a particular microarchitectural combination of reconfigurable parameters which leads to low power consumption is done by trial and error. Once a phase change is detected, a number of different configurations are cycled through and tested to find the best one for the current phase. This may take a considerable amount of time. Others such as [1] try to cut down on this tuning time by saving working set signatures and re-installing saved known configurations. Our approach directly selects a configuration based on our prediction from our SVR model.

A number of different machine learning techniques have been proposed to reduce the time needed to determine optimal microprocessor architectures at the design stages. In [11] the authors propose using optimized search algorithms and machine learning techniques to prune the architectural design space and thereby drastically reduce the time needed for determining the best configuration. The resulting selected multi-core processors' performance was shown to be within 1% of that determined by an exhaustive search of the entire design space in an order of magnitude less time. In [12] a genetic algorithm is applied to determine an optimal architecture. Their results for predicting Instructions Per Cycle (IPC) showed 2% mean error and 9% worst case error. In [13], the authors train neural networks on sample points to predict performance of
new configurations. Their models generally estimate IPC with 1-2% error and reduce required simulation by two orders of magnitude.

Other works rely on statistical methods. In [15], existing results from program benchmarks allow the authors to predict a new program’s performance by selecting specific microarchitecture-independent characteristics from both the benchmark programs and the application of interest and measuring their similarity. In [16] the authors use regression modeling with cubic splines for predicting performance and power on benchmark simulations. In this research power and performance were looked at separately and no attempt was made to determine an optimal configuration for both power and performance combined.

These last design-time approaches would not work for dynamic adaptation since they require a search of the design space at runtime. To our knowledge, none of the previous works at this time address the issue of processor variability, specifically with regard to leakage power. Our work seeks to jointly assess power and performance to select best configurations based on the currently running application’s specific resource needs for a user selected power level. Our framework can be used with different leakage models tailored to any leakage percentage therefore allowing for post-design optimization flexibility.

2.2 Program Phase Detection

It is necessary to define and discuss program phase detection in this section since it plays a major role in the majority of previous works on multi-configurable hardware [1-10,20-22,24,26]. A program phase is a period of program execution where the hardware resource demands are similar [24]. It has been shown that applications
exhibit repeating patterns of phases [27] and can have a number of long phases where program behavior is relatively stable. These periods of relative stability are considered a good time to change adaptable configuration parameters. Once a phase transition occurs, a different configuration may be more advantageous. Fig. 2 shows the phase behavior of a typical program.

Phase detection methods detect changes in a running program’s behavior over a fixed interval of time. Some set of variables is measured at fixed length sampling intervals and compared with the previous set. A new phase is detected by comparing the difference between the two sets to some threshold amount [24].

In our approach we did not specifically need to determine a phase change. The same hardware variables that provide phase change information were used as direct input to our model to determine best configurations by periodic sampling. We did not
need to compare these variables with the previous sampling since if our model predicted
the same configuration as was currently in use, we simply did not change it.

2.3 Our Related Work

In our previous research [17] we presented Interval Based Hierarchical Support
Vector Machine (IBH-SVM) for identifying optimal power aware combinations of
microarchitectural parameters from an exponentially large design space. We found
power aware configurations while considerably decreasing the number of software
benchmark simulations needed to select the most appropriate configurations. In that
work, we designed a software framework using Matlab and libsvm [25] for SVM
classification and regression. We used a modified version of this framework and
Support Vector Regression in [18] to estimate power by using a set of observed
variables that share a linear or non-linear correlation to the power consumption. We
showed that SVR can be used to cheaply and easily predict memory power usage based
on these observed variables. In [98] we use SVR to predict the CPU’s thermal status
with less than 10% percentage error using hardware counters collected from running
SPEC CPU2006 benchmarks. In that same work, we use SVR to predict current
workload with a percentage error of 0.08% for a heterogeneous training set of 6 different
integer and floating point benchmark workloads.

In [19,99] we modified the SVR technique and predicted power based on
the same observed variables but in a manner so that underestimation was limited while
still maintaining a high accuracy rate of prediction. In [69] we also modify the SVR
approach with a quadratic asymmetrical loss function to predict short term load forecasts.
for a power station. In this research we use the modifications of [19,99] except we now utilize an upper bound.
CHAPTER 3
SUPPORT VECTOR REGRESSION

In this chapter we outline some basics of machine learning and SVR that we use to predict global best power/performance configurations. We discuss training and validation techniques and some standard methods for reducing the number of feature in a dataset.

3.1 Machine Learning

Machine learning is a technology that allows computer programs to learn from data. After learning some pattern, the program can then make predictions on new data. Two major types of machine learning algorithms exist. Supervised learning, whereby the input examples are given with their corresponding output label, and unsupervised learning where examples are unlabelled and the algorithm is used to discover any existing patterns within the data. There is also semi-supervised learning which is a combination of these two techniques whereby only some of the data is labeled, and reinforcement learning where data is not labeled but the actions taken are rewarded if their outcome is successful.

Our use of machine learning in this work, is of the first type, supervised learning. In this type of learning we have some set of M examples \( \{(x_i, y_i), \ldots (x_m, y_m)\} \) to train with, and we want to learn a function \( f: X \rightarrow Y \), where \( X \) is the input space and \( Y \) is the output space. Typically \( x \) is expressed as an N-dimensional vector with each of its dimensions holding the value of some individual measurable fact. The input \( x \) is referred to as the feature set and \( y \) is referred to as the output.
In our work we will use a processor’s hardware counters as input features, and the best configuration will be the output.

3.2 Support Vector Regression (SVR) Overview

Support Vector Regression (SVR) is a machine learning technique that has proven to be an effective tool in real value function estimation. We selected it for our model for a number of reasons. Firstly, its effectiveness in high dimensional space [23] made it a good candidate for this problem since the number of observable parameters that we originally consider is relatively large. Secondly, it generalizes well due to its foundation in statistical learning theory whereby its performance can be measured [67] and we can use cross validation to determine the best model parameters. Thirdly, it allows for the use of kernel functions for non-linear information which takes into consideration the complexity of our data. And finally, the support vector regression algorithm used [25] was more readily adaptable for our different types of problem sets whereby we may need to penalize certain mispredictions more heavily than others.

In Vapnik’s ε-insensitive SVR [67], a flexible tube of minimal radius is formed symmetrically around the estimated function so that the absolute values of errors below a certain threshold ε are ignored. In this manner, points outside the tube are penalized but those within the tube, either above or below the function, receive no penalty as shown in fig. 3.

A simple output value y is predicted as:

\[ y_i = w \cdot x_i + b \]  
\( \{x_i, y_i\} \quad i = 1 \ldots L, \quad y_i \in \mathbb{R}, x \in \mathbb{R}^d \)
using the tube bounded by $\pm \varepsilon \forall i$. Where $x_i$ is a set of input data corresponding to an output $y_i$, and $w$ is a set of weights chosen to minimize the squared error between the true value ‘$t$’ and the predicted value ‘$y$’. The bias term $b$, accounts for the fact that the data may not have zero mean. The penalty function is characterized by only assigning a penalty if the predicted value $y_i$ is more than $\varepsilon$ away from the actual or ‘true’ value $t_i$ (i.e. $|t_i - y_i| \geq \varepsilon$). Those data points which lie outside the $\varepsilon$-tube are given the same penalty whether they lie above ($\xi^+$) or below ($\xi^-$) the tube ($\xi^+ > 0$, $\xi^- > 0$ $\forall i$):

$$t_i \leq y_i + \varepsilon + \xi^+$$

(2)

$$t_i \geq y_i - \varepsilon - \xi^-$$

(3)

Figure 3. SVR with $\varepsilon$-insensitive tube

The accuracy of the estimation is then measured by the loss function $L_{\varepsilon SVR}(t, y)$ and is shown in fig. 4:

$$L_{\varepsilon SVR}(t, y) = \begin{cases} 
0 & \text{if } |t - y| \leq \varepsilon \\
|t - y| - \varepsilon & \text{otherwise}
\end{cases}$$

(4)
The empirical risk is that loss which we observe from our mispredictions in experimentation on our data and is notated as:

$$R_{emp}(y) = \frac{1}{L} \sum_{i=1}^{L} L_{\varepsilon SVR}(t_i, y_i)$$

leading to the SVR error function:

$$C \sum_{i=1}^{L} (\xi_i^+ + \xi_i^-) + \frac{1}{2} ||w||^2$$

which should be minimized subject to the constraints $\xi^+ \geq 0$, $\xi^- \geq 0 \ \forall_i$ and (2) and (3). Here $C$ is a regularization constant allowing for the trade-off between training error and model complexity. A higher value of $C$ will reduce the training errors but may overfit the model and lead to poor generalization.

Introducing Lagrange multipliers: $\alpha_i^+ \geq 0$, $\alpha_i^- \geq 0$, $\mu_i^+ \geq 0$, $\mu_i^- \geq 0 \ \forall_i$

$$L_p = C \sum_{i=1}^{L} (\xi_i^+ + \xi_i^-) + \frac{1}{2} ||w||^2$$

$$- \sum_{i=1}^{L} (\mu_i^+ \xi_i^+ + \mu_i^- \xi_i^-)$$

$$- \sum_{i=1}^{L} \alpha_i^+ (\varepsilon + \xi_i^+ + y_i - t_i)$$
which leads to:

\[
\frac{\delta L_P}{\delta w} = 0 \Rightarrow w = \sum_{i=1}^{l} (\alpha_i^+ - \alpha_i^-) x_i
\]  

(8)

\[
\frac{\delta L_P}{\delta b} = 0 \Rightarrow \sum_{i=1}^{l} (\alpha_i^+ - \alpha_i^-) = 0
\]  

(9)

\[
\frac{\delta L_P}{\delta \xi_i^+} = 0 \Rightarrow C = \alpha_i^+ + \mu_i^+
\]  

(10)

\[
\frac{\delta L_P}{\delta \xi_i^-} = 0 \Rightarrow C = \alpha_i^- + \mu_i^-
\]  

(11)

Substituting (8) and (9) and maximizing \(L_P\) with respect to \(\alpha_i^+\) and \(\alpha_i^-\) \((\alpha_i^+ \geq 0, \alpha_i^- \geq 0 \ \forall j)\) where:

\[
L_D = \sum_{i=1}^{l} (\alpha_i^+ - \alpha_i^-) t_i - \varepsilon \sum_{i=1}^{l} (\alpha_i^+ + \alpha_i^-) \]

\[ - \frac{1}{2} \sum_{i,j} (\alpha_i^+ - \alpha_i^-) (\alpha_i^+ - \alpha_i^-) x_i \cdot x_j \]  

(12)

Since \(\mu_i^+ \geq 0\) and \(\mu_i^- \geq 0\) and (10) and (11), therefore \(\alpha_i^+ \leq C\) and \(\alpha_i^- \leq C\). Thus we need to find

\[
\max_{\alpha^+, \alpha^-} \left[ \sum_{i=1}^{l} (\alpha_i^+ - \alpha_i^-) t_i - \varepsilon \sum_{i=1}^{l} (\alpha_i^+ + \alpha_i^-) \right. \]

\[ \left. - \frac{1}{2} \sum_{i,j} (\alpha_i^+ - \alpha_i^-) (\alpha_i^+ - \alpha_i^-) x_i \cdot x_j \right] \]  

(13)

with \(0 \leq \alpha_i^+ \leq C, \ 0 \leq \alpha_i^- \leq C\) and \(\sum_{i=1}^{l} (\alpha_i^+ - \alpha_i^-) = 0 \ \forall i\).

Substituting (8) into (1), predictions can then be made by:
where \( k(x_i \cdot x') \) is a mathematical function called a kernel satisfying Mercer’s condition [28] which maps non-linear data into higher dimensional space where it can be linearly separated. We employ the most widely used kernel in our approach, called the radial basis function which is defined as:

\[
y' = \sum_{i=1}^{l} (\alpha_i^+ - \alpha_i^-) k(x_i \cdot x') + b
\]

(14)

The parameter \( g \) is a tunable constant.

The set \( S \) of support vectors (SV) \( x_S \) are those points which lie on or outside the tube and can be found with the indices \( i \), where \( 0 \leq \alpha_i^+ \leq C \) and \( 0 \leq \alpha_i^- \leq C \) and \( \xi_i^+ = 0 \) (or \( \xi_i^- = 0 \)). This subset of points which is usually smaller than the total number of points, determines the solution. The bias term \( b \), accounts for the fact that the data may not have zero mean and can be estimated by using one support vector and eq. 14 as:

\[
b = t_S - \varepsilon - \sum_{m \in S}^l (\alpha_m^+ - \alpha_m^-) k(x_m \cdot x_S)
\]

(16)

### 3.3 Training and Validation Techniques

Building an SVR model involves training on sample data while tuning a number of different parameters. The accuracy of the SVR technique is highly influenced by the selection of these meta parameters specific to the model. For regression, the parameters are the type of kernel, the penalty \( C \), the kernel width \( g \) and the width of the epsilon tube \( \varepsilon \). Increasing \( \varepsilon \) reduces the restriction on accuracy and decreases the number of support
vectors. $\varepsilon$ has an effect on the smoothness of the response. Tuning these parameters very specifically to one set of data may lead to what is called ‘overfitting’, and leads to poor generalization of the model. That is, the model performs inadequately on new unseen data. To estimate the accuracy of a model and to guard against overfitting, a technique called cross validation is used. With cross validation, a portion of the data is kept outside the training to be used to test the model’s accuracy on unseen data.

3.4 Structural Risk Minimization

Structural risk minimization (SRM) is an inductive principle used in machine learning to help select the best model for a given finite dataset. This principle, developed by Vapnik [67], guards against overfitting by taking the model complexity (also called capacity) into consideration and balancing it with the empirical accuracy in order to pick a model that generalizes well. Fig. 5 shows this representation where training error is our simulation accuracy results and the capacity or complexity terms are the SVR meta parameters. The more we tune these parameters specifically for our known dataset, the more we run the risk of not being able to generalize well on unseen data.

![Figure 5. Structural Risk Minimization [75]](image)
SRM states that the complexity of the model, and the empirical risk should be minimized at the same time. We train and validate using leave-one-out cross-validation to make sure we do not overfit our model. For each benchmark, we first train and build the corresponding model without including it. We observe the best overall accuracy and use the meta parameters determined to build a new model including all benchmarks. In this manner, we guard against overfitting while at the same time taking into consideration that microprocessors are designed for a specific set of programs. These programs are known and used in the design phase. If a new program is very different from the benchmark programs used to design the chip, they may not perform well and hence the decision would need to be taken as to whether this type of program is substantial enough to warrant a re-design. In our case, our model would simply need to be retrained with the new program included in the dataset.

3.5 Asymmetrical and Upper Bound SVR (AUB-SVR)

Although we have minimal over-predictions, in some cases, it may be absolutely critical when predicting configurations that we stay below a certain power constraint. In order to minimize the cases where we may possibly overestimate due to prediction error, we introduce a modified version of the SVR technique.

Our approach, Asymmetrical and Upper Bound Support Vector Regression (AUB-SVR), modifies the SVR loss functions and corresponding error functions such that the epsilon tube is only below the function as shown in Fig. 6 and Fig. 7. The penalty parameter $C$ is split into $C^+$ and $C^-$ so that different penalties can be applied to the upper and lower mispredictions. In this way, we can penalize over predictions more than under predictions.

For the $\varepsilon$-insensitive loss function Equations 3, 4 and 6 are modified as follows:
Introducing Lagrange multipliers: \( \alpha_i^+ \geq 0, \alpha_i^- \geq 0, \mu_i^+ \geq 0, \mu_i^- \geq 0 \ \forall i \)

\[
\frac{\partial L_p}{\partial w} = 0 \Rightarrow w = \sum_{i=1}^{L} (\alpha_i^+ - \alpha_i^-)x_i
\]  

\[
\frac{\partial L_p}{\partial b} = 0 \Rightarrow \sum_{i=1}^{L} (\alpha_i^+ - \alpha_i^-) = 0
\]  

where

\[
t_i \geq \xi_i^+ - y_i
\]  

\[
L_{\varepsilon-AUB-SVR}(t, y) = \begin{cases} 
0 & \text{if } 0 \leq (y - t) \leq \varepsilon \\
(t - y) & \text{if } (y - t) > \varepsilon \\
(y - t) + \varepsilon & \text{otherwise}
\end{cases}
\]  

\[
C^+ \sum_{i=1}^{L} \xi_i^+ + C^- \sum_{i=1}^{L} \xi_i^- + \frac{1}{2} \|w\|^2
\]  

Introducing Lagrange multipliers: \( \alpha_i^+ \geq 0, \alpha_i^- \geq 0, \mu_i^+ \geq 0, \mu_i^- \geq 0 \ \forall i \)

\[
L_p = C^+ \sum_{i=1}^{L} \xi_i^+ + C^- \sum_{i=1}^{L} \xi_i^- + \frac{1}{2} \|w\|^2
\]

\[
-\sum_{i=1}^{L} (\mu_i^+ \xi_i^+ + \mu_i^- \xi_i^-)
\]

\[
-\sum_{i=1}^{L} \alpha_i^+ (\xi_i^+ - y_i + t_i)
\]

\[
-\sum_{i=1}^{L} \alpha_i^- (\varepsilon + \xi_i^- + y_i - t_i)
\]
Substituting (21) and (22) and maximizing $L_D$ with respect to $\alpha_i^+$ and $\alpha_i^-$ ($\alpha_i^+ \geq 0, \alpha_i^- \geq 0 \ \forall_i$) where:

$$
\begin{align*}
\frac{\delta L_D}{\delta \xi_i^+} &= 0 \Rightarrow C^+ = \alpha_i^+ + \mu_i^+ \\
\frac{\delta L_D}{\delta \xi_i^-} &= 0 \Rightarrow C^- = \alpha_i^- + \mu_i^-
\end{align*}
$$

(23) (24)

Substituting (21) and (22) and maximizing $L_D$ with respect to $\alpha_i^+$ and $\alpha_i^-$ ($\alpha_i^+ \geq 0, \alpha_i^- \geq 0 \ \forall_i$) where:

$$
L_D = \sum_{i=1}^{l} (\alpha_i^+ - \alpha_i^-) t_i - \sum_{i=1}^{l} \alpha_i^+ - \varepsilon \sum_{i=1}^{l} \alpha_i^- \\
- \frac{1}{2} \sum_{i,j} (\alpha_i^+ - \alpha_i^-)(\alpha_i^+ - \alpha_i^-) x_i \cdot x_j
$$

(25)

Since $\mu_i^+ \geq 0$ and $\mu_i^- \geq 0$ and (23) and (24), therefore $\alpha_i^+ \leq C^+$ and $\alpha_i^- \leq C^-$. Thus we need to find
Substituting (21) into (1), we have the same prediction function and bias as in eqs. 14 and 16. It is only the selection of support vectors that differs.

\[
\max_{\alpha^+, \alpha^-} \left[ \sum_{i=1}^{L} (\alpha_i^+ - \alpha_i^-) t_i - \sum_{i=1}^{L} \alpha_i^+ - \varepsilon \sum_{i=1}^{L} \alpha_i^- \right. \\
\left. - \frac{1}{2} \sum_{i,j} (\alpha_i^+ - \alpha_i^-)(\alpha_j^+ - \alpha_j^-) x_i \cdot x_j \right]
\] 

(26)

with \(0 \leq \alpha_i^+ \leq C^+, \quad 0 \leq \alpha_i^- \leq C^- \) and \( \sum_{i=1}^{L} (\alpha_i^+ - \alpha_i^-) = 0 \forall_i \).

Substituting (21) into (1), we have the same prediction function and bias as in eqs. 14 and 16. It is only the selection of support vectors that differs.

3.6 AUB-SVR Error Bounds

Looking at the tradeoffs involved with using AUB-SVR as compared to SVR, we compare the empirical risks. By substituting the new loss function, AUB-SVR’s empirical risk becomes:

\[
R_{emp}(y) = \frac{1}{L} \sum_{i=1}^{L} L_{\varepsilon-AUB-SVR}(t_i, y_i)
\]

(27)

The maximum additional empirical risk for AUB-SVR can be computed to be:
\[ \sum_{i \in (t_i - y_i) < \varepsilon} (t_i - y_i) + \sum_{i \in (y_i - t_i) > \varepsilon} \varepsilon \]  

The actual error for AUB-SVR is bounded in the same way as for regular SVR. From Vapnik [67], the actual risk \( R(\alpha) \), with probability \( (1 - \eta) \), is bounded by:

\[ R(\alpha) \leq R_{\text{emp}}(\alpha) + \sqrt{\frac{h (\log I + 1) - \log \frac{n}{\eta}}{l}} \]  

where \( R_{\text{emp}}(\alpha) \) is the empirical risk, \( l \) is the number of training samples and \( h \) is the VC dimension or complexity. From [68] we have that,

\[ h \leq \left\lfloor \frac{A^2D^2}{m/2} \right\rfloor \]  

where \( D \) is the diameter of the smallest sphere containing all the training points, \( A \) is a constant and \( m \) is the margin or width of the \( \varepsilon \)-tube. Maximizing \( m \) leads to a smoother function and therefore less complexity of the model which leads to better generalization.

In regular SVR the margin \( m = 2\varepsilon \). In AUB-SVR we cut the epsilon tube in half so that \( m = \varepsilon \), eq. (30) still holds but with a higher error rate since a smaller margin leads to a larger \( h \).

### 3.7 AUB-SVR Related Work

Although we are not aware of other prior work specifically addressing our approach to upper bounding the SVR except our own in [19] where we use a lower bounding, we survey in this section some related work available in the literature. The authors in [90] use an asymmetric \( \varepsilon \)-insensitive loss function in Support Vector Quantile Regression (SVQR) to in an effort to decrease the number of support vectors. They modify the insensitiveness in respect to the quantile and achieve more sparseness in their model. Our work differs from theirs in that their aim was to decrease the number of
support vectors while maintaining the same accuracy as a regular SVQR, while our approach specifically seeks to limit underestimates at the possible risk of losing accuracy. Asymmetrical loss functions are discussed in [91] where the authors study different loss-functions for Bayes parameter estimation. They use a 2-sided quadratic loss function and a quasi-quadratic s-loss function and show the comparison and derive results to illustrate that this modified version shows a smaller increase of loss and can be used in real world situations where overestimation and underestimation have different importance. The authors in [92] study Bayesian risk analysis and replace the quadratic loss-function with an asymmetric loss-function to derive a general class of functions which approach infinity near the origin to limit underestimates. In [93], the authors presents a maximum margin classifier which bounds misclassification for each class differently thus allowing for different tolerances levels. In [94], the authors use a smoothing strategy to modify the typical SVR approach into a non-constrained problem thereby only solving a system of linear equations rather than a convex quadratic program. In [95], three different loss functions are compared for economic tolerance design: Taguchi’s quadratic loss function, Inverted Normal Loss Function and Revised Inverted Normal Loss Function.

3.8 Feature Extraction

We would like to select the smallest set of features necessary to make an accurate prediction with our model. In machine learning, this process is called feature extraction. We may not need to use all the counters that we have available. Some of them may contain redundant information or may be irrelevant to our prediction. We also may be able to join some of the features into a new smaller set of features
combinations. In this section we explain the different methods we attempted to reduce our feature space.

3.8.1 Principal Component Analysis

One method of reducing the dimensionality of our feature space is by using principal component analysis (PCA) [62]. This technique looks at the variance of the features and uses an orthogonal transformation to combine them into some smaller set of features while preserving almost all the same information as shown in fig. 8. The benefits of using PCA are that by combining features and then selecting the components with most variance, we may be able to get rid of some meaningless information that adversely affects our predictor. However, if variance is not the determining factor of relevance in our dataset, then we may be ignoring features that have minimal variance but are somehow essential information for predicting our function. We compare both a model using PCA and another without using PCA.

![PCA transformation](image)

Figure 8. PCA transformation
3.8.2 Entropy and Information Gain

Entropy is a measure of randomness within a feature [75]. If a feature X can take on values \{x_1, \ldots, x_n\} and has probability mass function \(P(X)\), its entropy can be calculated as:

\[
H(X) = -\sum_i P(x_i) \log_2 P(x_i)
\]  

High entropy of a feature indicates that it may not be useful or interesting as far as an indicator for our final prediction. We take the total entropy of all features and for each individual feature we subtract the entropy of the target feature from the total to find the information gain of that feature. We then rank features accordingly and attempt to remove those with lower information gain.

3.8.3 T-Test

A two sample t-test on two features measures their similarity in terms of their distributions, means and variances. It is used to determine whether the two variables are statistically different. This test is effective when the two variables being tested are independent of each other and have a normal distribution. [107] Our dataset is large (> 2000 samples) however the t-test performs well on even small datasets. The result of the test is a value between 0 and 1 which indicates to what degree the two features’ means are different from each other. The smaller the value, the more significant the difference is between the two features. In general, a value equal to or less than 0.05 is considered significant.
3.9 SVR Example

In this section, we give an example of how the SVR technique fits into the entire adaptable framework. We generate data from running each of the 60 SimPoints in each of the 486 configurations as shown in fig. 9. We use this output to determine best configurations for each SimPoint, for each power level. Then, these HW counter output vectors are used along with the corresponding best configuration as input to train the SVR.

![HW Counter Output Vector Including IPC and Power](image)

**Figure 9. An output vector for each SimPoint run on each configuration**

The hardware counter vector obtained for each SimPoint/configuration pair is unique and repeatable since a SimPoint has been chosen to represent a program phase which uses hardware resources in a similar manner.

We programmatically determine the best configurations from the data for each SimPoint at each power level. Each power level had approximately 40 ‘best configurations’. We build 20 models, one for each power level as shown in fig. 10.
Each model is trained with the 40x60 possibilities of 'best configurations' and SimPoints. Note, the 'best' configuration for one SimPoint, is not necessarily the 'best' configuration for a different SimPoint.

Figure 11 depicts the various stages of program execution in our framework. Initially, at (0), the single best static configuration is loaded into the processor. At (1), on every 1 million instruction boundary, hardware counters are input to the SVR prediction hardware (2) and if the predicted configuration is different from the current configuration, the processor is reconfigured (3).
Figure 11. Stages of program execution and SVR prediction
CHAPTER 4

MODEL SETUP AND EVALUATION METHODOLOGY

In this chapter we explain our empirical model methodology and evaluation. We used data collected from running 20 benchmarks over 486 different configurations. We explain our configurations and how they were generated. We then discuss the simulation method.

4.1 Configuration Set

When determining configurations, we did not choose all possible combinations of all parameters since some configurations do not make sense and are not necessary for simulating (such as where the LQ, SQ or RS would be bigger than the ROB). We sized the LQ, SQ and RS as fractions of the ROB size. We generated the different configurations (with a corresponding configuration number) based on the parameters in tables 1 and 2. The outer-most loop changed the width from smallest to largest which changed the corresponding ALU’s, ROB, LQ, SQ and RS. The next loops changed the il1 size, the d11 size the finally changing the d12 size was the inner most loop. We made an assumption that the increasing configurations would use an increased amount of power except for small numbers of border cases where perhaps the change in branch predictor might cause a slight power reduction. However, after data simulation on all the benchmarks and sorting by power, we saw that we needed to remap the configuration numbers to reflect the actual outcome in the data. Table 1 shows the
static part of the configurations that is the same for all. Table 2 shows the changing parameters and their values.

Table 1. Static portion of microarchitectural parameter configuration remains the same in all configurations

<table>
<thead>
<tr>
<th>Static Part of Configuration</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline length</td>
<td>10</td>
</tr>
<tr>
<td>Extra br mispred latency</td>
<td>5</td>
</tr>
<tr>
<td>Fetch speed</td>
<td>1</td>
</tr>
<tr>
<td>Return addr stack size</td>
<td>16</td>
</tr>
<tr>
<td>Memory access bus width</td>
<td>8</td>
</tr>
<tr>
<td>TLB miss latency (cycles)</td>
<td>30</td>
</tr>
<tr>
<td>dl1 latency (cycles)</td>
<td>3</td>
</tr>
<tr>
<td>dl2 latency (cycles)</td>
<td>15</td>
</tr>
<tr>
<td>il1 latency (cycles)</td>
<td>1</td>
</tr>
<tr>
<td>il2 latency (cycles)</td>
<td>15</td>
</tr>
<tr>
<td>Memory access latency &lt;first chunk&gt;</td>
<td>150</td>
</tr>
<tr>
<td>Memory system ports</td>
<td>2</td>
</tr>
<tr>
<td>BTB &lt;num_set&gt; &lt;associativity&gt;</td>
<td>1024</td>
</tr>
<tr>
<td>dtlb &lt;name&gt;:&lt;set&gt;:&lt;size&gt;:&lt;assoc&gt;:&lt;rep&gt;</td>
<td>16/4096:4:64</td>
</tr>
<tr>
<td>fetch queue size (#instr)</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 2. Dynamic portion of microarchitectural parameter configurations and their different values

<table>
<thead>
<tr>
<th>Dynamic Part of Configuration</th>
<th># of Configs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width (#instr)</td>
<td>6</td>
</tr>
<tr>
<td>Commit width (#instr)</td>
<td>6</td>
</tr>
<tr>
<td>Decode width (#instr)</td>
<td>6</td>
</tr>
<tr>
<td>Int alu</td>
<td>6</td>
</tr>
<tr>
<td>Int mult</td>
<td>6</td>
</tr>
<tr>
<td>Fp alu</td>
<td>6</td>
</tr>
<tr>
<td>Fp mult</td>
<td>6</td>
</tr>
<tr>
<td>RUU (ROB)</td>
<td>6</td>
</tr>
<tr>
<td>LQ (1/3 * ROB)</td>
<td>6</td>
</tr>
<tr>
<td>SQ (1/4 * ROB)</td>
<td>6</td>
</tr>
<tr>
<td>RS (ROB/2.5)</td>
<td>6</td>
</tr>
<tr>
<td>Branch predictor combined bimodal + gshare</td>
<td>3</td>
</tr>
<tr>
<td>bpred:comb (meta table size)</td>
<td>3</td>
</tr>
<tr>
<td>bpred:bimod (table size)</td>
<td>3</td>
</tr>
<tr>
<td>Gshare (l2size)</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th># of Configs</th>
</tr>
</thead>
<tbody>
<tr>
<td>IL1 cache</td>
<td>3</td>
</tr>
<tr>
<td>DL1 cache</td>
<td>3</td>
</tr>
<tr>
<td>DL2 cache</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>486</td>
</tr>
</tbody>
</table>
4.2 Leave-One-Out Cross Validation

Creating a supervised machine learning model typically involves training on a subset of data which is thought to be representative of the entire data space but not necessarily covering all of it. In our case, we are building a model for a known set of applications which are normally run in a data center. In regular processor design space exploration, chip parameters are selected based on some known set of benchmark applications which are important and highly utilized. If a small number of inconsequential applications do not perform well on the generally selected architecture, they are ignored. We take this into consideration when evaluating our model. Since benchmark suites are selected based on their representation of particular types of applications with different resource needs, leaving one benchmark out of our training set will overly penalize our accuracy. We would like it to be representative of how well our model will perform in a datacenter with known applications.

The meta-parameters of an SVR are critical to the outcome of the prediction accuracy. They must be tuned with care so as not to overfit the training data and render the model incapable of accurately predicting for new data. So in order to tune our model’s parameters and not overfit on our training dataset, we tune using a grid search for all SVR parameters based on a leave-one-out policy whereby for each benchmark, we build models for all the power levels without including that benchmark in the training set. We then use these parameters as the parameters for generating a new training set which includes all the benchmarks. We then check the accuracy on this model for all benchmarks.
4.3 Simpoints

Simulating each of the complete programs on each configuration would have been impossible in a reasonable amount of time so it necessitated using statistical simulation. With statistical simulation, a program’s characteristics can be condensed so that a much smaller amount of time is needed to capture these characteristics. Instead of running a whole program with possibly billions of instructions, we can cut the workload down significantly. Statistical sampling, established by [66], cuts down simulation time drastically by only simulating a portion of the program.

For our simulations, we use SimPoints as proposed by Sherwood [65], a commonly used technique in the research area of microprocessor design space exploration. In this technique, the entire program is run once and sampled at fixed intervals of instructions. Information in the form of a ‘basic block vector’ is logged in an attempt to capture the different program stages as it executes. A basic block (BB) is a sequence of executed code that has one entry and one exit point. A basic block vector (BBV) is a vector which holds the number of times each BB is executed in an interval. To generate the BBVs, the program is run in its entirety one time. Once the BBVs have been generated, the different parts of the program can then be compared using the metric of how often each BB is entered. This is due to the fact that program behavior is directly related to which part of the code is being executed. Once the program completes, the BBV’s are clustered using K-means and a characteristic set of ‘SimPoints’ with corresponding weights is provided. These points represent instruction offsets into the program that can be fast forwarded to and then full simulation is performed for the set interval of instructions. By multiplying all output counters by each
SimPoint’s corresponding weight and adding all together, a representation of the entire program is made. Fig. 12 shows SimPoints selected for a program.

![Simulation Points Chosen: gzip](image)

**Figure 12.** SimPoints chosen from the centers of program phases [106]

### 4.4 Measurement in Percent Improvement

Percent improvement is the difference between some baseline value and a new value divided by the baseline value and multiplied by 100. Our comparisons for our SVR models are based on the percent improvement of IPC using the following formula:

\[
\frac{(\text{TEST}_{\text{IPC}} - \text{BASELINE}_{\text{IPC}})}{\text{BASELINE}_{\text{IPC}}} \times 100
\]

where \( \text{TEST}_{\text{IPC}} \) is the IPC obtained using the configuration determined by our SVR model and \( \text{BASELINE}_{\text{IPC}} \) is the IPC of the best single configuration for the given power level. All comparisons are initially made on a SimPoint basis, then grouped into the
individual benchmarks which are a combination of those SimPoints. When comparing our different SVR models to each other, we use the average percent improvement over all benchmarks, over all power levels. This value is considerably lower than many of the individual performances of the benchmarks since one very badly performing benchmark can pull the whole average down even though most of the benchmarks performed well. The necessity of using this number for comparison purposes lies in the fact that we need one number which gives us an idea of overall performance.

4.5 Processor Simulator

For our simulations we modified the latest SimpleScalar ARM version 4.0 [29] and added the latest Wattch [59] power extension. This toolset is an out-of-order superscalar processor simulator integrated with a module to obtain power estimates. These tools log energy in nJ as well as numerous hardware counters.

SimpleScalar allows for modifying a processor configuration via a configuration file and outputs power and performance information including a wide range of counters such as cache hit/miss ratios, number of load/store operations performed, total power used and IPC (instructions per cycle). Wattch estimates the dynamic power consumed by calculating unit capacitances and activity factors as described in eq. 33:

\[ P_d = CV_d^2 \alpha F \]  

(33)
where \( C \) is capacitance (farads), \( V_{dd}^2 \) is supply voltage (V), \( F \) is clock frequency (Hz) and \( \alpha \) is a value between 0 and 1 representing on average how often clock ticks lead to switching activity. Table 3 lists the SimpleScalar counters that were output.

|   | sim_num_insn |   | sim_cycle |   | sim_num_refs |   | sim_num_loads |   | sim_num_branches |   | RUU_count |   | RUU_fcount |   | RS_count |   | RS_fcount |   | LQ_count |   | LQ_fcount |   | SQ_count |   | SQ_fcount |   | bpred2lev.lookups |   | bpred2lev.addr_hits |   | bpred2lev.dir_hits |   | bpred2lev.misses |   | bpred2lev.restack_pushes |   | il1.hits |   | il1.misses |   | il1.replacements |   | il1.writebacks |   | il1.invalidations |   | mem.page_count |   | mem.ptab_misses |   | mem.ptab_accesses |
|---|-------------|---|-----------|---|-------------|---|-------------|---|--------------|---|-----------|---|----------|---|---------|---|----------|---|----------|---|---------|---|----------|---|-----------|---|----------------|---|----------------|---|----------------|---|------------------|---|----------------|---|
| 1 | sim_num_insn | 24 | dl1.hits |   | dl1.hits |   | dl1.misses |   | dl1.replacements |   | dl1.writebacks |   | dl1.invalidations |   | dl2.hits |   | dl2.misses |   | dl2.replacements |   | dl2.writebacks |   | dl2.invalidations |   | itlb.hits |   | itlb.misses |   | itlb.replacements |   | itlb.writebacks |   | itlb.invalidations |   | mem.page_count |   | mem.ptab_misses |   | mem.ptab_accesses |

Wattch’s calculation of activity-factor based power measurements allows for ample approximation of relative power trends as the chip configuration is dynamically changed. We used the default parameter values of Wattch which are not current but are sufficient for comparison purposes to show the relative benefits of our approach. They are for .8um processor technology with a \( V_{dd} \) of 2.5V and a 600Mhz clock frequency
with aggressive, non-ideal conditional clocking where the power of active units scales linearly with usage.

4.6 Simulator Modifications

We modified SimpleScalar’s sim-outorder module in a number of ways. First we added Wattch to the latest ARM version of SimpleScalar. We made the pipeline length adjustable. We separated the RS from the RU and the LQ from the SQ. We added code to warm the branch predictor and cache while fast forwarding. We added code for generating BBV’s. We added code to output the counters every 1 million instructions.

Each benchmark is fast-forwarded according to SimPoints, then run in full-detail cycle-accurate mode gathering statistics for 1 million instructions.

4.7 Leakage Models

Total power is made up of two components: dynamic power and static power. Dynamic power is the power dissipated while the circuit is switching and static power is the power dissipated while the circuit is idle. Static power is also referred to as leakage power. Wattch does not account for leakage power which is becoming a dominant part of the power budget so we have estimated it based on the average energy consumption of the maximum configuration run on all benchmarks. We compare two leakage power models with a static to dynamic ratio of 30%/70% and 5%/95%.

4.8 Offline Model Building

We allow for a user selectable average power level in Watts which should not be exceeded by a running application. We build an SVR model for each of the 20 power levels. For each model we had the opportunity to tune the meta parameters differently.
However, after experimentation, all models used the same parameters. This is most likely due to the fact that the data patterns of hardware counters are similar across all power levels.

### 4.9 Data Simulation

For all our simulations we use a subset of the SPEC CPU2006 and MiBench benchmarks. SPEC CPU2006 is an industry standard, CPU intensive benchmark suite that stresses the CPU, memory and compiler of a processor. They are representative of compute intensive applications and are developed from real applications. The MiBench benchmark suite targets embedded systems based on standard user type applications. Most of the benchmarks are integer benchmarks.

We compiled the SPEC CPU2006 benchmarks using the ARM cross-compiler and used the pre-compiled MiBench ARM binaries. We first generate SimPoints for all benchmarks using an interval size of 1 million instructions and a maximum cluster size of 30. We select three SimPoints from each benchmark and reweight them to simulate running the whole program. Tables 4 and 5 show the SPEC CPU2006 and MiBench programs with their corresponding inputs (also taken from the benchmark suite), SimPoints and weights. Floating point benchmarks are marked with (fp), the others are integer. Although the four floating point benchmarks (lame, lbm, milc, specrandfp) performed well, we saw no specific trend in accuracy between floating point and integer benchmarks.
### Table 4. SPEC CPU2006 Benchmarks with SimPoint offset weights and inputs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Simpoint instr offset</th>
<th>orig. weight</th>
<th>weight</th>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mcf</td>
<td>866</td>
<td>0.101951</td>
<td>0.232096</td>
<td>inp.in</td>
</tr>
<tr>
<td>Mcf</td>
<td>2490</td>
<td>0.101951</td>
<td>0.232096</td>
<td></td>
</tr>
<tr>
<td>Mcf</td>
<td>7415</td>
<td>0.261412</td>
<td>0.552017</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.473558</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>gobmk</td>
<td>117</td>
<td>0.212</td>
<td>0.515815</td>
<td>trevor.d.ts</td>
</tr>
<tr>
<td>gobmk</td>
<td>491</td>
<td>0.096</td>
<td>0.233577</td>
<td></td>
</tr>
<tr>
<td>gobmk</td>
<td>671</td>
<td>0.103</td>
<td>0.260608</td>
<td></td>
</tr>
<tr>
<td>sjeng</td>
<td>85</td>
<td>0.264</td>
<td>0.476534</td>
<td>ref.txt</td>
</tr>
<tr>
<td>sjeng</td>
<td>583</td>
<td>0.15</td>
<td>0.270758</td>
<td></td>
</tr>
<tr>
<td>sjeng</td>
<td>671</td>
<td>0.14</td>
<td>0.252708</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.554</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Gcc</td>
<td>225</td>
<td>0.116212</td>
<td>0.31348</td>
<td>scilab.i</td>
</tr>
<tr>
<td>Gcc</td>
<td>553</td>
<td>0.169553</td>
<td>0.34169</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.307153</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>lbquantum</td>
<td>50</td>
<td>0.5255</td>
<td>0.648765</td>
<td>1.3978</td>
</tr>
<tr>
<td>lbquantum</td>
<td>157/</td>
<td>0.1195</td>
<td>0.147531</td>
<td></td>
</tr>
<tr>
<td></td>
<td>70/</td>
<td>0.165</td>
<td>0.207704</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.81</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>spectrandint</td>
<td>525</td>
<td>0.260141</td>
<td>0.384615</td>
<td></td>
</tr>
<tr>
<td>spectrandint</td>
<td>920</td>
<td>0.180067</td>
<td>0.275098</td>
<td></td>
</tr>
<tr>
<td>spectrandint</td>
<td>1125</td>
<td>0.230159</td>
<td>0.340287</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.676367</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>beip2</td>
<td>186</td>
<td>0.3185</td>
<td>0.382267</td>
<td>text.html</td>
</tr>
<tr>
<td>beip2</td>
<td>1105</td>
<td>0.1035</td>
<td>0.189214</td>
<td></td>
</tr>
<tr>
<td>beip2</td>
<td>1615</td>
<td>0.125</td>
<td>0.238519</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.547</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ibm (fp)</td>
<td>425</td>
<td>0.409205</td>
<td>0.623476</td>
<td>reference.dat</td>
</tr>
<tr>
<td>ibm (fp)</td>
<td>662</td>
<td>0.155578</td>
<td>0.237043</td>
<td></td>
</tr>
<tr>
<td>ibm (fp)</td>
<td>1783</td>
<td>0.091548</td>
<td>0.19482</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.6563288</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>milc (fp)</td>
<td>5914</td>
<td>0.112862</td>
<td>0.359356</td>
<td>su3.imp</td>
</tr>
<tr>
<td>milc (fp)</td>
<td>4105</td>
<td>0.121107</td>
<td>0.364791</td>
<td></td>
</tr>
<tr>
<td>milc (fp)</td>
<td>8786</td>
<td>0.3988215</td>
<td>0.259284</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.3319903</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>specrandint</td>
<td>631</td>
<td>0.583774</td>
<td>0.583774</td>
<td></td>
</tr>
<tr>
<td>specrandint</td>
<td>1035</td>
<td>0.416226</td>
<td>0.416226</td>
<td></td>
</tr>
<tr>
<td>specrandint</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Table 5. MiBench Benchmarks with SimPoint offset, weight and input

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Simpoint instr offset</th>
<th>original weight</th>
<th>weight</th>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math</td>
<td>63</td>
<td>0.640355</td>
<td>0.458128</td>
<td>large</td>
</tr>
<tr>
<td>Math</td>
<td>274</td>
<td>0.0660702</td>
<td>0.472906</td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td>407</td>
<td>0.06063524</td>
<td>0.068966</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.13971094</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Bitcants</td>
<td>92</td>
<td>0.161507</td>
<td>0.216216</td>
<td>1125000</td>
</tr>
<tr>
<td>Bitcants</td>
<td>171</td>
<td>0.135432</td>
<td>0.205406</td>
<td></td>
</tr>
<tr>
<td>Bitcants</td>
<td>356</td>
<td>0.430932</td>
<td>0.573379</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.746971</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>220</td>
<td>0.380312</td>
<td>0.520871</td>
<td>input_large.dat</td>
</tr>
<tr>
<td>qsort</td>
<td>492</td>
<td>0.193377</td>
<td>0.264973</td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>606</td>
<td>0.156291</td>
<td>0.214156</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.7298</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>susan</td>
<td>145</td>
<td>0.244854</td>
<td>0.391305</td>
<td>input_large.pgm</td>
</tr>
<tr>
<td>susan</td>
<td>153</td>
<td>0.75866</td>
<td>0.31884</td>
<td></td>
</tr>
<tr>
<td>susan</td>
<td>289</td>
<td>0.159151</td>
<td>0.289855</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.540071</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>tiff2bw</td>
<td>34</td>
<td>0.188811</td>
<td>0.380282</td>
<td>large.tif</td>
</tr>
<tr>
<td>tiff2bw</td>
<td>44</td>
<td>0.111888</td>
<td>0.225352</td>
<td></td>
</tr>
<tr>
<td>tiff2bw</td>
<td>64</td>
<td>0.193804</td>
<td>0.394366</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.496003</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>lame (fp)</td>
<td>697</td>
<td>0.0731518</td>
<td>0.256831</td>
<td>large.wav</td>
</tr>
<tr>
<td>lame (fp)</td>
<td>1050</td>
<td>0.0863813</td>
<td>0.303278</td>
<td></td>
</tr>
</tbody>
</table>

41
We fast-forward according to our determined SimPoints while updating the cache and branch predictor. We then run in full-detail cycle-accurate mode gathering statistics for 1 million instructions. This interval was chosen so as to be able to run all benchmarks in all 486 configurations in a reasonable amount of time. We generate a single vector for each SimPoint/configuration pair which is then used as input to our SVR model.

### 4.10 Power Levels

To determine the 20 power levels, we sort the data by power value and break it up into 20 separate segments of approximately equal size. This allows us to use the power level to determine which SVR model to apply and allows us to keep the dataset per model small enough so that the training and prediction times are much shorter than using all the data in one model. It also allows tuning of the SVR parameters on a per level basis. Grouping the data into smaller sets reduces the diameter $D$ of the smallest sphere containing all the training points. As noted in section 3.6, reducing $D$ reduces the complexity of the model. This can enhance accuracy in many cases, however, making
the model too small does not allow for patterns to be ascertained so a balance needs to be kept between shrinking the dataset size and making good predictions.

The min and max values (in Watts per million instruction interval) for all power levels are shown in fig. 13. The first and last power levels have larger differences between min and max since at these levels there was greater variation in the power values of the data at these two extremes.

![Graph showing min and max power levels](image)

Figure 13. The 20 power level ranges with min and max power for 1 million instruction interval

4.11 Determining Best Configurations

We determine two types of best configurations. 1) A set of dynamic configurations for a power level where each SimPoint may run with a different configuration, and 2) one static configuration for the power level that is able to run all SimPoints without exceeding the power constraint.

4.11.1 Best Dynamic Configurations

The best dynamic configuration for a particular SimPoint is the configuration with the highest IPC at that power level. All configurations for a power level do not
necessarily perform optimally on all SimPoints. Each configuration in the set is particular to a SimPoint or a number of SimPoints but not necessarily to all of them.

Fig. 14 shows our algorithm for determining best dynamic configurations.

For each SimPoint
N = number of features (hw counters)
L = number of different configurations
G\_i = ith configuration \( 1 \leq i \leq L \)
P\_i = output power for a single run using G\_i \( 1 \leq i \leq L \)
Sorted rows P\_i such that \( P\_1 < P\_2 < \ldots P\_i < P\_L \)
I\_i = output IPC for a single run using G\_i \( 1 \leq i \leq L \)
D\_ij = the jth feature for a single run using G\_i \( 1 \leq i \leq L, 1 \leq j \leq N \)

\[
\begin{align*}
D_{11} & \quad D_{12} \quad D_{13} \ldots D_{IN} \quad P\_1 \quad I\_1 \quad G\_1 \\
D_{21} & \quad D_{22} \quad D_{23} \ldots D_{2N} \quad P\_2 \quad I\_2 \quad G\_2 \\
D_{31} & \quad D_{32} \quad D_{33} \ldots D_{3N} \quad P\_3 \quad I\_3 \quad G\_3 \\
D_{41} & \quad D_{42} \quad D_{43} \ldots D_{4N} \quad P\_4 \quad I\_4 \quad G\_4 \\
D_{51} & \quad D_{52} \quad D_{53} \ldots D_{5N} \quad P\_5 \quad I\_5 \quad G\_5 \\
& \ldots \\
D_{L1} & \quad D_{L2} \quad D_{L3} \ldots D_{LN} \quad P\_L \quad I\_L \quad G\_L
\end{align*}
\]

To find the best configuration for a SimPoint for power below some level L, look at all the I\_i where i < L and find the highest I\_i > I\_L and replace the configuration G\_L with G\_i

\begin{verbatim}
ALGORITHM 1. FIND BEST CONFIGURATIONS
Input: L number of configurations, G\_i configurations \( 0 < i < L \), P\_i power for a run using G\_i \( 0 < i < L \), I\_i is the IPC for a run using G\_i \( 0 < i < L \)
Output: The triplets P I G;
Sort rows by P\_i such that \( P\_1 < P\_2 < \ldots P\_r < P\_i \)
for r = L : 1
   44axi = I\_r
   for s = r-1 : 1
      if I\_s > I\_r AND I\_s > 44axi
         44axi = I\_s
         G\_r = G\_s
      end
   end
end
\end{verbatim}

Figure 14. Best Configuration Algorithm
The set of dynamic best configurations per power level are shown in fig. 15. Best configurations are specifically best for a certain phase of a program. The same configuration is not necessarily best for other program phases.

Figure 15. Dynamic Best configurations for each power level

4.11.2 Best Static Configurations

The best static configuration for a power level must be able to run all the SimPoints without exceeding the power constraint. This forces the configuration to be bounded by the configuration acceptable for the most power intensive SimPoint. Fig. 16 shows the static configuration for each power level. Note that the smallest configuration for a power level is not necessarily the best. This is due to the fact that IPC does not necessarily correlate linearly with power. Some configurations that may be best for one SimPoint may cause another SimPoint to use more power but decrease its IPC.
4.12 Adaptive Model Overview

An overview of the adaptive model is shown in fig. 17. A power level is selected and the corresponding SVR model is loaded. At each interval, the hardware counters are used as input to the SVR which predicts the best configuration to reconfigure to.
4.13 Dynamic Adaptation Algorithm

The dynamic adaptation algorithm samples the hardware counters every 1,000,000 instructions. It saves the current configuration, and then uses the appropriate SVR model depending on the user selected power level. The new output configuration is checked with the current configuration and if different, the configuration is changed. If a misprediction occurs causing the power to exceed the power level constraint, the current configuration is changed to the best static configuration for that power level. Fig. 18 shows our dynamic adaptation algorithm.
ALGORITHM 2. DYNAMIC ADAPTATION

while true
    if !(pc\%1000000)
        currentConfig = config
        D = Read hw counters
        config = predictConfig(D)
        if config != currentConfig
            changeConfig(config)
        end
    end
    if avgPower > powerLevel
        config = staticConfig(powerLevel)
        changeConfig(config)
    end
end

Figure 18. Dynamic Adaptation Algorithm

4.14 Simulation Setup

To simulate the changing of configurations for running programs and to evaluate the accuracy of our model, we break the data up into the different power levels and randomly select rows corresponding to the SimPoints for each benchmark. This randomly selected row is used as input to the SVR model. The output configuration’s matching IPC and power are compared to that of the best static configuration for the power level. All SimPoint’s IPC and power are grouped into their benchmarks with their respective weights. We perform the random selection a number of times equal to the number of different configurations for that power level and take the average. This way, we are sure to select all the different configurations that would be possible at this power level for a particular SimPoint.
In this chapter we compare and evaluate the different models to show their strengths and weaknesses.

5.1 Comparison of Overall Model to DVFS

Before comparing any of the SVR models, we first look to compare our overall model to an established method for power/energy reduction called dynamic voltage and frequency scaling (DVFS). This technique is used to dynamically change the voltage and frequency of the CPU in order to obtain better power/performance based on the currently running workload needs [63]. Dynamic power is proportional to the square of the voltage so reducing the voltage can considerably reduce the power consumption. But reducing the voltage by some factor requires reducing the clock frequency by the same factor since the transistor speed is reduced. The benefit of DVFS is that it offers a cubic reduction in dynamic power while only linearly reducing performance. As for static power, DVFS reduces it linearly. Knowing when to change the voltage/frequency in a DVFS model greatly influences its effectiveness.

In order to show the efficacy of our approach, we compare our model to that of what could be achieved by a DVFS 20% model whereby we reduce the voltage by 20%. We take the maximum configuration output values of power and IPC and reduce them accordingly for our two leakage power models of 5%/95% and 30%/70%. This gives us the following formulas for computing the values:

Reduced Power = Reduced Dynamic Power + Reduced Static Power  \hspace{1cm} (34)

Reduced Power (5/95) = (max power)\times(0.95)\times(0.8)\times(0.8)+(max power)\times(0.5)\times(0.8)  \hspace{1cm} (35)
Reduced Power (30/70) = (max power)*(.7)*(.8)*(.8)+(max power)*(.3)*(.8)  \hspace{1cm} (36)
Reduced IPC = (max IPC) * (.8)  \hspace{1cm} (37)

We make two types of comparison. In the first, we fix the IPC to that of the DVFS20\% model and find the lowest power configuration that has equal or above IPC performance. For our second comparison, we fix the power to that of the DVFS20\% model and find the highest IPC configuration that has equal or below power. We compare the DVFS20\% model to both the static configuration and the dynamic configuration.

Figs. 19-22 show our results per benchmark. For our dynamic model with 5\% leakage, the average power savings was 17.81\% across all benchmarks with 4 benchmarks saving over 30\%. For the static model the average power savings was 9.62\%. The average IPC performance gain was 17.13\% for our dynamic model and 16.24\% for the static model.

The 30\% leakage model did not perform as well in comparison. For power, it had 4\% savings for the dynamic configuration and 0.1\% for the static configuration. For IPC improvement, it had 7.6\% dynamic and 0.02\% for static. Since the 30/95\% model did not show as much of an improvement compared to DVFS, we focus on the 5/95\% model in the rest of our experiments.

![5% Leakage - Fixed IPC](image)
Figure 20. IPC improvements with fixed power compared to DVFS 20%

Figure 21. % IPC Improvement for fixed power. Note static improvement is too small to show up on the graph.

Figure 22. % Power reduction for fixed IPC. Note static reduction is too small to show up on the graph.

5.2 Comparison Model Baselines

We use two different baseline models for comparison, in this section we explain their purpose and how we created them.
5.2.1 Ideal Model Baseline

The ideal model is derived from the best configurations for each SimPoint on each power level. It is the set of configurations with the highest IPC per SimPoint for the power level. This is the best dynamic performance model that can be obtained. This is how the SVR would perform if it had perfect prediction.

5.2.2 Static Model Baseline

The static model is developed based on selecting the best performing configuration that works over all SimPoints. That is, the highest IPC configuration that does not break the power level constraint for any SimPoints on that power level.

5.3 SVR Comparisons

We start by building a full SVR model using all 46 hardware counters. This is assumed to be the best performing SVR model since it includes all hardware counters i.e. all information obtainable. This model is impractical though since it takes a long time to train and predict and it uses more energy and storage than other more reasonable models.

5.3.1 Reducing SVR Computation

Computation time for the SVR has two parts; the training time, which is the time necessary to offline train the model and the prediction time which is the time needed to predict a configuration while the program is running. The training time, although not inconsequential, is much less critical since it is done once prior to implementation of the model. The prediction time is used repeatedly while the program is running and hence needs to be as efficient as possible.
Due to the nature of SVR, both these computation times are dependent on the number and dimension of the support vectors. We attempt to decrease both these in a number of different ways.

5.3.1.1 Reducing the Number of Observations

In a supervised machine learning approach, a dataset consisting of a number of observations with their corresponding outcomes is used to train a model. The number of observations included should ideally be just enough to cover all the information needed for making good predictions. On the one hand, the more observations that are available, the more information can be garnered from that data, however, large datasets can lead to unmanageably long training times. If reducing the number of observations in a dataset does not cause an intolerable decline in accuracy, then this is a viable step to take.

In our case, we decrease the number of observations in a model by breaking up the data into equally balanced power levels and building a separate, smaller more accurately tuned model for each level.

5.3.1.2 Reducing the Number of Features

The support vector’s dimension is based on the number of features. In our case, each of the 46 hardware counters is a feature. We additionally added 9 features for the current configuration which indicated the size of the pipeline width, ruu, lq, sq, rs, branch predictor, il1, dl1 and dl2 memories. Reducing the number of features corresponds to reducing the number of hardware counters that we use. There are a number of different methods for this reduction. We elaborate on the 3 that we used.
5.3.1.2.1 Reducing Features Using PCA

We used PCA to decrease the total number of features, keeping those which contributed at least 99% to the total variation, from 55 down to 8 with a 17% loss of total average accuracy (as shown in table 10), however, when comparing the energy of this model to that of a model using reduced counters (as shown in section 5.19), it was determined that reducing hardware counters is more beneficial.

5.3.1.2.2 Reducing Features Using Information Gain

We calculated the information gain for each feature and ranked them from smallest to largest. This ranking gives some idea of the overall usefulness of the feature as far as adding information to the model. We built models using the 10 highest ranking features and achieved 4.58% average improvement as shown in table 10 which was not better than our final heuristical selection as described in section 5.3.1.2.4. Table 6 shows the feature rankings obtained by this method. The smallest rank is 1 meaning that this feature contains the least amount of information.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Information Gain</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.0171</td>
<td>il1.writebacks</td>
</tr>
<tr>
<td>2</td>
<td>-0.0171</td>
<td>il1.invalidations</td>
</tr>
<tr>
<td>3</td>
<td>-0.0171</td>
<td>dl1.invalidations</td>
</tr>
<tr>
<td>4</td>
<td>-0.0171</td>
<td>dl2.invalidations</td>
</tr>
<tr>
<td>5</td>
<td>-0.0171</td>
<td>rtlb.writebacks</td>
</tr>
<tr>
<td>6</td>
<td>-0.0171</td>
<td>rtlb.invalidations</td>
</tr>
<tr>
<td>7</td>
<td>-0.0171</td>
<td>dl2.writebacks</td>
</tr>
<tr>
<td>8</td>
<td>-0.0171</td>
<td>dl2.invalidations</td>
</tr>
<tr>
<td>9</td>
<td>-0.0161</td>
<td>mem.page_count</td>
</tr>
<tr>
<td>10</td>
<td>-0.0160</td>
<td>RUU_fcount</td>
</tr>
<tr>
<td>11</td>
<td>-0.0157</td>
<td>dl2.misses</td>
</tr>
<tr>
<td>12</td>
<td>-0.0154</td>
<td>dl2.replacements</td>
</tr>
<tr>
<td>13</td>
<td>-0.0151</td>
<td>dl2.writebacks</td>
</tr>
<tr>
<td>14</td>
<td>-0.0151</td>
<td>dl2.misses</td>
</tr>
<tr>
<td>15</td>
<td>-0.0147</td>
<td>rtlb.replacements</td>
</tr>
<tr>
<td>16</td>
<td>-0.0136</td>
<td>SQ_fcount</td>
</tr>
<tr>
<td>17</td>
<td>-0.0135</td>
<td>bpred2lev.addr_hits</td>
</tr>
<tr>
<td>18</td>
<td>-0.0135</td>
<td>bpred2lev.dir_hits</td>
</tr>
<tr>
<td>19</td>
<td>-0.0135</td>
<td>LQ_fcount</td>
</tr>
<tr>
<td>20</td>
<td>-0.0134</td>
<td>bpred2lev.retstack_pushes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>21</td>
<td>-0.0133</td>
<td>dl1.writebacks</td>
</tr>
<tr>
<td>22</td>
<td>-0.0132</td>
<td>sim_num_branches</td>
</tr>
<tr>
<td>23</td>
<td>-0.0127</td>
<td>mem_ptab_misses</td>
</tr>
<tr>
<td>24</td>
<td>-0.0112</td>
<td>itlb.misses</td>
</tr>
<tr>
<td>25</td>
<td>-0.0112</td>
<td>itlb.replacements</td>
</tr>
<tr>
<td>26</td>
<td>-0.0108</td>
<td>dl1.misses</td>
</tr>
<tr>
<td>27</td>
<td>-0.0108</td>
<td>dl1.replacements</td>
</tr>
<tr>
<td>28</td>
<td>-0.0099</td>
<td>dl2.hits</td>
</tr>
<tr>
<td>29</td>
<td>-0.0095</td>
<td>SQ_count</td>
</tr>
<tr>
<td>30</td>
<td>-0.0064</td>
<td>RS_count</td>
</tr>
<tr>
<td>31</td>
<td>0.0003</td>
<td>RUU_count</td>
</tr>
<tr>
<td>32</td>
<td>0.0004</td>
<td>LQ_count</td>
</tr>
<tr>
<td>33</td>
<td>0.0006</td>
<td>mem_ptab_accesses</td>
</tr>
<tr>
<td>34</td>
<td>0.0046</td>
<td>bpred2lev.lookups</td>
</tr>
<tr>
<td>35</td>
<td>0.0132</td>
<td>dl1.hits</td>
</tr>
<tr>
<td>36</td>
<td>0.0153</td>
<td>bpred2lev.misses</td>
</tr>
<tr>
<td>37</td>
<td>0.0174</td>
<td>sim_num_refs</td>
</tr>
<tr>
<td>38</td>
<td>0.0186</td>
<td>il1.replacements</td>
</tr>
<tr>
<td>39</td>
<td>0.0186</td>
<td>ii1.misses</td>
</tr>
<tr>
<td>40</td>
<td>0.0217</td>
<td>dl1b.hits</td>
</tr>
<tr>
<td>41</td>
<td>0.0278</td>
<td>RS_fcount</td>
</tr>
<tr>
<td>42</td>
<td>0.0280</td>
<td>sim_num_loads</td>
</tr>
<tr>
<td>43</td>
<td>0.0340</td>
<td>ipc</td>
</tr>
<tr>
<td>44</td>
<td>0.0696</td>
<td>itlb.hits</td>
</tr>
<tr>
<td>45</td>
<td>0.0697</td>
<td>il1.hits</td>
</tr>
</tbody>
</table>

5.3.1.2.3 Reducing Features Using T-Test Results

We attempted to use the resulting information obtained from using a t-test on each pair of counters but this did not prove to be successful. Table 7 shows the t-test results for all pairs of hardware counters that showed some significant correlation. Cells highlighted in yellow contain t-test values which are greater than 0.5 which indicates that they are correlated. Out of the 46 features, only 10 could be removed using this criterion.
5.3.1.2.4 Reducing Features Using Heuristical Selection – Final HW Counter Set

The previous methods did not prove as successful as our heuristical selection based on expert knowledge of the dataset and multiple sampling leaving certain features out and observing the change in accuracy while performing cross validation. We started by building a model with a set of all 46 features, performing cross-validation and noting model accuracy. We then removed one feature at a time and used cross-validation to determine the accuracy of the model built without that one particular feature. We repeatedly removed each feature and looked for any relatively large difference in the accuracy to indicate that a particular feature was individually contributing to the overall accuracy. This did not prove successful since the accuracy difference from removing only one feature did not clearly demonstrate that certain features were more important than others.
We then determined to look at what seemed to be the most significant hardware counters from an expert’s point of view. It was decided that we should look at all ‘misses’, (i.e. all branch predictor and cache misses) and the IPC. We then added individual hardware counters such as total number of branch predictions, total number of references and total number of level 1 page table accesses. All while performing cross-validation and checking the accuracy. We added additional counters to our set if the accuracy through cross-validation increased more than our threshold value. The final group was selected based on the fact that we wanted to keep a minimal amount of counters while also having an overall average accuracy above 5%.

The final input vector to the SVR is made up of the following hardware counters and information as shown in table 8. Note that the current configuration is also a necessary input since the hardware counters for a given SimPoint will be different depending on the which configuration it is running on.

<table>
<thead>
<tr>
<th>Feature #</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>num.refs</td>
<td>total number of loads and stores committed</td>
</tr>
<tr>
<td>2</td>
<td>branches</td>
<td>total number of branches committed</td>
</tr>
<tr>
<td>3</td>
<td>bpred.miss</td>
<td>branch predictor misses</td>
</tr>
<tr>
<td>4</td>
<td>il1.miss</td>
<td>total number of il1 misses</td>
</tr>
<tr>
<td>5</td>
<td>dl1.miss</td>
<td>total number of dl1 misses</td>
</tr>
<tr>
<td>6</td>
<td>itlb.miss</td>
<td>total number of itlb misses</td>
</tr>
<tr>
<td>7</td>
<td>dtlb.miss</td>
<td>total number of dtlb misses</td>
</tr>
<tr>
<td>8</td>
<td>ptab.access</td>
<td>total page table accesses</td>
</tr>
<tr>
<td>9</td>
<td>ipc</td>
<td>instruction per cycle</td>
</tr>
<tr>
<td>10</td>
<td>Width</td>
<td>pipeline width</td>
</tr>
<tr>
<td>11</td>
<td>il1 ways</td>
<td>number of il1 ways</td>
</tr>
<tr>
<td>12</td>
<td>dl1 ways</td>
<td>number of dl1 ways</td>
</tr>
<tr>
<td>13</td>
<td>dl2 ways</td>
<td>number of dl2 ways</td>
</tr>
</tbody>
</table>
5.3.1.3 Reducing the Number of SVs

Specifically reducing the number of support vectors in a model can decrease both training and predictions time but most likely will incur an accuracy penalty. Depending on the nature of the problem, the decline in accuracy may be less important than the critical bounding of the computation time. We create a model whereby we specifically reduce the number of support vectors in each model by creating pseudo SV’s which are combinations of a number of similar SV’s. We do this at the expense of some accuracy and leave it as a choice to use this model or not depending on the nature of its use.

Reducing support vectors is a recognized improvement to the SVM model. Most previous works [76-88] focus on reducing the number of support vectors for classification problems. The majority of these approaches direct their attention to the training stage of the SVM [76-84]. In [76] they use clustering to partition the training data and replace a number of non-support vector points with one from the cluster. In [80] they propose a Core Vector Machine (CVM) and use the minimum enclosing ball problem. In [78] they propose Cutting-Plane Subspace Pursuit (CPSP) which selects arbitrary support vectors for a reduced training set and creates a new set from a cutting plane model. Other approaches focus on the prediction computation reducing a set of support vectors learned from a regular model [77-84]. These all work on classification problems.

Few approaches work on regression [87-88], these also attempt to reduce the support vectors during the training stage. They also attempt to completely maintain the accuracy as compared to the original model.
Our approach is for regression and is simple to compute. We are willing to trade off some accuracy in order to benefit from the savings in computation time. Our novel algorithm groups the SV's by coefficient weight, checks for similarity between the vectors, and then averages their values into a new single vector. We illustrate the case as follows:

Suppose for some set of L SV's of dimension N and their \( \alpha \) coefficients:

\[
\begin{align*}
\alpha_1 &\quad SV_{11} \quad SV_{12} \quad SV_{13} \ldots SV_{1N} \\
\alpha_2 &\quad SV_{21} \quad SV_{22} \quad SV_{23} \ldots SV_{2N} \\
\alpha_3 &\quad SV_{31} \quad SV_{32} \quad SV_{33} \ldots SV_{3N} \\
\alpha_4 &\quad SV_{41} \quad SV_{42} \quad SV_{43} \ldots SV_{4N} \\
\alpha_5 &\quad SV_{51} \quad SV_{52} \quad SV_{53} \ldots SV_{5N} \\
\vdots &
\end{align*}
\]

\[\alpha_L \quad SV_{L1} \quad SV_{L2} \quad SV_{L3} \ldots SV_{LN}\]

We want to predict for a new input vector \( q \),

\[ q_{11} \quad q_{12} \quad q_{13} \ldots q_{1N} \]

Using the SVR prediction formula:

\[
y(q) = \sum_{i=1}^{L} \alpha_i K(SV_i', q) \tag{38}
\]

Then:

\[
y(q) = \alpha_1 (K(SV_1, q)) + \alpha_2 (K(SV_2, q)) + \alpha_3 (K(SV_3, q)) + \alpha_4 (K(SV_4, q)) + \ldots + \alpha_L (K(SV_L, q)) \tag{39}
\]

Now suppose some set of SVs has the same coefficient value,

i.e., \( \alpha_1 = \alpha_2 = \alpha_3 \)

Then if \( SV_1 \sim SV_2 \sim SV_3 \)

\[
y(q) \sim 3\alpha_1 (K(SV_{AVG123}, q)) + \alpha_4 (K(SV_4, q)) + \ldots + \alpha_L (K(SV_L, q)) \tag{40}
\]

where \( SV_{AVG123} = AVG(SV_1 + SV_2 + SV_3) \)
We note that for SVR regression, the bounds of the values for this coefficient lie between \(-C\) and \(+C\). More than half of the SV’s ended up laying on one of the boundaries of the epsilon tube. These SV’s share the same coefficient value (i.e. the ones on the upper boundary of the tube share \(+C\) while the ones on the lower boundary of the tube share \(-C\)).

By grouping this way, we choose the first vector of the group and for each of the other vectors; we perform a similarity measure using Euclidean distance, to determine the closeness of each vector. If the vectors are sufficiently close based on our heuristically chosen similarity factor, we group it together with other close vectors, we take their mean, reducing down to one vector and we multiply the corresponding coefficient by the number of vectors removed \(+1\). This was effective in shrinking the support vector set by up to 22\% however this incurred an accuracy loss of 45.8\% on average over all benchmarks and power levels but many individual benchmarks performed satisfactorily. Depending on the needs of the user, this model can be selected. Our algorithm for reducing the SV’s is shown in fig. 23. Table 9 shows three cases of reduced models using this approach for different similarity factors.
Algorithm 3. Reducing Support Vectors

**Input:** \( L \) = number of features, \( N \) = number of SVs,
Vector set \( \{ V_N \} \) of the form
\( \alpha_1, F_{11}, F_{12}, \ldots, F_{1L} \)
\( \alpha_2, F_{11}, F_{12}, \ldots, F_{1L} \)
\( \ldots \)
\( \alpha_N, F_{11}, F_{12}, \ldots, F_{1L} \)

**Output:** Reduced vector set \( \{ V_R \} \) \( R < N \)
sort \( \{ V_N \} \) by coefficient

\[ \text{CoefficientCounter} = 1 \]

for \( i = 1 : N \)
\[ V_{\text{same}} = \text{set of SVs with same coefficient} \]
choose first vector in \( V_{\text{same}} \), \( = FV \)
for each vector in \( V_{\text{same}} \)
\[ \text{If vector is close to } FV \text{ using Euclidean distance} \]
\[ \text{add to } V_{\text{removed}} \]
\[ \text{CoefficientCounter++} \]
end

take the average of the removed vectors, \( V_{\text{avg}} = \text{mean}(V_{\text{removed}}) \)
multiply the coefficient by \( \text{CoefficientCounter} \)
remove the set \( V_{\text{removed}} \) from \( V_N \)
add the averaged vector \( V_{\text{avg}} \) with its correspondingly updated coefficient to \( V_R \)

Figure 23. Algorithm for reducing support vectors

Table 9. Reduced SVs with corresponding reduction in total accuracy

<table>
<thead>
<tr>
<th>Similarity within</th>
<th>Avg SVs</th>
<th>Avg Reduced SVs</th>
<th>% Reduced</th>
<th>SVR Avg. % IPC Improvement</th>
<th>Red. Avg. % IPC Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>2273</td>
<td>2156</td>
<td>5.15%</td>
<td>5.38</td>
<td>4.85</td>
</tr>
<tr>
<td>0.10</td>
<td>2273</td>
<td>2101</td>
<td>7.57%</td>
<td>5.38</td>
<td>4.83</td>
</tr>
<tr>
<td>0.30</td>
<td>2273</td>
<td>1988</td>
<td>12.54%</td>
<td>5.38</td>
<td>3.64</td>
</tr>
<tr>
<td>0.45</td>
<td>2273</td>
<td>1807</td>
<td>20.50%</td>
<td>5.38</td>
<td>2.91</td>
</tr>
<tr>
<td>0.50</td>
<td>2273</td>
<td>1772</td>
<td>22.04%</td>
<td>5.38</td>
<td>2.91</td>
</tr>
</tbody>
</table>

5.3.2 SVR Performance Comparison

The performance comparison we use is the average overall performance across all SimPoints over all power levels. This performance statistic is appreciably smaller than the individual performance of many of the benchmarks; however we need
to use one number for comparison purposes of the different models. Table 10 shows the comparison for the different models. Our optimum SVR model achieves an average 5.35% improvement in IPC across all benchmarks and all power levels.

Models with higher and lower performance are shown for comparison purposes. The first row of the table shows the average % IPC improvement while the second row shows the hardware counters used as input features. The ideal model is what can be achieved if the SVR had perfect prediction capability. The other models, such as the ones using all the hardware counters, were determined not to be energy and computation-time efficient as shown in section 5.19.

Table 10. Performance comparison for different SVR models showing the hardware counter used

| Model | Ideal | All | All PCA-8 | refs branches | bp.miss | il1.miss | dl1.miss | dl2.miss | ilfb.miss | dll.b.miss | ptab_access | ipc | Top 10 info Gain | refs bp.miss il1.miss dl1.miss dl2.miss ilfb.miss dll.b.miss ptab_access ipc |
|-------|-------|-----|-----------|---------------|---------|----------|----------|----------|-----------|------------|---------------|-----|----------------|-------------------|-------------------|
| Avg. % IPC Improvement | 10.52 | 7.93 | 6.56 | 5.38 | 4.58 | 4.18 | 3.86 | 2.63 | -0.28 | -2.76 |

Figs. 24-25 show the IPC % improvements per benchmark per power level. Here we show the performance comparison of three models. The first is the ‘Ideal’ model which is based on the best configurations as determined exhaustively from the dataset. The second model was created based on including all benchmarks in the training set. We label this model as ‘Seen’ to express the fact that this particular benchmark was seen in the training. The other model leaves the particular benchmark being compared out of the training set. We label this model as ‘Unseen’. The unseen model was used to select and tune the parameters of the SVR so as not to overfit. However, we use the
comparison results of the ‘Seen’ model since we make the assumption that when processors are designed for a datacenter, the typical applications to be run are known. Our ‘Unseen’ model overly penalizes our performance since benchmark programs in a benchmark suite are specifically chosen to be different from each other.

As can be seen, when viewed by benchmark, most have a good increase in performance overall except for math, qsort, libquantum and specrandint. These programs may be different enough from the other benchmarks such that their pattern of configuration mapping is not as readily determined.
Figure 24. % IPC improvement per SPEC CPU 2006 compared to best single configuration per power level using SVR model.
Figure 25. % IPC improvement per MiBench benchmark compared to best single configuration using SVR model.
Fig. 26 shows the aggregated results for all benchmarks. We show the % IPC improvement over all power levels and then over only power levels 1 through 8. When focusing on this subset of power levels, we see the average performance improvement is 14%. This is to point out that in general, the lower power levels saw the greatest improvements. This is due to the fact that at the higher power levels, there is less room for improvement since the maximum configurations can be used on most SimPoints without breaking the constraint.

![Figure 26. Average IPC % Improvement PL1-8 comparison](image)

### 5.3.3 Comparison of Different Leakage Models

Our primary model for simulation is the 5/95% leakage model since this was shown to be the most advantageous compared to DVFS. To show that our models are tailored to a particular leakage ratio, we build a 30/70% model and simulate the data. We then run the 30/70% data on the both models. Figure 27 shows the results that running with the wrong leakage model does not perform well. Average improvement over all benchmarks and power levels was -2.28% which is 133% worse than the correct 30/70% leakage model which had 6.76% improvement. It is noted that the 30/70% leakage model performs better than the 5/95% model when compared to the single
configuration for a power level, but since the overall performance of the 30/70% model does not compare well to DVFS it is not considered.

![30% Leakage Data](image)

Figure 27. Comparison % IPC improvement using the correct model and wrong model, with 30% leakage data

### 5.3.4 AUB-SVR Comparison

We also developed a model which strictly inhibits over predictions in power by using our AUB-SVR technique. This model gives an almost 100% decrease in the number of SimPoints breaking the power constraint with a decrease of 17% in total average accuracy compared to the regular SVR as shown in table 11. But as shown in fig. 28-29 most individual benchmarks compared favorably. Fig. 30-31 show the power over all 20 benchmarks over all power levels for the two models. The SVR model breaks the power constraint on 3% of the SimPoints. The AUB-SVR model restricts the upper bound and only one SimPoint is above the power level.

<table>
<thead>
<tr>
<th>Model</th>
<th># SimPoints Over PL</th>
<th>% of Total Over PL</th>
<th>% IPC Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVR</td>
<td>36</td>
<td>3</td>
<td>5.49</td>
</tr>
<tr>
<td>AUB-SVR</td>
<td>1</td>
<td>0.08</td>
<td>4.55</td>
</tr>
</tbody>
</table>
Figure 28. % IPC improvement per SPEC CPU2006 benchmark compared to best single configuration using AUB-SVR model
Figure 29. % IPC improvement per MiBench benchmark compared to best single configuration using AUB-SVR model
5.3.5 Reduced Support Vector Comparison

Reducing the number of support vectors lessens the computation costs. We examined 6 different similarity models based on the Euclidean distance between support vectors with the same coefficient value. Table 12 shows the reduction in SV’s which had a maximum of 22% but with an overall reduction in accuracy of 45.8%.
Table 12. Reduced SV model compared to regular SVR model

<table>
<thead>
<tr>
<th>Similarity</th>
<th>Avg # SVs</th>
<th>Avg Reduced # SVs</th>
<th>% Reduced SVs</th>
<th>Regular SVR % IPC Improvement</th>
<th>SVR with Reduced SVs % IPC Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>2273</td>
<td>2156</td>
<td>5.16%</td>
<td>5.38%</td>
<td>4.85%</td>
</tr>
<tr>
<td>0.10</td>
<td>2273</td>
<td>2101</td>
<td>7.58%</td>
<td>5.38%</td>
<td>4.83%</td>
</tr>
<tr>
<td>0.30</td>
<td>2273</td>
<td>1988</td>
<td>12.56%</td>
<td>5.38%</td>
<td>3.64%</td>
</tr>
<tr>
<td>0.40</td>
<td>2273</td>
<td>2004</td>
<td>11.85%</td>
<td>5.38%</td>
<td>3.41%</td>
</tr>
<tr>
<td>0.50</td>
<td>2273</td>
<td>1772</td>
<td>22.04%</td>
<td>5.38%</td>
<td>2.91%</td>
</tr>
</tbody>
</table>

5.4 Adaptation Overhead

In this section we discuss the energy and storage overhead of the adaptation model.

5.4.1 Prediction Energy Overhead

We estimate the energy in nJ needed to do the actual prediction by breaking the calculation up to determine the total number of multiplies, adds, and shifts involved. We then multiply these values by the corresponding SimpleScalar constant to get the total energy for one prediction. We examine two different approaches to our model, one which uses all available hardware counters and PCA and one which uses only a chosen subset of counters without using PCA. Table 13 shows the notation.

Table 13. Notation for Prediction Calculations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>original # of features</td>
</tr>
<tr>
<td>PF</td>
<td># of features after PCA</td>
</tr>
<tr>
<td>Nsv</td>
<td># of support vectors</td>
</tr>
<tr>
<td>SV</td>
<td>Support Vector, each SV has PF components after PCA</td>
</tr>
<tr>
<td>A</td>
<td>$(\alpha^+ - \alpha^-)$ coefficients (PF)</td>
</tr>
<tr>
<td>B</td>
<td>bias (1)</td>
</tr>
<tr>
<td>G</td>
<td>rbf kernel parameter (1)</td>
</tr>
</tbody>
</table>
To predict a configuration using an SVR with the PCA model we take the following steps:

**Normalization computation:**

Using minmax for each of the F features

\[ y = (ymax - ymin) \cdot (x - xmin)/(xmax - xmin) + ymin \]

\( xmin = c1 \quad (xmax - xmin) = c2 \) (constants)

\[ y = 2^*(x-c1)/(c2) \pm 1 \]

**PCA conversion:**

Subtract stored constant mean from each feature: F substracts

Compute dot product of stored eigenvectors and input vector

Stored eigenvector matrix \((PF \times F)\) . input vector \((F \times 1)\)

**RBF kernel computation:**

\( g = \) stored constant

\[ K(x_i, x) = \exp( ( g^* [ (x_i \cdot x_i) + (x \cdot x) + 2^*(x_i \cdot x)] ) \]

\[ K(x_i, x) = \exp( - ( g^*(x_i \cdot x_i) + g^*(x \cdot x) + g^*2^*(x_i \cdot x))) \]

The first dot product \((g^*(x_i \cdot x_i)\) is stored since these are SVs and \(g\) which are constants

Each of the other two dot products take

\[ (2^*PF)+3 \text{ MULTIPLIES} \]

\[ (2^*(PF-1)) + 2 \text{ ADDS} \]

\[ 1 \text{ EXP} \]

We estimate the exponential calculation using the Taylor series expansion up to the 7th term which has \(b\):

\[ e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^2}{3!} + \frac{x^2}{4!} + \ldots \]
For $n = 7$

- 6 ADDS
- 6 MULTIPLIES for numerator (saving last multiply)
- Store divisors
- 5 DIVIDES
- 1 SHIFT for divide by 2

Total Prediction Cost With PCA:

<table>
<thead>
<tr>
<th>MULTIPLIES</th>
<th>$\text{NSV} \times ((2\times PF)+15)) + (F + (PF\times F))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDS</td>
<td>$\text{NSV} \times ((2\times PF)+7) + (F + F + (PF\times (F-1)))$</td>
</tr>
<tr>
<td>SHIFTS</td>
<td>$\text{NSV} \times 1 + F$</td>
</tr>
</tbody>
</table>

Using the following estimates for the number of support vectors, features and PCA features:

- $\text{NSV} = 2200$
- $F = 51$
- $PF = 8$

Total Multiplies = $(2200\times(31)) + 459 = 68,569$
Total Adds = $(2200\times(23)) + 502 = 51,102$
Total Shifts = $2200 + 51 = 2251$

F_MULT + F_MULT_CLOCK = $3.570261 + 1.887436 = 5.457697$ nJ
F_ADD + F_ADD_CLOCK = $3.570261 + 1.887436 = 5.457697$ nJ
I_SHIFT + I_SHIFT_CLOCK = $0.505265 + 0.371641 = 0.876906$ nJ

Total Energy for Prediction with PCA = $655,102$ nJ

Without PCA, and using 11 counters:
- $\text{NSV} = 2200$
- $F = 11$

Total Prediction Cost w/out PCA using 11 features
Total MULTIPLIES = $\text{NSV} \times (1+(2\times F)+3+6+5)$
Total ADDITIONS = $\text{NSV} \times (1+(2\times (F-1))+2+6)$
Total SHIFTS = $\text{NSV} \times 1$

Total Multiplies = $(2200\times(37)) = 62,459$
Total Adds = $(2200\times(29)) = 46,502$
Total Shifts = $2200$

Total Energy for Prediction with NO PCA = $794,387$ nJ

Fig. 32 shows the comparison of the energy consumed running the average benchmark 1 million instructions in the maximum configuration (33,692,907 nJ) compared to the two prediction costs which are around 2% of this.
We would also like to show an estimate for the energy consumed by the counters. To determine this value, we take the maximum counter value from our training data to determine the number of bits needed for each particular counter. We divide the number of bits by 32 and take the percentage of energy consumed by approximately 20% of the energy of a 32 bit add. We did not use a percentage of the full energy of a 32 bit add because our counters are trivial and therefore can be designed for optimum energy efficiency and hence would use less energy than a 32 bit add on the ALU.

We find the energy consumed by 11 counters over 1 interval = 1,346,823 nJ, while the energy for 51 counters is 12,684,095 nJ.

Total Energy No PCA and 11 counters = 794,387 + 1,356,823 = 2,151,210 nJ
Total Energy with PCA and 51 counters = 655,102 + 12,684,095 = 13,339,197 nJ

Fig. 33 shows the two total energy costs. Although we show slightly better accuracy using all counters and PCA (as shown in table 10), the total energy consumption is 550% more, therefore we conclude that for minimal loss of accuracy, the model with less counters and no PCA calculation is superior.
Figure 33. Comparison of energy consumption (mJ) for the two types of prediction cost and for running 1 million instructions in the maxim configuration, including the cost of incrementing the counters.

### 5.4.2 Storage Overhead for 1 SVR Model

The storage overhead for our framework is minimal since we only store the SVR model for the current power level. The storage necessary for one SVR model in doubles is shown in table 14.

<table>
<thead>
<tr>
<th></th>
<th>number of</th>
<th>storage for one</th>
<th>total storage (double)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SV</td>
<td>2200</td>
<td>16</td>
<td>35200</td>
</tr>
<tr>
<td>coeff</td>
<td>2200</td>
<td>1</td>
<td>2200</td>
</tr>
<tr>
<td>bias</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>exp divisor</td>
<td>6</td>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td><strong>37438</strong></td>
</tr>
</tbody>
</table>

### 5.4.3 New Configuration Overhead

Changing to a new configuration requires minimal overhead which is amortized over 10’s of billions of instructions since this is the frequency with which new phases occur. The reconfiguration cost is the time it takes to flush the pipeline and then execute a sequence of move instructions to load the new configuration into the control.
registers for every reconfigurable functional block. Caches and predictors do not need to be invalidated. Therefore, the overhead in clock cycles $= 2 \times \text{pipeline\_length} + n \times \text{move\_execution\_time}$, where $n$ is the number of functional blocks that are reconfigured. We assume this amount to be negligible.
CHAPTER 6

CONCLUSION

6.1 Conclusion

We have developed a framework using a post-design-modifiable machine learning model that can be used to determine best configurations for adaptive processors improving performance by an average of 5.38% over all benchmarks and power levels and as much as 30% on some benchmarks at certain power levels. Our framework is unique in that it can adapt to leakage variability in processors by using different SVR models for different leakage percentage chips.

To show that our adaptable model is relevant and superior compared to another established power saving technique, we evaluate our SVR model compared to that of a DVFS20% model and showed an average 17.8% decrease in power and an average 17.1% improvement in IPC across all 20 benchmarks.

We developed two additional models targeting specific purposes 1) strictly limiting the number of power overestimates and 2) reducing the overall prediction computation time.

For 1) we used our AUB-SVR technique to reduce by approximately 100% the number of power overestimates. This technique is useful in situations where it is absolutely critical to always stay below a certain power constraint. Our regular SVR model stays below the power constraint 97% of the time over all SimPoints but with the AUB-SVR model we can stay beneath the power constraint almost 100% of the time while losing 17% average accuracy over all SimPoints over all power levels. This decrease in accuracy is based on the total average over all SimPoints and power levels.
but most individual benchmarks perform better than this. For 2) we used our technique for reducing the number of support vectors in the SVR model. This reduced the energy of a prediction by between 5.1% and 22% with the tradeoff of a 9.88% to 45.81% reduction in average accuracy.

We determined the least number of hardware counters necessary for best predictions (from 46 down to 11) by heuristical means after attempting to use information gain and t-test results which did not prove as successful. We compared the SVR prediction computation costs of using all the hardware counters versus using only 11 and showed a 550% reduction in energy.

We showed that leakage percentage needs to be taken into account when building a model and that using the wrong leakage model decreases average performance by 133%. Our framework recognizes that one model is not sufficient for all leakage percentages and has the ability to build separate models for each.

Finally, we determined a best performing subset of approximately 40 configurations for each power level which is 92% smaller than the original number of 486 configurations. These best configurations were determined per power level and can change depending on the leakage percentage so that each model is tailored to a specific chip leakage percent.

6.2 Future Work

Future work may include more benchmarks for a larger dataset or more targeted models for specific types of applications. We consider adding a larger number of floating point applications and focusing on the differences that may become apparent if a separation is made between program types.
Our approach has been modeled on a single processor design. Future work can show that it is easily adapted to multiprocessors by allowing each of the cores to utilize our approach on their running applications.
BIBLIOGRAPHY


85


[75] [http://svms.org/srm/srm.pdf](http://svms.org/srm/srm.pdf)


[94] Yuh-Jye Lee; Wen-Feng Hsieh; Chien-Ming Huang; , "ε-SSVR: a smooth support vector machine for ε-insensitive regression,"


