Runtime Hardware Reconfiguration using Machine Learning

Tanmay Gangwani

University of Illinois, Urbana-Champaign
gangwan2@illinois.edu

Abstract

Tailoring the machine hardware to varying needs of the software running on top of it is crucial to achieving energy-efficient execution. However, the characteristics of the code are non-deterministic and depend on system environments, precluding the use of static methods for tuning the hardware to match the requirements. We propose the use of machine learning models which can be trained to suggest energy-optimal hardware configurations, given a previously unseen application. We evaluate our model using application traces from an x86 simulator, and measure the closeness of the predicted hardware configuration to the one achieving the best energy-efficiency for the application. The machine learning model is trained using Google TensorFlow [1].

I. Introduction

The shift to multi-core hardware has put more focus on design principles that reduce the overall power consumption of the chip. This is because using excessive energy entails high cooling costs and poses a threat to physical stability of the chip. To conserve energy, hardware manufacturers provide platform features to switch off parts of the chip not in use [2]. Turning off unused area, albeit for a short while, is beneficial since transistors drain leakage current if they are powered up. Applications for a multi-core environment exhibit various phases, with each phase being potentially different in terms of the amount of the chip resources it needs for execution. Micro-managing hardware to adapt to the phases, thus, can result in substantial energy savings and cost reduction.

Adapting hardware to program phases is challenging because of the non-deterministic nature of applications. In general, a program could take different execution paths depending on user-provided inputs, while also demanding varying hardware resources. As such, statically defining the best hardware for a program phase is not possible. Our idea is to use *softmax* machine learning technique to train a model using data from pre-run program phases as input. Once trained, the model can predict the *best* hardware configuration for an unseen test program phase. Selective parts of the hardware can then be switched off to reap power benefits during the execution of the phase.

Table 1: *Input Features and Labels*

Features	Label-classes (3 values per class)
ROB occupancy (histogram)	ROB size
L1D-mpki	L1D cache size
L2D-mpki	L2D cache size
IPC	Dispatch width
Branch misprediction rate	Branch predictor size
Reuse distance (histogram)	-

II. The Model

We aim to learn a relationship between the program phases and the best hardware configurations. In machine learning jargon, the program phases form the input features, and the hardware configurations are the output labels. The program phases are characterized by a set of hardware counters measured during the run. The counters used, along with the labels, are outlined in Table 1. We consider 5 label classes, each of which is a hardware resource with possibility for reconfiguration on the fly. Each label class can take 1 of the 3 possible values, defined separately for the class - e.g. 48, 64, 96 for ROB size; 1, 2, 4 for Dispatch width. We measure the hardware counters to generate a feature vector of 44 values - 20 each for the ROB occupancy and Reuse distance histograms, and 1 each for others. We run every program phase with all the possible permutations of the labels, and select the top k configurations based the energy-efficiency values. Thus, from each program phase we get a set of *good* configurations (labels), and the corresponding hardware counters that form the feature space. We use the numbers to model a conditional distribution $P(\vec{Y}|\vec{X})$ of *good* microarchitectural configurations using soft-max [3]. Similar to the approach adopted in [4], we consider the hardware parameter values (label classes) to be independent of each other, given the counters (features):

$$P(\vec{Y}|\vec{X}) = \prod_{i=1}^{5} P(y|\vec{X}) \tag{1}$$

As noted by the authors in [4], there exist dependencies between hardware parameter values as one would expect in a hardware system. However, the model assumes that *good* parameter values are conditionally independent, given the features, and not marginally independent. Such an assumption, similar to naive Bayes, makes the problem of computing the overall conditional distribution $P(\vec{Y}|\vec{X})$ tractable.

A decision on a test sample is done as follows:

$$\vec{Y}* = \underset{\vec{Y}}{\operatorname{argmax}} P(\vec{Y}|\vec{X}), \tag{2}$$

where due to conditional independence, we can compute the maximum for each label class separately, using a different distribution $P(y|\vec{X})$. These distributions are defined by the soft-max function:

$$P(y = k|\vec{X}) = \frac{exp(\vec{w}_k \vec{X})}{\sum_{i=1}^{K} exp(\vec{w}_i \vec{X})} \tag{3}$$

III. Evaluation

I. Simulator

We use traces from single-threaded SPEC benchmarks [5] to train our model. Each trace represents a particular phase in a SPEC program. These are generated using the SimPoint methodology [6], and run on a cycle-approximate x86 simulator, Sniper [7], in detailed mode for 30M instructions following a 100M warmup phase. Sniper is modified to produce the desired performance counters, which are the *features* of our model. The hardware parameter values (labels) are changed using the Sniper-config file. We generate input data — features and labels — for 10-12 *good* configurations from each SPEC trace. Sniper uses McPAT [8] to provide energy values for each run. Energy-efficiency of a configuration is measured in terms of $\sqrt{ipc}/watt$ for the complete run of the application phase.

ipc is the number of instructions executed per cycle, and *watt* refers to the power consumption in Watts.

Since the features represent hardware counter values, their ranges differ significantly. To have bettter generalization, we standardize the feature values using Python *scikit-learn*. Next, the dimensionality of the feature space is reduced to improve computation time. We use Random Forest [9] to find the top k features for each label class, and then use them to find the parameters of the soft-max function. We train 5 soft-max functions for the 5 label classes, following our conditional independence assumption.

II. TensorFlow

TensorFlow is a rich library of machine learning algorithms from Google. It uses a data-flow graph model where the *nodes* represent mathematical operations and the *edges* represent communication of tensors between operations. We use the in-built soft-max function in TensorFlow. Gradient descent is used as the optimization algorithm in a stochastic training methodology. The error function used is cross-entropy, along with L2 regularization to prevent overfitting. The objective function is:

$$-\sum_{m} y_{.} \log y_{pred} + \frac{1}{2}||w||^2 \tag{4}$$

The TensorFlow source files, along with scripts for feature generation, training and testing are available on github. [1]

IV. Results

We run a total of 90 traces, each representing a different phase of a SPEC benchmark. Of these, data from 68 traces (739 feature vectors) is used to train the parameters of the soft-max functions. We predict the accuracy of our model on the remaining 22 traces. For the train-traces, multiple *good* configurations are generated by changing the hardware parameter values using the simulator knobs. For the test-traces, we generate the performance counters (features) by running the phase with the maximum value for each of the hardware parameters. We refer to this setting as *base*, for basic configuration, as would be the case in a system which is provisioned to run all general applications without any dynamic reconfiguration. The configuration produced by the machine learning model using the input performance counters is referred to as *pred*, for predicted configuration. We measure success in terms of the number of application phases where *pred* achieves a more energy-efficient execution than *base*. We also compare *pred* against a second configuration, referred to as *best*, for the overall most energy-efficient configuration for the phase. Such a configuration would be found by an online exhaustive search space algorithm, which is globally optimal, but hard to realize in practice.

Figure 1 plots two energy efficiency ratios - *pred/base* and *pred/best*. Out of the 22 traces, *pred* performs better than *base* in 16 of them. The maximum ratio is 1.09x, indicating a 9% improvement in energy efficiency. The ratio, averaged across all applications is 1.02x. For *pred/best*, we see an average ratio of 0.96x across applications; i.e. our prediction lags behind the exhaustive search technique by only 4%.

[1]https://github.com/tgangwani/DynReconfig

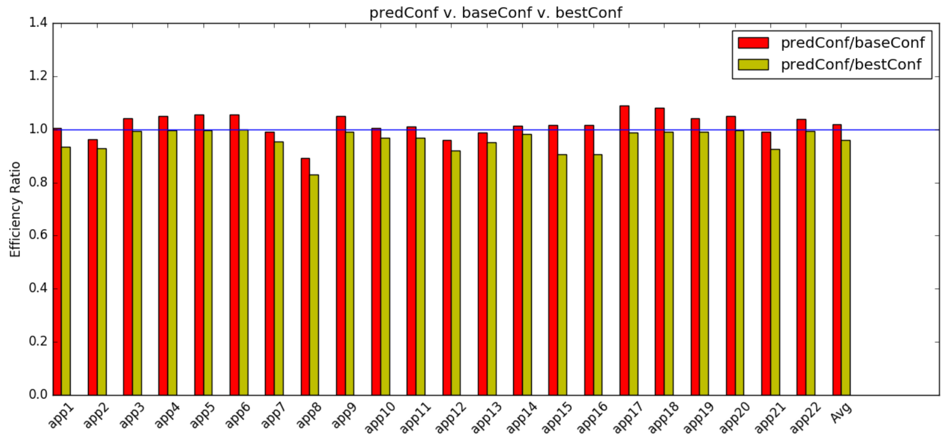

Figure 1: *Energy efficiency ratios for different configuration combinations*

I. Model Accuracy

In this subsection, we provide some discussion on the soft-max functions learned using TensorFlow. Our each input vector is of dimension $1x44$, and we have 3 possible values for each label class. Hence, we learn a $44x3$ matrix of weights and a $1x3$ vector of biases for each of the label classes, of which we have 5. We omit the label class *L2D cache size* from the discussion [2]. The accuracies on the training data for the other label classes are - ROB size (97%), L1D cache size (73%), Dispatch width (97%) and Branch predictor size (59%). This indicates difficulty in correctly learning the optimal L1D cache size and Branch predictor size based on the performance counters used in this work. We believe this to be a major source of efficiency-loss for the predicted configuration (figure 1). Correlating to the accuracy numbers, figure 2 shows how the objective (loss) function changes with iterations of the SGD algorithm, for ROB size and Branch predictor size. The error in latter converges to a value that is more than 4 times higher than the convergence value for the former.

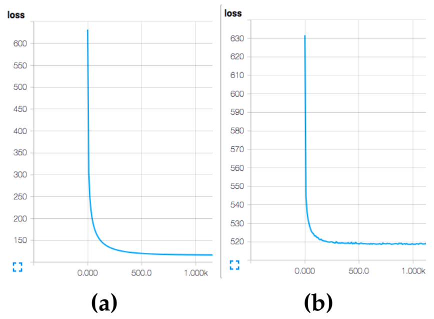

Figure 2: *Loss for ROB and Branch predictor size*

V. Conclusion and Future Work

In this work, our goal was to dynamically tune hardware to software, rather than the other way round, as done by programmers today. By learning a mapping between the data obtained from performance counters during an application run and the hardware resources required for efficient execution, we were able to suggest hardware reconfiguration opportunities for new, un-

[2] A last minute bug was found in the manner data was generated with different values for this label class. The affected feature vectors were removed from the training set.

seen applications. We implemented a collection of soft-max functions in TensorFlow to realize this goal. The results indicate that while sensitivity to some hardware structures can be captured using this technique, the impact of others are hard to predict. Future work could throw a bigger net, and evaluate more hardware resources which have a significant impact on performance and power, like load-store queues and NoC. We could also look to other algorithms like neural nets to learn complex mappings. Interestingly, we noted that our prediction accuracies are also dependent on the energy-efficiency metric chosen for optimization, i.e. learning is different when optimizing for $ipc^3/watt$ and $\sqrt{ipc}/watt$.

References

[1] https://www.tensorflow.org

[2] http://events.linuxfoundation.org/sites/events/files/slides/LinuxConPowerCapping_jpan.pdf

[3] https://en.wikipedia.org/wiki/Softmax_function

[4] Dubach, Christophe, et al. "A predictive model for dynamic microarchitectural adaptivity control." Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2010.

[5] https://www.spec.org/cpu2000/

[6] http://cseweb.ucsd.edu/čalder/simpoint/

[7] http://snipersim.org

[8] http://www.hpl.hp.com/research/mcpat/

[9] https://en.wikipedia.org/wiki/Random_forest