Machine Learning-based Prediction for Dynamic, Runtime Architectural Optimizations of Embedded Systems

Ruben Vazquez
Electrical and Computer Engineering
University of Florida
Gainesville, United States
ruben.vazquez@ufl.edu

Ann Gordon-Ross
Electrical and Computer Engineering
University of Florida
Gainesville, United States
anngordonross@ufl.edu

Greg Stitt
Electrical and Computer Engineering
University of Florida
Gainesville, United States
gstitt@ece.ufl.edu

***Abstract*—Embedded systems have been becoming increasingly complex over recent years, with performance becoming comparable to desktop computing systems. However, embedded systems need to adhere to greater design constraints (e.g., area and energy constraints) compared to desktop computing systems. Architectural specialization is a technique that can aid in meeting the stringent design constraints by introducing configurable hardware that can be tuned at runtime to optimize a goal (e.g., minimum energy, minimum execution time) for an application. However, traditional approaches (i.e., exhaustive, heuristic searches) often take considerable time to search a large design composed of different configurable parameters (e.g., cache size, associativity, etc.) and parameter values (e.g., 4 kB, 2-way, etc.). In addition, the presence of application phases (i.e., repeating execution behavior) allow for finer-grained tuning at the cost of even greater exploration overhead.**

In this paper, we apply machine learning to reduce/eliminate the design space exploration overhead associated with finding the best set of configurable parameters for configurable L1 instruction and data caches. Our prediction methodology consists of artificial neural networks (ANNs) which take the execution statistics of an application phase as input and outputs *a best* cache configuration (i.e., combination of configurable parameter values) for the instruction and data caches). Our results show that we can achieve an average energy degradation of less than 5% for the instruction and data caches with an average of 20% phase misclassification percentage and 20% less cache switches than the case where the best cache configuration is chosen for every application phase.

Keywords—Embedded systems, constraints, architectural specialization, tuning, energy, machine learning, exploration, caches, artificial neural networks, phases

I. Introduction

Embedded systems have been becoming increasingly complex over recent years. With the advent of smart devices, embedded systems, such as mobile devices, are being designed to handle general workloads at a performance level that is becoming comparable to desktop computing systems. However, the design constraints for embedded systems remain much stringent compared to desktop systems, especially with respect to area and energy consumption. As a result, techniques are required to continue improving the performance of embedded devices while adhering to the greater constraints that need to be considered for embedded system design.

Architectural specialization is one such technique that can be leveraged to meet the design constraints of embedded devices [2][10][16][19]. During architectural specialization, different hardware components (e.g., processors, caches, etc.) can be designed to be configurable, where parameters (e.g., frequency, associativity, replacement policy) can be changed to different parameter values (e.g., 1.3 GHz, 2-way, least recently used, etc.) in response to the changing workloads and requirements of the system. The process of changing the parameter values of different parameters is known as tuning. Through tuning the different configurable parameters, application workloads can see increased benefits, such as reduced energy consumption or reduced execution time. In addition, different tuning goals can be specified. For example, if reduced execution time is desired, cores can be tuned to a higher frequency to complete application workloads. A similar idea can be applied if reduced energy consumption is desired. Tuning is great for meeting the design constraints of a system, but designers need to determine what components of a system should be configurable to maximize the benefits that can be obtained through tuning.

A great option for configurability is the cache hierarchy. Based on prior work [2][8], the cache hierarchy is responsible for 50% of the energy consumed by a system, thus making the cache hierarchy an excellent choice for architectural specialization. Through tuning, the configurable parameters of the cache can be adjusted to obtain a cache configuration that yields the most benefits (e.g., minimal energy consumption, etc.) for a given application workload. In this paper, we define a cache configuration as set of parameter values that is defined for each configurable parameter. For example, a cache with configurable total size, associativity, and line size can have a valid cache configuration of (2 kB, 2-way, 32 B). For a given workload, an optimal cache configuration can be determined by running the workload with each available cache configuration and choosing the configuration that yields the most benefits. However, exploration quickly becomes infeasible for either exhaustive or heuristic searches as the configuration space increases (i.e., number of parameters increase and/or number of available parameter values increase).

Further exacerbating the issue of increasing exploration overhead, applications also exhibit patterns of repeating execution behavior, known as phases [17][22][24]. Based on

978-1-7281-2769-9/19/$31.00 ©2019 IEEE

prior work [17][22][24], any unique phase is likely to repeat during the lifetime of an application's execution. Also, since each phase's execution behavior differs from another phase's execution behavior, the cache configuration yielding the best benefits (i.e., the *optimal* cache configuration) also differs between different phases, thus requiring a new exploration for each unique phase that occurs during an application's lifetime, which is clearly infeasible for exhaustive and heuristic searches and makes the exploration overhead even worse. So, an efficient method is required to find the *optimal* cache configuration for each phase of an application.

We can use machine learning techniques [3][6][12][26] to develop an efficient method that alleviates the exploration overhead of determining the optimal cache configuration for each phase of an application. Machine learning techniques use mathematics and statistics to learn the relationship between a set of input data and a set of output data. In addition, feature reduction techniques [15][25] aid machine learning by selecting appropriate features from a feature set (i.e., CPI, instruction profile, etc.) that maximize the performance of a given machine learning technique. Instead of performing exploration exhaustively or heuristically, which can become costly as the cache configuration design space increases, we propose to use machine learning as a substitute for exploration by predicting *a best* cache configuration using a set of runtime statistics (i.e., CPI, instruction profile, etc.) gathered from a single profiling run of an application instead evaluating multiple cache configurations per application.

Much prior work has been done in performing prediction of a best configuration for some hardware component in a computing system. Khakhaeng et al. [12] used machine learning to predict the data cache line size for several data mining applications, Baldini et al. [3] used neural network works to predict the GPU performance of applications from CPU runs and to predict which GPU would yield the most benefits for a given application, Dutta et al. [6] used machine learning to predict the power of GPUs using different DVFS states, and Wu et al. [26] use machine learning models to predict the power of GPGPUs over different number of compute units, core frequencies, and memory bandwidths. However, the above prior work is quite limited. In [16], only data cache line size is predicted for data mining applications, which does not take into account any optimizations for the instruction cache, increasing configuration design space considerations, or a diverse benchmark set. In [3][6][26], only GPUs and GPGPUs were considered, but embedded systems do not use GPUs, thus limiting the effectiveness of their approach to certain computing domains.

In this paper, we propose a runtime method to predict *a best* cache configuration for each phase of an application. Our method has no restriction on the type of computing system, being widely applicable to desktop, embedded, and other computing systems with configurable hardware components. Feature reduction techniques are used to determine the best features to use for the machine learning model to make accurate predictions. We start with predicting the line sizes for the instruction and data caches and show that our model generalizes to predicting total size, associativity, and line size for the instruction and data caches.

This work was supported by the National Science Foundation (CNS-0953447 and CNS-1718033). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

In addition, we observe the mispredictions for different phases and analyze the effect of phase length on the mispredictions. Lastly, we measure the number of cache configuration switches and cache flushes incurred by the applications as a result of our prediction methodology. We show that our prediction methodology incurs an average energy degradation of less than 5%, average phase misprediction of 20%, and about an average 20% less cache switches and cache flushes compared to an approach that predicts the optimal cache configuration for each application phase.

II. Related Work

In this section, we discuss several prior works that are relevant to our methodology. Namely, we discuss works that have used machine learning to perform architectural specialization on different hardware components of a computing system, such as caches, GPUs, and GPGPUs, to show the relevance of using machine learning as a substitute to design space exploration of the cache configuration space. Further, we discuss works related to feature reduction and phases, which we incorporate into our methodology.

A. Predicton for architectural specialization

Architectural specialization is very useful technique for meeting the design constraints of embedded systems by configuring the hardware for each specific application. Khakhaeng et al. [12], designed a neural network to predict the data cache line size for several data mining applications. However, the results yield a classification accuracy of 100% for the neural network, which is unrealistic. Further, the work is limited to a configurable data cache which misses opportunities for a configurable instruction cache, the application pool is very limited, and no generalization can be supported, either to a general application workload or to a larger configuration space (i.e., including instruction cache, as well as other configurable parameters, such the total cache size and the associativity). In our paper, we greatly expand on [12] by performing predictions for both the instruction and data caches, using the concept of phases to perform finer-grained optimizations, using a larger application set, and performing an analysis of predicting line size while also generalizing the neural network's predicton capability to a larger cache configuration space (i.e., predicting total cache size, associativity, and line size).

In addition to [12], other prior works have supported the use of machine learning to aid in architectural specialization of a computing system. Baldini et al. [3] created a neural network to predict the best GPU to run an application to achieve the best speedup. The network was able to perform these predictions with accuracies above 90%. Dutta et al. [6] use machine learning techniques to perform power predictions of GPUs at different dynamic voltage and frequency scaling (DVFS) states, which can be used to achieve good energy savings. The power predictions were able to be made with a mean absolute error of 3.5.%, showing that machine learning techniques can accurately model nonlinear metrics, such as power. Wu et al. [26] created a model that can predict the performance and power of an application using different GPU configurations (i.e. number of compute units (CU), core frequency, and memory bandwidth) within 15% of real hardware. The author's show that a model can be created to estimate the performance and power of an application running on a GPU with a specific configuration, without having to evaluate all GPU

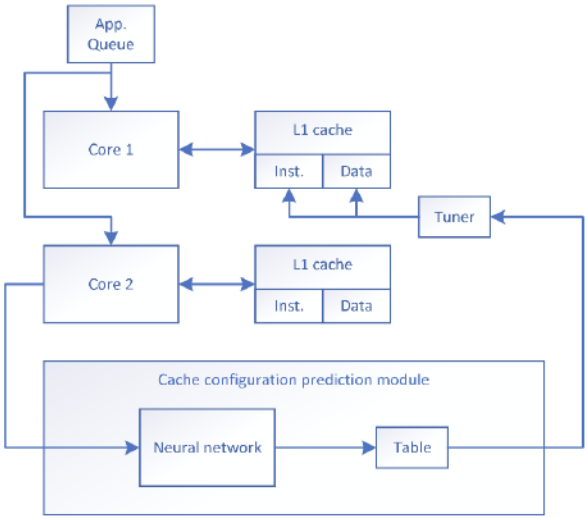

Fig. 1. Block diagram of phase-based cache configuration prediction system

configurations, thus showing the capability of a learned model as a substitute to design space exploration and evaluation. We build on the previous idea by training a machine learning model to predict the best cache configuration for an application without having to explore and evaluated all cache configurations.

B. Feature reduction

Feature reduction techniques aim at reducing the size of the input to a model to improve the performance (i.e. accuracy) of the model as well reducing the computational complexity of the model. Song et al. [25] used a technique known as principal component analysis (PCA) to reduce the number of features for a facial recognition application. Using PCA, the authors were able to reduce the number of features from 2000 to 1000, resulting in a 50% decrease of the feature space. Lu et al. [15] improved upon PCA by proposing a new method that reduces the computational burden of the PCA while still reducing the feature set for given applications. When comparing the authors' new method with PCA and other existing feature reduction methods for content-based image retrieval, the new method exhibits either better or comparable performance to PCA. In our paper, we use PCA to perform feature reduction on the features that are collected during a run of an application. Details on PCA and the collected features will be discussed in section III.

C. Phases

An application typically exhibits repeating patterns of program behavior. These repeating patterns of program behavior, known as phases, are very important in the analysis of program behavior as well as for performing finer-grained optimizations throughout an application's execution. For these reasons, much prior work has focused on determining the phases of an application. Sherwood et al. [24] design hardware that detects phase changes at runtime and predicts the next phase that will occur in an application. The authors' results show that 80% of application phases can be determined using less than 500 bytes of on-chip memory and predictions of phases can be made with an error of less than 15%. Nagpurkar et al. [17] develop a framework for online phase detection and perform an analysis of different online phase detectors. Sembrant et al. [22] develop ScarPhase, an online phase detection library that can make use of hardware counters to determine the phase of an application at runtime with less than 2% runtime overhead. In our paper, we use the idea of phases to provide finer-grained optimizations by reconfiguring a configurable cache during execution of an application and analyze the number of cache switches/flushes that are exhibited by the application.

TABLE 2:
Collected features and features description

Features	Features description
Instruction class profile	Count of instructions in a certain class (i.e., loads, unconditional branches, etc.)
Instruction profile	Count of assembly level instructions (i.e., j, lw, add, etc.)
Branch instruction profile	Count of different classes of branch instructions (i.e., unconditional direct, conditional indirect, etc.)
Addressing model profile	Count of instructions using different addressing modes
load/store address segment profile	Count of areas where memory was accessed (i.e., heap, stack, etc.)
Text and data address profiles	Count of accesses to specific text (instruction) and data addresses

TABLE 1:
Available cache configurations (size_associativity_linesize)

2K_1W_16	**2K_1W_32**	**2K_1W_64**
4K_1W_16	4K_1W_32	4K_1W_64
4K_2W_16	4K_2W_32	4K_2W_64
8K_1W_16	8K_1W_32	8K_1W_64
8K_2W_16	8K_2W_32	8K_2W_64
8K_4W_16	8K_4W_32	8K_4W_64

III. PHASE-BASED CACHE CONFIGURATION PREDICTION

Fig. 1 shows our phase-based cache configuration prediction system. We note that our methodology is entirely performed at runtime, except for the feature reduction (Section III.B), where the kept features are first determined offline and subsequently those kept features are the only features that are collected and normalized at runtime. First, if the profiling of the application has not been done, the application arrives and is sent to a profiling core to collect execution statistics for each phase of the application at runtime. For each new phase of an application, the execution statistics are collected (Section III.A) and sent to the cache configuration prediction module, where a trained artificial neural network (ANN) makes a prediction about the best cache configuration for the application's phase based on the execution statistics (Section III.C). TABLE 1 shows the cache configurations available in our system, where size and linesize are given in Bytes. The application ID, phase ID, and best cache configuration are stored for later occurrences of the profiled phase. If a profiled phase is executed again, the best configuration is retrieved from a stored table in memory.

A. Profiling and feature selection

When an application first arrives and has not been profiled, the application is sent to a dedicated core to collect the execution statistics for each unique phase. From now on, execution statistics will be referred to as features to match the terminology of later sections. While the application is executing, when a new phase is detected and that phase has not been profiled, the features for that phase are collected. Based on prior work [17][22][24], we assume that these phases are detected at runtime. The collected features are then used to make a prediction about the best cache

configuration for the phase, which can then be stored and retrieved for subsequent executions of the same phase.

TABLE 2 describes the features that are collected during the runtime profiling of the application. All the features are collected using sim-profile, which is part of SimpleScalar [4]. Section III.B describes the process of feature reduction that was used in order to reduce the described feature set significantly while retaining only the most useful features for prediction.

B. Feature reduction and normalization

Feature reduction is a technique for determining what features from the total feature set should be kept to be given as input to the neural network. In this paper, the total feature set is the execution statistics that are collected for each phase of an application at runtime. For our experiments, feature reduction is performed offline to determine what features should be kept. Once those features are determined, only those features are collected during runtime.

When collecting many features, some of the features may contain little new information or redundant information. For example, if the number of load instructions between two different phases is the same (or differ by a small amount), then we learn no new information that can help to differentiate between the phases (i.e., little or no variance). In addition, some features may contribute redundant information. For example, the number of conditional branches and the number of unconditional branches may differ by a factor of approximately two for all profiled phases. So, after choosing to keep either the number of conditional branches or the number of unconditional branches, there is no need to choose the other feature since the kept feature provides redundant information (i.e., the features are correlated).

In this paper, we use a technique called Principal Component Analysis (PCA) to reduce our total feature set. PCA is a linear technique that aims at reducing the number of features present in the total feature set by keeping only those features with the most information (i.e., high variance). Features exhibiting low variance (i.e., little or no new information) can be safely removed without significantly affecting the quality of the feature set. We refer the reader to [15][25] and Section IV.A for more details about PCA and how it is applied in our experiments.

Once the features are collected, normalization is performed on the feature set. Many of the features that are collected have values within different ranges. For example, the number of instructions for an application may range in the millions and billions whereas memory address accesses can range from very few (i.e., tens or hundreds) to very many (i.e., several hundred thousand). With different scaling for each feature, some features with very high values may end up unfairly biasing the machine learning model compared to low-valued features. To alleviate these issues, normalization is performed to appropriately scale each feature with respect to each other feature so that no single feature biases the machine learning model excessively. The details of how the normalization is performed is discussed further in Section IV.A.

C. Neural network model

After the features have been collected and normalized, the features are then given as input to the machine learning model. In our experiment, we chose to use the artificial neural network (ANN) for our machine learning model. ANNs have many different advantages compared to other machine learning models. Linear regression models map inputs to outputs with a linear relation. While useful, the relationship between best cache configuration, which is based on the application's energy consumption, and the input features cannot be modeled well by linear methods. Decision trees are also very useful, in terms of simplicity and interpretation. However, decision trees may bias certain outcomes unfairly (i.e., greater depth for different cache configurations). ANNs, on the other hand, can model nonlinearities, have been proven to exhibit the property of universal approximation, and can be easily parallelized, which is a focus of future work.

ANNs are typically structured with the following layers: an input layer, one or more hidden layers, and an output layer. Each layer contains a number of processing elements, which takes in the input and calculates the output through a non-linear function. Each layer is also fully-connected (i.e., the outputs of all processing elements in a layer are connected to the inputs of all processing elements in the next layer). In our experiments, we create two ANNs to predict the best cache configurations for the instruction and data cache for different application phases.

The features collected during profiling are given as inputs to the ANNs. In our experiments, after varying the number of layers and the number of processing elements per layer, we set each network with one hidden layer containing 10 processing elements and an output layer containing 18 processing elements, corresponding to the 18 available cache configurations. The outputs of the ANNs are given as 1-hot outputs, where for each given input, exactly one of the outputs is close to one while the others are zero, corresponding to the predicted cache configuration. The details of the setup and training of the ANNs are given in Section IV.A.

When considering machine learning models, such as artificial neural networks, we need to also consider the impact of the network in terms of area and energy consumption. We cannot assume that our design has negligible area and energy overheads, especially for mobile devices, which have strict design constraints with respect to both area and energy. Fortunately, much prior work has been done [9][13][14][20] for porting ANNs to mobile phones and FPGAs that are very area and energy efficient. We leave the hardware overhead analysis of our design as future work but note that machine learning models, such as convolutional neural networks and deep neural networks, which are complex compared to our design, have been successfully ported to mobile devices with acceptable area and energy overheads [9][13][14][20], thus demonstrating the feasibility of using neural networks on mobile devices and other area and energy-constrained devices.

IV. Experimental Results

A. Setup

All of our experiments are executed on an Intel® Xeon® E5520 CPU running at 2.25GHz. Fig. 1 shows the block diagram of our system with the phase-based cache configuration prediction module. When an application arrives, the system determines whether the application has been profiled. If the application has been profiled, the best configurations have already been predicted and are stored in the table. Then, for each phase of the application, the best configuration is retrieved, and the caches are tuned to the

energy(total) *= energy(fill) + energy(dynamic) + energy(CPU_stall) + energy(static)*

energy(fill) *= cache_misses*(line_size/8)*energy(offchip_access_perword)*

energy(dynamic) *= (cache_accesses*energy(dynamic_per_access))/1000000000*

energy(CPU_stall) *= (miss_cycles*energy(CPU_stall_per_cycle))/1000000000*
miss_cycles *= cache_misses*penalty*
penalty *= 40 if line_size = 16*
penalty *= 60 if line_size = 32*
penalty *= 90 if line_size = 64*

energy(static) *= (((cache_misses*penalty) + cache_accesses)*(15%*energy(dynamic_average)))/1000000000*

Fig. 2. Energy model for the L1 instruction and data cache

retrieved configuration. Otherwise, the application phases are profiled by collecting the execution statistics and sending the collected execution statistics to the two ANNs to predict the best instruction and data cache configuration for each phase of the application. The predicted cache configurations, along with the application ID and phase ID are stored in the table for subsequent retrieval when the phase is executed again.

Our configurable cache design is based on the design by Zhang et al. [27]. The three configurable parameters available are cache size, associativity, and line size. Way shutdown is used to tune the cache size and way concatenation is used to tune the associativity. The physical line size of the cache is 16B, however the logical line size can be tuned be fetching different numbers of blocks from memory into the cache. We also note that other configurable designs are available, such as the design by Navarro et al. [18]. However, the design by Navarro et al. focuses on a configurable associativity only so we focus on the design by Zhang et al.

In our experiments, we used benchmarks from the EEMBC AutoBench [7] and MiBench [11] benchmark suites as our application set. The AutoBench suite contains benchmarks that are representative of kernels executed on embedded devices in the automotive, industrial, and Fig. 1general-purpose applications. The MiBench suite also contains kernels that are representative of workloads on embedded devices, containing benchmarks from a diverse range of domains, including automotive, network, and security domains. We use all the benchmarks from EEMBC and most of the benchmarks from MiBench. Some of the MiBench benchmarks could not be cross compiled to run on the SimpleScalar [4] simulator. The benchmarks that could not be compiled were the only ones that we omit. In addition, we run the MiBench benchmarks with both the small and large inputs to completion to capture varying execution behavior based on input sizes. These benchmarks constitute a wide variety of diverse application domains with varying input sizes to represent an embedded systems workload.

The profiling of the application phases is done using sim-profile, which is part of the SimpleScalar [4] suite of simulators, with the -all option. We also modified sim-profile to collect the data address profile in addition to the text address profile. SimPoint [23] is used to determine the phases for the applications. We use SimPoint with a phase interval length of 100000 instructions to determine the phases of an application. We note that many different phase interval lengths can be chosen; however, prior work shows that the trends for different phase interval lengths are consistent [5]; therefore, we use a length of 100000 to determine a diverse amount of phases for our benchmark workload, which execute fewer instructions compared to other standard benchmark suites targeted for desktop computing systems.

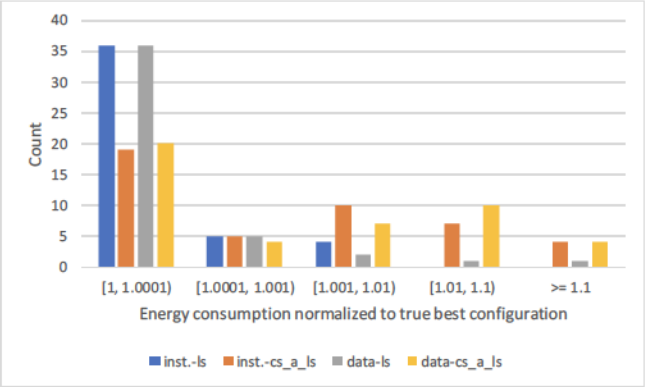

Fig. 3. Energy consumption normalized to the true best configuration of benchmarks for different cache configuration spaces

We use Principal Component Analysis [15][25] to determine what execution statistics to collect at runtime. Specifically, Principal Component Analysis is used offline to perform the feature reduction of the execution statistics by keeping 95% of the variance (i.e., information) of the execution statistics. Once the execution statistics that we will collect have been determined, only those determined execution statistics will be collected at runtime. By performing Principal Component Analysis, we are able to reduce the feature set to about 100 features as compared to over 10000 features that we are able to collect from the profile, thereby achieving at least a 100X reduction in the execution statistics required to make a prediction about the best instruction and data cache configurations for application phases.

The training and inference (i.e., prediction) of the artificial neural networks was done using TensorFlow [1]. The two models (one for instruction cache configuration prediction and one for data cache configuration prediction), each contain an input layer, one hidden layer containing 10 processing elements with rectified linear unit activation, and an output layer containing 18 processing elements with softmax activation. The training set, validation set, and testing set consisted of 60%, 20%, and 20% of the application phases, respectively. The machine learning models were trained using the RMSprop optimizer using categorical crossentropy as the loss function.

Fig. 2 shows the energy model that we use to calculate the energy consumed by the instruction and data cache. The cache statistics (e.g., cache accesses and cache misses) required by the model are obtained by running the application phases using sim-cache, which is part of the SimpleScalar [4] suite of simulators. The cache specific values, (e.g., energy(CPU_stall_per_cycle), energy(dynamic_per_access), etc.) were obtained using CACTI [21] for a 0.18 µm technology node for each cache configuration. We use the energy model to calculate the energy consumed by the instruction and data caches for each application phase for every available instruction and data cache configuration. The energy degradation for each application phase is then determined by computing the ratio of the energy consumption of the application phase using the predicted cache configuration to the energy consumption of the application phase using the optimal cache configuration.

B. Results

Fig. 3, Fig. 4, and Fig. 5 display our results. In summary, our approach incurs less than a 5% energy degradation over

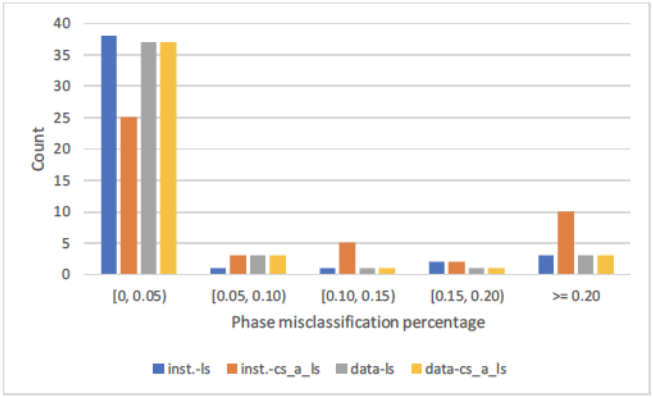

Fig. 4. Phase misclassification percentage of benchmarks for different cache configuration spaces

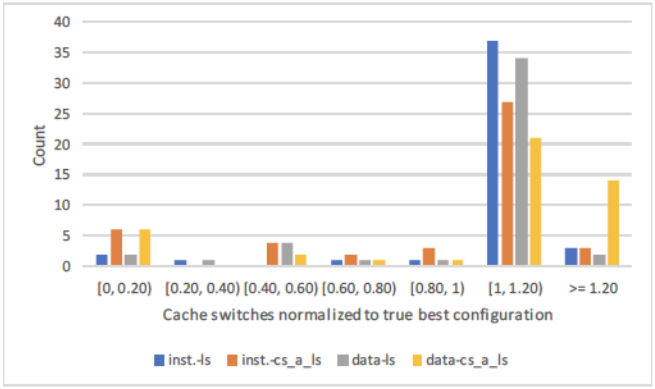

Fig. 5. Cache switches normalized to true cache switches of benchmarks for different cache configuration spaces

all the benchmarks when predicting the instruction and data cache configurations. In addition, our results show that our prediction methodology can attain an average phase misprediction of 20% with 20% less cache switches and cache flushes compared to perfectly predicting the instruction and data cache configuration for each benchmark. The figures show our results when our configuration space includes only a configurable line size (inst.-ls and data-ls) and a configurable cache size, associativity, and line size (inst.-cs_a_ls and data-cs_a_ls). All the results are normalized to the case where the instruction and data cache configurations are the true best configuration for each benchmark phase. We analyze the results in the next paragraphs.

Fig. 3 shows the energy consumption of the benchmarks when the prediction is performed over different cache configuration spaces. We can see from the results that most of the benchmarks exhibit less than 1% energy degradation, which generalizes to larger design spaces with more configurable parameters and configurable parameter values. There are some exceptions, such as *qsort* and *search*, which exhibit higher energy degradations due to less predictable control flow (e.g., quicksort algorithm for qsort) resulting in higher phase misprediction. However, the benchmarks, *qsort* and *search*, exhibit about a 20% energy degradation which does not significantly offset the gains from lower exploration overhead while maintaining acceptable energy consumption.

Fig. 4 shows the phase misclassification percentage of the benchmarks when the prediction is performed over different cache configuration spaces. The phase misclassification is kept very low, with very few exceptions. However, even for benchmarks with phase misclassification percentages over 20%, energy degradation stays below 5% for most benchmarks, supporting the existence of near-best cache configurations for phases such that energy consumption stays close to the optimal (i.e., minimum) for the phases. In addition, our larger configuration spaces show similar trends, supporting the generalization of our methodology to design spaces with more configurable parameters and/or more configurable parameter values.

Fig. 5 shows the cache switches of the benchmarks when the prediction is performed over different cache configuration spaces. Cache switches should be kept low to reduce the impact of cache flushes. From the figure, we can see that for most of the benchmarks, the number of cache switches are about equal compared to the case when the cache configuration is predicted perfectly. In some cases, misprediction may reduce the number of cache switches while also keeping the energy performance at an acceptable level (i.e. energy degradation of about 5%). When generalizing to larger design spaces, the figure also supports the same trends.

However, for some benchmarks (i.e., *AIFFTR01* and *patricia*), low data reuse causes very high data cache switching. For these cases, we develop a finite-state machine to stop cache configuration switching when the configuration is switching too often. In these cases, we can reduce the cache switching significantly while maintaining near-optimal energy performance. In addition, short phase interval lengths (we use 100000 instructions in this paper) can cause the cache to switch often. In future work, we plan to expand our FSM to include more options for reducing the cache switching frequency and will explore dynamically changing the phase interval length to reduce cache switching frequency and to create machine learning models that are independent of the phase interval length with respect to the collected features.

V. Conclusions and Future Work

In this paper, we presented a runtime methodology to predict *a best* cache configuration for the instruction and data caches for each application phase. We analyzed the energy degradation, phase misclassification, and cache switches/flushes over all of the applications, which showed that our methodology can achieve an average energy degradation of less than 5%, a phase misclassification percentage of 20%, and 20% less cache switches/flushes compared to a method that can predict the optimal cache configuration for the instruction and data caches for each application phase.

Currently, we are working on building our neural network models on FPGAs to achieve significant speedup and hardware overhead reduction compared to running the models in software alongside our applications. For future work, we plan to explore dynamically changing phase interval lengths for cache switching reduction, unsupervised machine learning models for training time reduction, and methods of determining phase similarity across applications.

References

[1] M. Abadi *et al.* TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M. H. Alsafrjalani and A. Gordon-Ross, "Quality of service-aware, scalable cache tuning algorithm in consumer-based embedded devices," *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, Boston, MA, 2016, pp. 357-360.

[3] I. Baldini, S. J. Fink and E. Altman, "Predicting GPU Performance from CPU Runs Using Machine Learning," *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, Jussieu, 2014, pp. 254-261.

[4] D. Burger, T.M. Austin, S. Bennet, "Evaluating future microprocessors: the Simplescalar tools set". Technical Report CS-TR-1996-1308, CS. Dept., Univ. of Wisconsin, Madison, Aug. 1996.

[5] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, San Diego, CA, USA, 2003, pp. 217-227.

[6] B. Dutta, V. Adhinarayanan, and W. Feng. 2018. GPU power prediction via ensemble machine learning for DVFS space exploration. In Proceedings of the 15th ACM International Conference on Computing Frontiers (CF '18). ACM, New York, NY, USA, 240-243.

[7] EEMBC. The Embedded Microprocessor Benchmark Consortium http://www.eembc.org/benchmark/automotive _sl.php, Sept. 2013.

[8] S. Gianelli, E. Richter, D. Jimenez, H. Valdez, T. Adegbija and A. Akoglu, "Application-Specific Autonomic Cache Tuning for General Purpose GPUs," *2017 International Conference on Cloud and Autonomic Computing (ICCAC)*, Tucson, AZ, 2017, pp. 104-113.

[9] V. Gokhale, J. Jin, A. Dundar, B. Martini and E. Culurciello, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, Columbus, OH, 2014, pp. 696-701.

[10] A. Gordon-Ross, F. Vahid, N. Dutt. IEEE/ACM International Symposium on Low Power Electronics and Design, August 2005.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01). IEEE Computer Society, Washington, DC, USA, 3-14.

[12] S. Khakhaeng and C. Chantrapornchai, "On the finding proper cache prediction model using neural network," *2016 8th International Conference on Knowledge and Smart Technology (KST)*, Chiangmai, 2016, pp. 146-151.

[13] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, D. Shin. (2016). Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications.

[14] N. D. Lane *et al*., "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices," *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Vienna, 2016, pp. 1-12.

[15] Y. Lu, I. Cohen, XS. Zhou, and Q. Tian, "Feature selection using principal feature analysis," in Proceedings of the 15th international conference on Multimedia. ACM, 2007, pp. 301–304.

[16] A. Martins *et al*., "Configurable Cache Memory Architecture for Low-Energy Motion Estimation," *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, Florence, Italy, 2018, pp. 1-5.

[17] P. Nagpurkar, P. Hind, C. Krintz, P. F. Sweeney and V. T. Rajan, "Online phase detection algorithms," *International Symposium on Code Generation and Optimization (CGO'06)*, New York, NY, USA, 2006, pp. 13 pp.-123.

[18] O. Navarro, M. Huebner, (2018) Runtime Adaptive Cache for the LEON3 Processor. In: Voros N., Huebner M., Keramidas G., Goehringer D., Antonopoulos C., Diniz P. (eds) Applied Reconfigurable Computing. Architectures, Tools, and Applications. ARC 2018. Lecture Notes in Computer Science, vol 10824. Springer, Cham

[19] O. Navarro, T. Leiding and M. Hübner, "Configurable cache tuning with a victim cache," 2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), Bremen, 2015, pp. 1-6.

[20] B. Reagen *et al*., "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, 2016, pp. 267-278.

[21] G. Reinman, and N.P. Jouppi, COMPAQ Western Research Lab: CACTI2.0: An Integrated Cache Timing and Power Model, 1999.

[22] A. Sembrant, D. Eklov and E. Hagersten, "Efficient software-based online phase classification," *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, 2011, pp. 104-115.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. *In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, October 2002. San Jose, California.

[24] T. Sherwood, S. Sair and B. Calder, "Phase tracking and prediction," *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, San Diego, CA, USA, 2003, pp. 336-347.

[25] F. Song, Z. Guo and D. Mei, "Feature Selection Using Principal Component Analysis," *2010 International Conference on System Science, Engineering Design and Manufacturing Informatization*, Yichang, 2010, pp. 27-30.

[26] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena and D. Chiou, "GPGPU performance and power estimation using machine learning," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, CA, 2015, pp. 564-576.

[27] C. Zhang, F. Vahid and W. Najjar, "A highly configurable cache architecture for embedded systems," *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, San Diego, CA, USA, 2003, pp. 136-146.