Predicting Thread Profiles across Core Types via Machine Learning on Heterogeneous Multiprocessors

Conference Paper · November 2016

DOI: 10.1109/SBESC.2016.017

Vinicius Petrucci
Micron Technology, Inc
54 PUBLICATIONS **919** CITATIONS

Predicting Thread Profiles across Core Types via Machine Learning on Heterogeneous Multiprocessors

Cha V. Li
Department of Computer Science
University of Pittsburgh, USA

Vinicius Petrucci
Department of Computer Science
Federal University of Bahia, Brazil

Daniel Mossé
Department of Computer Science
University of Pittsburgh, USA

***Abstract*—Given that energy consumption has become one of the most important issues in computer systems, Heterogeneous Multiprocessors (HMPs) have been introduced, where *large* high-performing and *small* power-efficient cores can co-exist on the same platform and share the processing of the workload. Clearly, the concept is the same whether it is multiple processors on a board or a chip multiprocessor with several cores on a chip. With the advent of HMPs, thread scheduling becomes much more challenging, while having to deal with thread to processor-type mapping. In particular, it is important that the operating system is able to understand the workload behavior when a thread is to be migrated to a core of a different type. In this paper, we describe a thread characterization method that explores machine learning techniques to automate and improve the accuracy of predicting thread execution across different processor types. We use hardware performance counters and use machine learning to predict performance when moving a thread to another core type on heterogeneous processors. We show that our characterization scheme achieves higher structural similarity (SSIM) values when predicting performance indicators, such as instructions per cycle and last-level cache misses, commonly used to determine the mapping of threads to processor types at runtime. We also show that support vector regression achieves higher SSIM values when compared to linear regression, and has very low (1%) overhead.**

I. Introduction

Heterogeneous Multiprocessors (HMP) with *large* and *small* cores are becoming more prevalent due to their energy efficiency benefits. The large cores have higher clock frequencies, complex instruction sets and large on-chip caches, whereas small cores have reduced clock frequencies, possibly simpler instruction sets, and reduced on-chip caches. Clearly, the concept is the same whether it is multiple processors on a board or a chip multiprocessor with several cores on a chip; in this paper we refer to "processors" or "cores" interchangeably. Chip manufacturers, including ARM, Intel, and AMD, are moving towards HMPs in the latest generation of mobile devices, where power constraints set by batteries is among the leading motivation [1], [2], [3], [4], [5]. As "Software as a Service" continues to grow in popularity, cloud computing will also profit from HMPs, since thousands of servers operate continuously but with variable load and thus power management is critical in reducing energy costs [6], [7].

To take advantage of HMPs for power-management, an intelligent thread scheduling approach can be broken down into three general stages (Figure 1). The first stage focuses on thread characterization given a set of workload attributes, the second stage is thread modeling/prediction, which applies the characterization to predict the future behavior of the thread. Finally, thread allocation scheduling uses the prediction models to assign threads to appropriate processors.

The goal of machine learning is to take a collection of observations (independent variables) and draw a reasonable outcome (dependent variables) from those observations. Supervised learning refers to situations where we know the outcomes for a collection of observations *ahead of time*. For predicting thread phases, we are interested in reliably guessing the future behavior of a thread based on a set of observations we currently know. By knowing the outcomes for each observation through initial data collection, we can turn this task into a *supervised learning problem*.

The benefit of knowing the outcomes *a priori* is that they can be used to determine the *error* of a learning model and help to improve it. Independent variables are called *features* or *attributes* and serve as inputs to learning models; dependent variables, called *true values* or *labels*, are what learning models try to predict. Feature vectors are a set of values attached to each of the features and a *sample* is a pair consisting of a feature vector and its label. Therefore, the problem is to discover a relationship that best maps feature vectors to labels given a set of samples. In our case, we want to discover a relationship between observed counter values at the source processor and the counter we want to predict at the target processor.

A typical thread execution goes through multiple phases. For example, a thread can initially be memory-bound as it loads data from memory to caches and then can become CPU bound as it executes the main part of its task. The ability to identify, predict, and adapt to thread phases plays an important role in thread scheduling and power management, and has been used in many works dealing with HMPs [8], [9], [10], [11], [12].

Several approaches have been proposed to address HMP scheduling [7], [8], [9] and the techniques used have included concepts from machine learning such as Markov models [13],

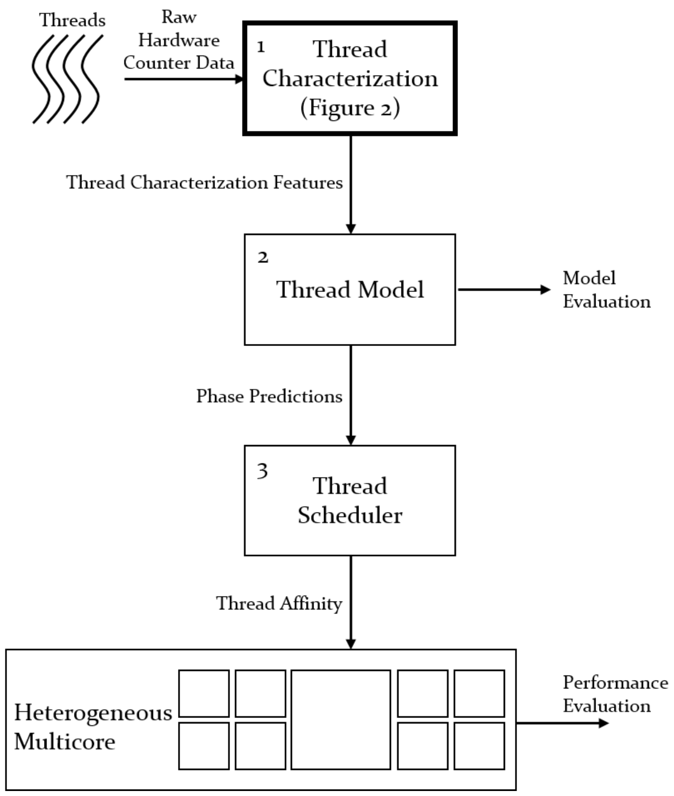

Fig. 1: The 3 stages of an “intelligent thread scheduler” for HMPs. Our work focuses primarily on thread characterization (stage 1) and the use of stage 2 for evaluation. Previous works have focused heavily on stage 2 and 3 as well as evaluating system performance. Stage 1 is further expanded in Figure 2.

[14], regression models [6], [15], [16], reinforcement learning [17], [18], and linear programming [9], [12].

Our contributions stem mainly from two observations of shortcomings in previous work. First, most previous work does it intuitively and manually, using non-statistical approaches to select hardware counters, requiring prior understanding of the available counters. Second, previous works use the counter values directly and do not consider performing any kind of transformation on them to extract even more information. Such a step is critical in processor platforms where only a few hardware counters can be measured simultaneously; in fact, even modern CPUs have a very limited number of hardware registers used for collecting those performance counters.

Our work addresses the above issues by using a statistical approach and combine it with a *characterization method*, transforming all available performance counter data into a more expressive and representative dataset. This transformed data contains snapshots that essentially summarize an arbitrarily long history (e.g., the average of the last n samples of a counter). We then fit a regression model to the transformed (representative subset) data.

In this work we are the first to propose a thread characterization method via machine learning that can predict the values of performance indicators/counters for the next time unit of a given thread execution in any processor in the HMP. We evaluate our characterization method using regression models for predicting thread phases across different processor types. We show that our method improves structural similarity (SSIM) values for both statistically chosen hardware counters and hand-picked counters when compared to using the chosen counters directly. Our results indicate that support vector regression (SVR) model can provide additional improvements to SSIM requiring only a small amount of added execution time when compared with linear regression.

The paper is structured as follows: Section II presents our thread characterization approach, Section III presents the machine learning models and experimental setup used to evaluate our approach, Section IV presents our results. Section V concludes the paper.

II. Thread Characterization Method

In this section we describe how performance datasets are transformed from raw hardware performance counter data to more representative characterization data (depicted on Stage 1 of Figure 1). This new data representation will be used as input to machine learning models in our thread characterization method (shown in Figure 2).

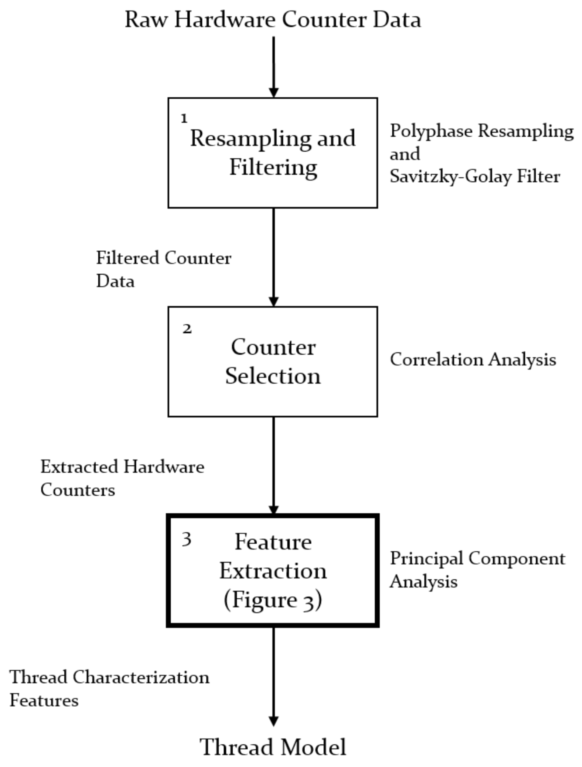
Raw Hardware Counter Data
1 Resampling and Filtering
Polyphase Resampling and Savitzky-Golay Filter
Filtered Counter Data
2 Counter Selection
Correlation Analysis
Extracted Hardware Counters
3 Feature Extraction (Figure 3)
Principal Component Analysis
Thread Characterization Features
Thread Model

Fig. 2: High-level summary of our “characterization method” representing the 3 sub-stages of stage 1 in Figure 1. Our method takes raw hardware counter data and transforms it into data that can improve the accuracy of our models. The third stage of this process is further expanded in Figure 3.

There are several reasons to preprocess the raw performance dataset. First, if samples are gathered periodically, which they typically are, the number of samples gathered from the a large processor (e.g., Intel i7) and small processor (e.g., Intel Atom) differ greatly due to speed/frequency differences. This means timestep t on a small processor may not correspond to timestep t on a large processor. Second, the amount of variance (noise) in the collected data is very high due to CPU contention, measurement errors, and other similar interference. Third, the datasets contain measurements of roughly 60 counters but the number of counters that can be tracked simultaneously at runtime is limited both by the software and hardware, and thus we must select a small set of predictive counters. Finally, raw datasets represent instantaneous counter values as opposed to counter *trends* that are more valuable for estimating future values.

Simply put, our method effectively estimates the value of a single performance counter on the *target processor* at timestep $t+1$ given the behavior of a key subset of performance counters in the past t timesteps on the *source processor*. Note that the target and source cores can be the same. Below we explain the three phases shown in Figure 2.

A. Resampling and Filtering

Datasets, represented by $\mathcal{D}$, are matrices with M rows x $N+1$ columns, where D_{ij} is the i^{th} observation of feature j or the performance counter j measured at timestep i. Features are denoted as $\mathbf{f}_{A,n}$ where A represents a specific dataset, and n indicates the n^{th} feature ($1 \leq n \leq N$). A fully specified feature vector is represented as $\mathbf{x}_A^{(m)}$, where m indicates the sample number, $1 \leq m \leq M$, and $A \in \{S, T\}$ indicates whether the dataset is from the source or target datasets, respectively. A label vector is denoted by $\mathbf{y}$ and is the last column, $(N+1)$, of a dataset. A specific value from a feature vector or label vector is written as $x_{A,n}^{(m)}$ and $y^{(m)}$, respectively. $\mathbf{X}$ is a feature matrix such that $\mathbf{x} \in \mathbf{X}$ and $\mathbf{X} \subset \mathcal{D}$. Using this notation, a sample is written as $\mathbf{z} = (\mathbf{x}^{(m)}, y^{(m)})$ and a whole dataset can be written as $\mathcal{D} = (\mathbf{X}, \mathbf{y})$.

Given that computing resources in the small or large processors have different processing capabilities, if data is collected at a fixed period, sample numbers from different processors will correspond to different parts of the application being measured. For example, the m^{th} sample taken from a benchmark running on an Atom processor and the m^{th} sample taken from the same benchmark running on an i7 processor do not represent the same point in the benchmark's execution, given the vastly different architectural differences (e.g., clock speed and cache sizes).

To address this issue, one of the datasets must be "stretched" (upscaled) or "compressed" (downscaled), such that the m^{th} sample of each dataset corresponds roughly to the same point in the benchmarks execution. We used a polyphase resampling algorithm [19] to accomplish this. More specifically, the behavior of a performance counter over a given time interval, $\mathbf{f}$, is interpolated by a factor of $\frac{U}{L}$ where U is the length of source dataset (upscale factor), L is the length of target dataset (downscale factor).

We also filter the datasets to reduce the variance (noise) of each counter, while preserving the phases as much as possible. We experimented with moving, weighted, and jumping averages, but the averaging algorithms tend to "flatten" peaks and spikes. Therefore we decided to use a Savitzky-Golay (SG) filter [20] because SG filters maintain features such as local maxima and local minima.

Once all the datasets are filtered, they are combined such that the target counter is the last column of the new dataset:

$$\mathcal{D}_{train} = \begin{bmatrix} x_{S,1}^{(1)} & x_{S,2}^{(1)} & \cdots & x_{S,N}^{(1)} \\ x_{S,1}^{(2)} & x_{S,2}^{(2)} & \cdots & x_{S,N}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{S,1}^{(M)} & x_{S,2}^{(M)} & \cdots & x_{S,N}^{(M)} \end{bmatrix} \cup \begin{bmatrix} x_{T,j}^{(1)} \\ x_{T,j}^{(2)} \\ \vdots \\ x_{T,j}^{(M)} \end{bmatrix}$$

where $\mathcal{D}_{train}$ is for the training set (ahead of execution), $x_{T,j}$ represents the j^{th} counter value (the label) on the target processor type that the models will try to predict given data from the source processor type (the features $x_{S,i}, 1 \leq i \leq N$). $\mathcal{D}_{test}$ is constructed similarly, and used at run time.

B. Performance Counter Selection

The number of performance counters that can be measured simultaneously is typically limited to a small number, due to hardware and software constraints. Thus, our goal is to discover the "best" subset of counters (e.g., 4 counters as provided by Intel platform [1]) from the source dataset of 50+ counters. Intuitively, we want counters that capture the information in the source dataset and counters that appear to correlate with the target counter. Our approach to performing this task uses correlation analysis and univariate analysis to extract source counters that are *representative* and *predictive*.

Selecting *representative counters* involves calculating the covariance matrix of the source dataset and "grouping" sets of counters that are highly correlated with each other. As in prior works [16], [21], we use Spearman's ranking to determine how well two variables (two source counters in our case $x_{S,i}$ and $x_{S,j}$) are related. Our goal is to select a set of counters that *represent the source dataset* and that can be chosen to be independent of the target counter.

We consider that two performance counters, $x_{S,i}$ and $x_{S,j}$, are *correlated* if $\rho_{i,j} > .65$. A ρ of .65 may seem low but this is done to improve the *generalizability* property of the selected counters for a given set of applications and workloads. If we set a ρ threshold that is too high we risk selecting counters that are only highly correlated *in the given data*. Setting it too low will result in too many chosen counters. Therefore, we set the threshold just above 50%. Our assumption is that two counters that are slightly correlated in our training dataset will *also be* at least slightly correlated in unseen data (test data). In our case, we define a counter as *representative* if the counter covers at least 65% of the original counters. "Covers" in this context means that the other counters do not need to be considered, if the coverage is high enough (the very definition of "representative" counters). The covering threshold is also 65% for similar reasons. Representative counters should, intuitively, cover at least 50% of all counters, however, setting a threshold too high will result in too few counters being chosen.

Selecting *predictive counters* is a process where we measure the correlation of each source counter with the target counter. Representative counters are prioritized over predictive counters to improve the generalizability of our models, therefore, predictive counters are only used when fewer than four representative counters are extracted. In case there are more than four representative counters selected, the representative counters are truncated and predictive counters ignored. For example, after the selection process, considering a platform with only four hardware counters available, we determine $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ having the following structure:

$$\begin{bmatrix} x_{S,1}^{(1)} & x_{S,2}^{(1)} & x_{S,3}^{(1)} & x_{S,4}^{(1)} & x_{T,j}^{(1)} \\ x_{S,1}^{(2)} & x_{S,2}^{(2)} & x_{S,3}^{(2)} & x_{S,4}^{(2)} & x_{T,j}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{S,1}^{(M)} & x_{S,2}^{(M)} & x_{S,3}^{(M)} & x_{S,4}^{(M)} & x_{T,j}^{(M)} \end{bmatrix}$$

C. Feature Extraction and Reduction

Since we are more interested in the performance counter trends than instantaneous counter values, we first have to define both a set of trends we want to pursue and a history length, H, to extract those trends from. In our problem, we are interested in the mean μ, minimum min, and maximum max values of the past H consecutive samples of each selected counter. The trends we are interested in are called *extracted features*. Intuitively, this strategy is similar to storing numerous history table summaries.

Figure 3 provides a summary of the extraction and reduction process. To extract performance counters our process is to put all performance counters through two steps of processing: feature extraction and feature reduction (through principal component analysis). This figure represents the detailed view of stage 3 from Figure 2.

In performing feature extraction, we will have the following transformation considering, for instance, a subset of four performance counters:

$$\begin{bmatrix} x_{S,1}^{(t-H)} & x_{S,2}^{(t-H)} & x_{S,3}^{(t-H)} & x_{S,4}^{(t-H)} \\ \vdots & \vdots & \vdots & \vdots \\ x_{S,1}^{(t-1)} & x_{S,2}^{(t-1)} & x_{S,3}^{(t-1)} & x_{S,4}^{(t-1)} \\ x_{S,1}^{(t)} & x_{S,2}^{(t)} & x_{S,3}^{(t)} & x_{S,4}^{(t)} \end{bmatrix}$$

$$\downarrow$$

$$\tilde{\mathbf{x}}^{(t)} = (\mu_{S,1}, \dots, \mu_{S,4}, min_{S,1}, \dots, min_{S,4}, max_{S,1}, \dots, max_{S,4})$$

$$\tilde{y}^{(t)} = x_{T,j}^{t+1}$$

$$H < t < M$$

Therefore, a sample in our extracted features datasets ($\tilde{\mathcal{D}}_{train}$ and $\tilde{\mathcal{D}}_{test}$) is defined as:

$$\tilde{\mathbf{z}} = (\tilde{\mathbf{x}}^{(t)}, \tilde{y}^{(t)})$$

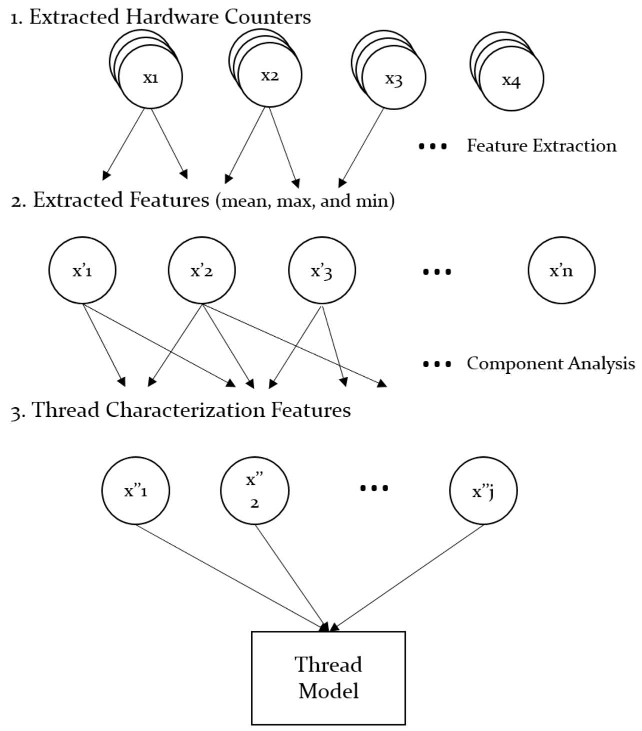
1. Extracted Hardware Counters
x1
x2
x3
x4
Feature Extraction
2. Extracted Features (mean, max, and min)
x'1
x'2
x'3
x'n
Component Analysis
3. Thread Characterization Features
x''1
x''2
x''j
Thread Model

Fig. 3: Process of feature extraction and reduction.

Using the above formulation and procedure, we can extract an arbitrary number of features describing the raw data.

The final stage of our thread characterization method performs principal component analysis (PCA) to reduce the number of extracted features through a linear combination of the features; this is especially important for regression if many extracted features are desired [22]. Since the number of directly measurable counters is limited, it is critical to extract as much information as possible from these performance counters and then reduce the extracted information into a simpler representation.

III. Evaluation Methodology

In this section we describe the experimental platform and the machine learning models used in our method evaluation, as well as the metrics we report.

A. Experimental Setup

We performed performance data collection on two separate heterogeneous machines: an *Intel Atom (small)* CPU (1.66Ghz, 1MB L2) machine with 2GB of RAM and an *Intel i7 (large)* CPU (3.8Ghz, 8MB L3) machine with 16GB of RAM. All of our experiments and data-handling were implemented using the Octave (version 3.6.3) environment [23] with *libsvm* (version 3.14) [24], a library that implements variations of support vector machines including the regression model.

We used a set of SPEC benchmarks shown in Table I (left) along with how many samples, taken at 1 second intervals, each program contributed to the overall data. Each of the

TABLE I: Number of performance counter measurements (left) contributed by each benchmark for each processor type. Data split (right) used for evaluating our characterization technique.

Benchmark	Measurements	
	Intel Atom	Intel i7
astar	1,259	163
sjeng	4,223	495
bwaves	7,419	596
calculix	11,820	795
milc	2,859	391
namd	4,580	431
soplex	1,058	102
Total	*33,219*	*2,973*

Dataset	Benchmark
training	bwaves
	calculix
	milc
	namd
testing	astar
	sjeng
	soplex

benchmarks was executed on both CPUs to collect the performance counter data for evaluation. All together, the Atom dataset is composed of 59 counters and 33,219 samples and the i7 dataset is composed of 56 counters and 2,973 samples. The approximately 10x difference in counter measurements is due to the frequency differences between the Atom and i7 and other architectural features (e.g., a faster CPU finishes the benchmarks faster). The samples were collected using the `perf` program built into Linux 2.6+ systems [25]. Note that the benchmarks were divided into training and testing sets as shown in Table I (right).

B. Regression Models

We instantiate our thread prediction method using two regression models for phase prediction, namely linear regression (LR) and support vector regression (SVR). Each of these models is simply "plugged" into the "Thread Model" stage previously shown in Figure 1.

Linear regression (LR) is a popular method due to its simplicity and relatively good performance [6], [15], [16]. Support Vector Regression (SVR) is a modification of Support Vector Machines designed for regression and has been used for predicting the performance measures of a distributed shared memory multiprocessor [26]. Intuitively, the SVR model is searching for the best-fit regression line based on data points that are sufficiently *far* from the line (given by a threshold parameter). This differs from LR because LR considers all data points.

C. Structural Similarity Metric

We use Structural Similarity (SSIM) to represent phase preservation (i.e, how to predict the phases in one type of processor from phase data in another type of processor). SSIM was first used to measure the similarity between images [27], but it is not specific to images, and therefore we adopt it for measuring phase preservation. Although error rates (absolute or MSE) are a common performance metric, they do not imply a high SSIM value. On the other hand, a high SSIM value does not imply a low error, however, it does imply phase preservation. Assuming a high SSIM, a large error is typically a vertical shift that can be reduced through normalization while preserving phase.

TABLE II: The representative performance counters were automatically chosen by our extraction technique for each source processor. Extraction was performed using the training data.

Source processor Type	Representative Counters
Intel Atom	`BUS-TRANS-BURST`
	`BRANCH-INSTRUCTIONS-RETIRED`
Intel i7	`BPU-CLEARS`
	`L1D-CACHE-LOCK-FB-HIT`
	`LAST-LEVEL-CACHE-MISSES`

Similar to correlations, SSIM values range from 1 (perfect structural preservation) to -1 (perfect inverse structural preservation). In our results, we present the absolute values of the achieved SSIM values, that is, no negative numbers. A high (close to 1) absolute value indicates that the similarity is either directly or inversely proportional to the actual execution behavior.

IV. Results

We performed experiments to evaluate our approach using two sets of hardware performance counters: `extracted` and `picked`. `Extracted` includes the representative and predictive counters chosen using the analysis in Section II and `picked` contains the hard-coded counters for IPC and LLCM; these specific counters were selected because they have been used in prior work to guide thread scheduling decisions [8], [9], [10]. For `extracted`, Table II lists the counters chosen to represent the source processor type. Predictive counters are not presented since they depend on the chosen target counter.

We compare the results of our characterization approach to those of using the raw values of `extracted` and `picked` when used to predict IPC and LLCM across processor types. The extracted features we chose to use were the mean, the maximum, and the minimum values based on history length $H = 5$ (cf. Section II-C).

We use our approach to perform phase prediction using different machine learning models on the test data described in Table I (right). The models were judged on two metrics: phase preservation and prediction time, both of which were measured using 5-fold cross-validation. As usual, the use of cross-validation with data-splitting was to minimize the effects of *sample ordering* on training performance.

A. Prediction Accuracy

The SSIM values presented in this section are the best values achieved by each of the models. In the case of SVR, the parameters other than the weights were fitted by permuting their values. LR has no additional parameters so no additional fitting was required. We use a unique model for each combination of source and target counters, as opposed to one global model that may not model the finer characteristics of each target counter.

Tables III and IV present the overall SSIM results (i.e., the average over many different runs and parameters) for each combination of target and source counters for each model. In each table, `extracted` and `picked` are further broken

TABLE III: SSIM Results Predicting Atom to i7

Linear Regression (LR)

	`extracted` Source Counters		`picked` Source Counters	
Target Counter	Direct	Preprocessed	Direct	Preprocessed
IPC	0.6644	**0.7516**	0.7524	0.7438
LLCM	0.5074	**0.6603**	0.7646	**0.7905**

Support Vector Regression (SVR)

	`extracted` Source Counters		`picked` Source Counters	
Target Counter	Direct	Preprocessed	Direct	Preprocessed
IPC	0.6550	**0.8240**	0.7961	**0.9638**
LLCM	0.5663	**0.8526**	0.7962	**0.8185**

TABLE IV: SSIM Results Predicting i7 to Atom

Linear Regression (LR)

	`extracted` Source Counters		`picked` Source Counters	
Target Counter	Direct	Preprocessed	Direct	Preprocessed
IPC	0.4062	0.4055	0.5201	**0.5458**
LLCM	**0.2711**	0.1157	**0.7836**	0.7672

Support Vector Regression (SVR)

	`extracted` Source Counters		`picked` Source Counters	
Target Counter	Direct	Preprocessed	Direct	Preprocessed
IPC	0.5697	0.5675	0.5867	**0.8534**
LLCM	0.3771	**0.6301**	0.8533	0.8618

down into *direct* and *preprocessed*, the first represents using the values of the counters directly and the second represents using the characterization produced by our method (based on input counters).

We make a couple of interesting observations about the collected data. First, in the vast majority of scenarios our characterization method achieves higher SSIM values when predicting the given target counter. This is because the cases where our characterization does not perform better than direct counter use, it only performs about 1% worse. There is only one example where our characterization performs significantly worse than direct counter use and that is in the LR case, under the `extracted` LLCM case (see Table IV). However, this outlier is unimportant due to the next observation.

The second observation is that SVR achieves higher SSIM values than LR *in all scenarios by a non-trivial amount*. Examining the Atom-to-i7 data (Table III), we can see that preprocessed SVR can achieve up to a 30% improvement in SSIM over preprocessed LR. The smallest improvement by SVR is found in the `picked` preprocessed LLCM case where only a 2% improvement is found. The average improvement is 18%. The i7-to-Atom preprocessed SVR results in Table IV show that the greatest improvement for SVR is more than $2\times$ in the `extracted` LLCM case, followed by a 56% improvement in the `picked` IPC case. Looking at all SVR SSIM values, it is clear that SVR offers an advantage over LR.

Although `picked` outperforms `extracted` in all cases, clearly it is not a generic method; in addition, we make the following observations. First, we chose to predict IPC and LLCM, so it should not be a surprise that choosing IPC and LLCM as the source counters lead to good results. Second, in the Atom-to-i7 direction `extracted` performs comparably well with `picked` after characterizing the counters. This is interesting in that it suggests that there exist scenarios where automatically and statistically choosing counters to predict phases in HMPs is better than intuitively selecting counters. This is reassuring for cases where no intuitive counters are known.

B. Prediction Overhead

Another metric of importance is the overhead of the schemes used for guiding thread decisions; if the time and energy it takes to carry out prediction of thread performance outweighs the benefits in terms of performance or energy consumption, it would be a waste to carry out such predictions and "smart" allocation. Therefore, we measured the time it took to carry out the prediction and report it below.

We consider CPU time in microseconds for carrying out the prediction on the source core type. Given that the initial motivation is to reduce power consumption, without increasing the time of execution, the thread in charge of predicting the phase of other threads should not itself be a significant source of CPU usage. More specifically, *prediction time* is the amount of CPU time needed for a model to make a single prediction given some counter data. We calculated this value by dividing the total CPU time required by a model to make all predictions by the number of predictions made.

Table V summarizes the execution time required by each model for predictions in both core types. The bulk of our characterization method happens when training the model, which is done once and offline. Therefore, the time required for preprocessing is left out of the results.

TABLE V: Average prediction overhead (μs) for each model for each set of performance counters.

Atom to i7		
Model	Avg Exec. Time (μs)	
	extracted	picked
LR	12.05	11.94
SVR	75.38	48.97

i7 to Atom		
Model	Avg Exec. Time (μs)	
	extracted	picked
LR	12.24	13.79
SVR	62.44	52.61

Although SVR is about $4-6\times$ slower than LR, in reality this translates into only approximately $40-60\mu s$ difference in absolute time. This small increase in the execution time of prediction is justified when considering the magnitude of SSIM improvements and the number of cases that were improved. Note that this prediction is only done at context switch times (from the OS perspective). The CPU time required for prediction in the worst case is less than $100\mu s$. Considering a typical scheduling period of $10ms$, the overhead is 1% (0.1 ms / 10 ms).

C. Discussion

The results presented in this paper can provide a foundation for advanced investigations and interesting new directions. Our proposed approach can be applied to other domains such as memory management, embedded systems, and cloud computing, beyond intelligent thread scheduling for power management on HMPs. Given specific computing environments, the use of additional context-dependent features may improve the performance of our machine learning models.

Although we explored a few options for extracted features (mean, max, and min), there is potential for discovering new extracted features based on performance counter behavior. Also, the machine learning models used in our evaluation use MSE (Mean Square Error is the typical loss function) as their loss function, however, it may be possible to adopt SSIM as the loss function to perform the optimization.

V. Conclusion

We described a method for thread characterization on heterogeneous multiprocessors that uses techniques from machine learning to extract additional information from hardware performance counters beyond that of just raw counter values. We then use this new information to train regression models for the task of thread phase prediction with respect to IPC and LLCM measures. We also introduce the use of a new metric, structural similarity (SSIM), for judging the performance of phase prediction methods. Our results show that our characterization method improves SSIM values for both statistically-chosen hardware counters and hand-picked counters when compared to using the chosen counters directly. We also show that the support vector regression (SVR) model provides additional improvements to SSIM at only a small cost in additional prediction time.

References

[1] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer, "Quickia: Exploring heterogeneous architectures on real prototypes," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, 2012.

[2] ARM. (2011) big.LITTLE processing. [Online]. Available: http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php

[3] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: a platform for os-level power management," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 289–302. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519097

[4] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," *Proceedings of the 19th High Performance Computer Architecture*, 2013.

[5] Y. Zhu, M. Halpern, and V. J. Reddi, "The role of the cpu in energy-efficient mobile web browsing," *Micro, IEEE*, vol. 35, no. 1, pp. 26–33, 2015.

[6] J. L. Berral, R. Gavalda, and J. Torres, "Adaptive scheduling on power-aware managed data-centers using machine learning," in *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, ser. GRID '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 66–73. [Online]. Available: http://dx.doi.org/10.1109/Grid.2011.18

[7] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang, "Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 246–258.

[8] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *EuroSys'10*.

[9] V. Petrucci, O. Loques, D. Mosse', R. Melhem, N. Gazala, and S. Gobriel, "Thread assignment optimization with real-time performance and memory bandwidth guarantees for energy-efficient heterogeneous multi-core systems," in *RTAS'2012*, Beijing, China, 2012.

[10] V. Petrucci, O. Loques, and D. Mossé, "Lucky scheduling for energy-efficient heterogeneous multi-core systems," in *HotPower'12*.

[11] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto, "Leveraging Core Specialization via OS Scheduling to Improve Performance on Asymmetric Multicore Systems," *ACM Trans. Comput. Syst.*, vol. 30, no. 2, pp. 6:1–6:38, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2166879.2166880

[12] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, and S. Gobriel, "Energy-efficient thread assignment optimization for heterogeneous multicore systems," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, pp. 15:1–15:26, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2566618

[13] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th annual international symposium on Computer architecture*, ser. ISCA '03. New York, NY, USA: ACM, 2003, pp. 336–349. [Online]. Available: http://doi.acm.org/10.1145/859618.859657

[14] R. Sarikaya, C. Isci, and A. Buyuktosunoglu, "Program behavior prediction using a statistical metric model," in *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '10. New York, NY, USA: ACM, 2010, pp. 371–372. [Online]. Available: http://doi.acm.org/10.1145/1811039.1811092

[15] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 46–55, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1577129.1577137

[16] M. Y. Lim, A. Porterfield, and R. Fowler, "Softpower: fine-grain power estimations using performance counters," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 308–311. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851517

[17] H. Jung and M. Pedram, "Supervised learning based power management for multicore processors," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 29, no. 9, pp. 1395–1408, Sep. 2010.

[18] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 39–50. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.21

[19] R. W. S. Alan V. Oppenheim, *Discrete-Time Signal Processing*. Prentice Hall, 2009.

[20] A. Savitzky and M. J. E. Golay, "Smoothing and differentiation of data by simplified least squares procedures." *Analytical Chemistry*, vol. 36, no. 8, pp. 1627–1639, 1964.

[21] J. L. Myers and A. D. Well, *Research design and statistical analysis*. Lawrence Erlbaum, 2002.

[22] C. M. Bishop, *Pattern Recoginition and Machine Learning*. Springer, 2007.

[23] Octave-forge - extra packages for gnu octave. [Online]. Available: http://octave.sourceforge.net/

[24] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[25] perfmon2: the hardware-based performance monitoring interface for linux. [Online]. Available: http://perfmon2.sourceforge.net/

[26] M. F. Akay and I. Abaskele, "Predicting the performance measures of an optical distributed shared memory multiprocessor by using support vector regression," *Expert Systems with Applications*, vol. 37, no. 9, pp. 6293 – 6301, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417410001314

[27] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *Image Processing, IEEE Transactions on*, vol. 13, no. 4, pp. 600 –612, april 2004.