Workload Prediction for Adaptive Power Scaling Using Deep Learning

Stephen J. Tarsa, Amit P. Kumar, and H.T. Kung
Intel Corporation, Microarchitectures Research Lab, Santa Clara, CA
Harvard University, Cambridge, MA

Abstract—We apply hierarchical sparse coding, a form of deep learning, to model user-driven workloads based on on-chip hardware performance counters. We then predict periods of low instruction throughput, during which frequency and voltage can be scaled to reclaim power. Using a multi-layer coding structure, our method progressively codes counter values in terms of a few prominent features learned from data, and passes them to a Support Vector Machine (SVM) classifier where they act as signatures for predicting future workload states. We show that prediction accuracy and look-ahead range improve significantly over linear regression modeling, giving more time to adjust power management settings. Our method relies on learning and feature extraction algorithms that can discover and exploit hidden statistical invariances specific to workloads. We argue that, in addition to achieving superior prediction performance, our method is fast enough for practical use. To our knowledge, we are the first to use deep learning at the instruction level for workload prediction and on-chip power adaptation.

I. INTRODUCTION

Mechanisms like dynamic voltage and frequency scaling (DVFS) enable adaptive power management, and promise to improve operating efficiency by tailoring device parameters to workloads at runtime. Such adaptation requires anticipating future circuit states, and online workload modeling is one predictive approach that minimizes static or steady-state assumptions about workloads. This flexibility is important since performance characteristics like power consumption vary widely by user and application mix [1] [2].

In this paper, we use statistical relationships learned from hardware performance counters to predict periods of low instruction throughput, during which voltage and frequency can be scaled to reclaim power. We target predictions that are long-range and low-latency, meaning that look-ahead time is maximized to allow for chip adjustment, and predictive models update quickly when workloads change.

The most popular method for workload prediction is regression [3] [4] [5], which fits a polynomial to counter measurements and extrapolates future states. However, we show that regression accuracy degrades at long ranges, as future states are unlikely to be a simple extrapolation of prior measurements. Moreover, to capture fine-grained behaviors, regression coefficients must continually be updated even if the high-level workload is the same.

Instead, we implement modeling and prediction using hierarchical sparse coding and Support Vector Machine (SVM) classification, as depicted in Figure 1. This approach first codes counter measurements in terms of a few atoms selected from a dictionary of patterns, or features, learned from training data. These feature vectors are then concatenated and coded again to capture feature interrelationships over a larger spatial scale. An SVM classifies the resulting sparse vectors with a common label when they precede an event of interest.

The process of sparse coding cuts away noise from measurement data, and improves SVM classification accuracy when data is subject to non-Gaussian variations. In addition, hierarchical models can capture statistical patterns embedded in large state spaces from a modest number of training examples. We show that training time can be reduced even more by bootstrapping feature dictionaries using a canonical set of Layer 1 features, which are common across workloads. As a result, when workloads change, only a small number of training samples are required to update our predictor.

We adopt hierarchical sparse coding to capture complicated signature patterns appearing over time, and show that this improves prediction range over regression and heuristic techniques. This is because data vectors that have been hierarchically sparse coded are classified in a transformed domain: the feature space. By performing statistical inference on feature vectors, we exploit workload-specific invariant patterns that are typically “hidden,” or not immediately observable in the raw data domain. In this paper, we use clustering on training data to extract this hidden structure. This form of deep learning has yielded major application gains in fields like computer vision, speech recognition, and machine translation (e.g. [6] [7]). To our knowledge, we are the first to apply sparse hierarchical models to chip performance data for adaptive optimization.

II. TARGET SCENARIO AND DATA COLLECTION

A. Device, Workloads, and Event Prediction

We collect performance counter data using gem5, an architecture-level simulator with full system support, including frame buffer rendering and an interactive shell [8]. Snapshots of counter values are taken every 500µs from a simulated 1.0 GHz ARM v7a chip.

![Figure 1](image-url)
Our primary workload is the BBENCH benchmark [9], running atop Android Gingerbread. Figure 2 plots committed instructions/cycle for three phases of activity: OS boot, web browser startup, and a web surfing phase. During surfing, web sites are loaded from off-chip using Android’s built-in browser, and Javascript code simulates user link clicks. We see that instruction throughput drops below 25% during nearly 20% of the surfing phase, making these intervals a good target for DVFS. Here, low instruction throughput arises under two sets of circumstances: first, while waiting on web page I/O, and second, during computation-dominated intervals due to the architecture-level interactions of chip components. This latter class of intermittent dips make up 7.3% of the overall workload, and are difficult to predict with heuristics. By applying hierarchical sparse coding, we will find predictive signatures for these dips, and reclaim power by voltage/frequency scaling.

To generalize gains from our approach, we also report prediction performance for the ASIMBENCH/Moby benchmark suite [10]. This set of workloads includes examples of a game, audio and video playback, and document manipulation applications running under Android ICS. Though these workloads capture additional application behaviors, they lack the simulated user interactions of BBENCH that cause pertinent statistical patterns to recur over time. We therefore present these limited results with the caveat that incorporating user interaction is a necessary next-step to assess power savings.

Our workload prediction scenario is related to the well-studied problem of workload phase detection. Phase detection is often motivated by the desire to identify large stable regions of a workload, which ensure that overheads resulting from optimization-driven adjustments can be sufficiently amortized. Detection techniques like working set signatures, basic block vectors, and conditional branch counters (see [11] for a review) apply a threshold to one or more hardware counters to identify deviations relative to a long-term average. We will show in Section IV B, short unstable workload regions have few prediction opportunities, since the number of recurrent patterns is limited; detecting rapid phase changes is one potential way of short-circuiting ineffective predictions within these regions.

B. Performance Counter Configuration

Our sparse-coding predictor will use deep learning to discover signatures that span multiple measurement windows over time, and multiple counters across the chip. Counters capture events like committed instructions, data table hits, misses, flushes, etc. This approach contrasts with standard regression modeling that focuses on one or two counters most correlated with the metric of interest, e.g. in [4].

Typically, the choice of performance counters to include on a chip is based on both the desired performance monitoring data, as well as layout and design constraints. In this section, we use gem5 to also characterize the statistical properties of counter configurations. This allows us to choose a small number of counters that still give good prediction performance.

First, we collect data from 120 simulated hardware counters during the execution of our test workloads. This data represents a superset of possible hardware configurations. Every 500μs, deltas from previous values are recorded, and we use the resulting data vectors to calculate a correlation matrix. We then group together counters whose correlational magnitude exceeds 0.98, since these counters are statistically interchangeable due to their near-total correlation. By choosing one counter from each group, we then have a minimal configuration that provides comprehensive architecture-level statistics.

We identify 34 different groups from the 120 possible counters studied on our ARM v7a-chip. We find, for example, that the number of integer register reads is interchangeable with the number of committed integer operations, though these are collected from different locations on the chip. Such statistics-driven counter selection is useful to control the data-collection overhead of our predictor without sacrificing accuracy.

Principal Component Analysis (PCA), a standard technique for dimensionality reduction, was also applied. Even though PCA found a lower dimensional basis for our data, that representation relies on linear combinations of all 34 counters. Therefore, this analysis does not allow us to reduce the number of counters deployed on-chip, though it implies that compressive techniques such as random linear combination are worth investigating to reduce acquisition costs [12] [13].

III. HIERARCHICAL SPARSE CODING

Sparse coding [14] formalizes dictionary learning and feature extraction by the following minimization:

$$\min \|X - DZ\|_2 \quad \text{s.t. } \|z_i\|_0 \leq k \text{ for } i = 1...t$$

with X an $n \times t$ data matrix containing t snapshots of n counters, D an $n \times m$ dictionary of m features, Z a sparse $m \times t$ matrix of feature coefficients, and k a sparsity constraint. We also put non-negativity constraints on D and Z to improve coding stability for classification under data variation [7].

Sparse coding generalizes k-means, a method for finding clusters in data and representing vectors by their associated cluster centroid [15]. In sparse coding, when $k = 1$, cluster centroids become dictionary atoms, and data-to-cluster assignments correspond to Z’s coefficients. When $k > 1$, a data vector is represented by a sparse, positive, linear combination of k cluster centroids, rather than just one. K-SVD [14] trains D by iteratively fixing Z and using rank-one approximation to update D’s columns, and then fixing D to update Z by
Orthogonal Matching Pursuit (OMP) [16]. After training, OMP computes sparse representations relative to \( D \) for new data vectors.

We choose sparse coding over alternative learning methods such as Convolutional Neural Networks for several reasons: first, a sum of commonly occurring patterns is an appropriate intuitive model for performance counters on a chip with multiple concurrently operating functional circuits. Second, sparse coding yields good classification performance using a linear SVM when few labeled training examples are available [7]. And third, both K-SVD and OMP consist primarily of inner-product computations that can be built in hardware, e.g. [17].

We sparse code data hierarchically, as depicted in Figure 1. At Layer 1, canonical features present in 500\( \mu s \) measurement windows are extracted from raw counter data. At Layer 2, we concatenate sparse feature vectors from multiple measurement windows over time, and again cluster vectors to capture feature co-occurrences. Finally, the outputs of Layer 2 are fed to a linear SVM that implements prediction by assigning a common label to vectors preceding our target event. When detecting one among many events, multiple SVMs or decision trees can be used at this last step. As previously mentioned, we convert each snapshot of counter values into a delta from the previous measurement window. Furthermore, we normalize values to lie between 0 and 1, ensuring that learning using distance minimization treats approximation error in all counters equally.

Sparse hierarchical models are a primary driver of breakthroughs associated with deep learning for three major reasons. First, imposing sparsity on signals is a powerful denoising step that is critical when dealing with natural data variations. Second, by hierarchically learning features and their interrelationships, the model can express feature combinations not directly represented in training data. This means that a few training examples can be generalized for good statistical performance on a larger data set. Third, hierarchical models often include a non-linear pooling step that corrects for alignment variations by maximizing feature response over shifted measurement patches, similar to a convolutional filter. In this paper, we found little gain from pooling since \textit{gem5}'s timing is deterministic and repeatable, however we expect this tool be important when we expand to data measured from hardware.

IV. WORKLOAD PREDICTION

A. BBENCH Performance

We first present results comparing prediction performance between hierarchical sparse coding, linear regression, and static heuristics, for detecting sub-25\% instruction-throughput dips during BBENCH. We define prediction \textit{accuracy} as the portion of all 500\( \mu s \) windows that are correctly labeled, based on whether their instruction throughput is above or below 25\%:

\[
\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total # of Windows Classified}}
\]

We also care about false alarm statistics, especially since there is a recovery cost associated with false positive alarms. We therefore measure \textit{precision} and \textit{recall}. Precision is the number of correctly predicted dips over the total number of alarms:

\[
\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}
\]

Fig. 3. Throughput dips make up nearly 7.3\% of the surfing phase’s computations, and the dotted black line depicts accuracy if we make no positive dip predictions at all. Curves above this line have positive predictive value overall, whereas curves below the dashed line make incorrect predictions to be worse than doing nothing. Precision and recall are shown for sparse coding over \( w=2 \) windows, and characterize the value of positive dip predictions: precision tells how often positive alarms are correct, and recall tells how many dips that occur in the workload are correctly identified. For example, curves below the dashed do-nothing baseline have prec. < 0.50, since positive alarms are mistaken on average.

Recall is the number of correctly predicted dips over the total number of dips that actually occur during the workload:

\[
\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}
\]

Sparse coding dictionaries are trained using the entire web surfing phase of our dataset, and SVM accuracy is measured under cross-fold validation. This captures the best-case scenario, in which data volume is always sufficient for training, regardless of acquisition time. We code over \( w = 1, 2, 4, 8 \) trailing measurement windows of 500\( \mu s \) snapshots to find predictive effects over different time scales. Since parameters such as dictionary size and sparsity constraints may impact performance, we test a wide range of model configurations, and skim off the best performing (we analyze how model configuration affects performance in Section VI).

To compare, we fit regression coefficients for linear predictors of different orders based on the instruction throughput metric directly. In Figure 3, we show data from an order-8 regression, which performed best. Regressions are computed using a sliding window of measurement data, and curves are extrapolated and thresholded to implement dip prediction.

We also compare to a heuristic based on the observation that one dip often precedes another. The heuristic waits until a dip has been observed, and naïvely assumes that another will follow. This represents the simplest predictor.

Figure 3 plots overall prediction accuracy against look-ahead range. First, we see that the static heuristic works only at short look-ahead ranges, when it can pick up the latter portions of dips spanning multiple measurement windows. Regression successfully extends prediction range, and captures dips that can be extrapolated from prior measurements. However, sparse coding consistently has best accuracy and range for look-ahead ranges of 3 windows or more. Furthermore, we see that regression and heuristics make so many mistakes at long ranges that they are worse than doing nothing. In contrast, sparse coding always has positive predictive power, even at a range of 16 windows. Practically, by extending look-ahead range from 6 windows to 16, range is increased by almost 3x.

B. ASIMBENCH/Moby Performance

In this section, we present prediction accuracy for workloads in the ASIMBENCH/Moby benchmark suite, which include a video game, audio and video playback, and viewers.
We proportionally penalize negative scaling due to incorrect scaling decisions. Given these, we define the time scale as a potential limit on prediction opportunities. Here, we break out performance per phase and report only on workload regions with at least $8,000$ measurement snapshots. This conservatively ensures that models can be trained. Benchmark descriptions and overall prediction accuracy are shown in Table I.

Hierarchical sparse coding performs best when workloads have recurring patterns. In this set of benchmarks, media workloads that are cyclic and driven by a regular sampling rate have best prediction performance. For Frozen Bubble Phase 2, MXPlayer Phase 3, and tpod Phase 2, dip prediction is nearly perfect: $94.0\%, 97.8\%$, and $97.6\%$, respectively.

For Frozen Bubble and tpod, an order-8 regression yields sub-$60\%$ accuracy, indicating that dip prediction is not extrapolations of prior counter values sampled every $500\mu$s. Though adapting measurement sampling rate to workloads may improve regression, our method needs no such workload-specific adjustment. Furthermore, adjustment may not be possible in cases like video streaming or games, since frame rates are variable, and affected by background computation and I/O.

We also find that short phases with a high degree of irregular variation lead to few useful long-range signatures. Fitting this description are the k9 Mail benchmark, and most applications’ initialization Phase 1’s. Here, phase detection for stable-region identification could be used to short-circuit predictions that are unlikely to be useful. However, for k9 Mail, we note that the lack of user interaction over a large time scale is a potential limit on prediction opportunities.

V. DYNAMIC POWER FOR VOLTAGE SCALING

The false negative and false positive rates of a predictor impact power savings due to missed opportunity costs and recovery costs for incorrect scaling decisions. Given these, we model realized dynamic power during instruction throughput dips when our predictive frequency/voltage scaling is applied:

$$P_{\text{dyn}} = Pr(\text{True Pos.}) \times (V_{\text{rd}})^2 (f_{\text{rd}}) (a_{rd}) + Pr(\text{False Neg.}) \times (V_o)^2 (f_o) (a_o) + Pr(\text{False Pos.}) \times (V_o)^2 (f_o) (a_o + a_{plt})$$

For simplicity, this model assumes constant capacitance and linear frequency/voltage scaling. As in [18], we proportionally scale power by the amount of realized switching activity in

Despite these considerations, the model consists of terms representing power consumption for correct predictions, false negatives, and false positives, respectively. When a dip is correctly predicted, voltage and frequency are reduced from $V_o = 1.0$ to $V_{\text{rd}} = 0.25$, and $f_o = 1.0$ to $f_{\text{rd}} = 0.25$. For false negatives, voltage and frequency remain at $V_o = 1.0$ and $f_o = 1.0$. For false positives, we penalize incorrect scaling with increased switching activity: $(a_o + a_{plt})$. This assumes that we detect higher-than-predicted activity, and execute additional recovery steps.

VI. DICTIONARY TRAINING AND CONFIGURATION

For low-latency prediction, we must ensure that sparse coding dictionaries can be trained quickly under changing workloads. To this end, we first demonstrate the impact of hierarchy on training time. Figure 5 shows prediction accuracy as the number of training samples increases, comparing a 2 layer model with a 1 layer model that codes concatenated measurement vectors directly.

Prediction accuracy for both models follows the same basic shape. With few training samples, predictors capture single-window effects, and converge to a local maximum. As data is added, variations arising from underlying feature interrelationships are observed. Initially, this added data complexity dilutes the training set, hurting prediction. However, when enough samples are acquired to fully capture second-order effects, prediction converges to a new, higher maximum. Here, we see that hierarchically treating feature interrelationships reduces training data requirements by roughly $3x$. Different scenarios. $a_o = 0.75$ represents ordinary operation, based on the observed average activity during BBENCH. $a_{rd}$ represents reduced activity during dips in instruction throughput. Finally $a_{plt}$ represents additional penalty activity to compensate for incorrect scaling decisions.

When instruction throughput drops, gating is used to reduce switching activity. However, gating mechanisms are imperfect and design specific, so we use a gating efficiency term $g$ to compare different scenarios. $a_{rd}$ is therefore defined as $a_{rd} = 1.0 - (0.75 + g)$ for an instruction throughput drop of $75\%$.

The model consists of terms representing power consumption for correct predictions, false negatives, and false positives, respectively. When a dip is correctly predicted, voltage and frequency are reduced from $V_o = 1.0$ to $V_{\text{rd}} = 0.25$, and $f_o = 1.0$ to $f_{\text{rd}} = 0.25$. For false negatives, voltage and frequency remain at $V_o = 1.0$ and $f_o = 1.0$. For false positives, we penalize incorrect scaling with increased switching activity: $(a_o + a_{plt})$. This assumes that we detect higher-than-predicted activity, and execute additional recovery steps.

Figure 4 plots $P_{\text{dyn}}$ per dip for our best sparse coding model, against look-ahead range. Gating efficiency establishes the do-nothing baseline power consumption during a dip, so we vary $g = 0.15...0.66$ to capture savings for a range of chip designs. Power savings are also parameterized by the recovery cost $a_{plt}$, which we vary from $+10\%$ to $+40\%$ switching activity. When the recovery cost is $+25\%$ activity, predictive voltage scaling successfully reduces power consumption with a $4$ window heads up, or $2\text{ms}$. If we can tolerate only a $1\text{ms}$ chip adjustment time, then this savings is a $50\%$ gain over a $g = 0.33$ gating-efficient design without voltage scaling.

<table>
<thead>
<tr>
<th>Workload Name</th>
<th>Phase</th>
<th>Description</th>
<th>% Low-Load Throughput</th>
<th>S. C. Pred. Acc.</th>
<th>Look-Ahead 2</th>
<th>Look-Ahead 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acrobat Reader</td>
<td>All</td>
<td>Display PDF file</td>
<td>93.7%</td>
<td>98.3%</td>
<td>98.5%</td>
<td>98.7%</td>
</tr>
<tr>
<td>Frozen Bubble</td>
<td>Phase 2</td>
<td>Initialize and begin game</td>
<td>23.5%</td>
<td>94.0%</td>
<td>88.9%</td>
<td></td>
</tr>
<tr>
<td>KV Mail</td>
<td>All</td>
<td>Display emails</td>
<td>59.7%</td>
<td>89.5%</td>
<td>85.7%</td>
<td></td>
</tr>
<tr>
<td>KingSoft Office</td>
<td>Phase 2</td>
<td>Open doc/xps/ppt files</td>
<td>48.9%</td>
<td>88.7%</td>
<td>85.5%</td>
<td></td>
</tr>
<tr>
<td>MXPlayer</td>
<td>Phase 1</td>
<td>Play video</td>
<td>51.8%</td>
<td>91.0%</td>
<td>85.4%</td>
<td></td>
</tr>
<tr>
<td>Frozen Bubble</td>
<td>Phase 2</td>
<td>Play mp3</td>
<td>11.8%</td>
<td>97.6%</td>
<td>95.1%</td>
<td></td>
</tr>
<tr>
<td>Frozen Bubble</td>
<td>Phase 2</td>
<td>Play video</td>
<td>6.6%</td>
<td>98.7%</td>
<td>97.8%</td>
<td></td>
</tr>
<tr>
<td>MXPlayer</td>
<td>Phase 2</td>
<td>Play audio</td>
<td>23.5%</td>
<td>91.0%</td>
<td>88.7%</td>
<td></td>
</tr>
<tr>
<td>MXPlayer</td>
<td>Phase 2</td>
<td>Play video</td>
<td>11.8%</td>
<td>97.6%</td>
<td>95.1%</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE I. PREDICTION ACCURACY BY WORKLOAD**

Fig. 5. Initially, sparse coding models initially converge to a local maximum prediction accuracy, based on the predictable power of single-window features. When more variations driven by underlying interrelationships are observed, additional training is needed to capture their predictive power. A 2 layer model learns these second-order effects from roughly $3x$ fewer training samples.

For simplicity, this model assumes constant capacitance and linear frequency/voltage scaling. As in [18], we proportionally scale power by the amount of realized switching activity in PDF and Microsoft Office documents. For each workload, we report the percentage of computation time during which instruction throughput is below $25\%$, and the prediction accuracy using signatures spanning 2 windows.
Deep learning holds much promise for workload modeling and adaptive chip optimization. We believe this study is an important first step to begin exploring those opportunities.