Prediction-Guided Performance-Energy Trade-off for Interactive Applications

Daniel Lo, Taejoon Song, and G. Edward Suh
Cornell University
Ithaca, NY, USA
{dl575, ts693, gs272}@cornell.edu

ABSTRACT

Many modern mobile and desktop applications involve real-time interactions with users. For these interactive applications, tasks must complete in a reasonable amount of time in order to provide a responsive user experience. Conversely, completing a task faster than the limits of human perception does not improve the user experience. Thus, for energy efficiency, tasks should be run just fast enough to meet the response-time requirement instead of wasting energy by running faster. In this paper, we present a predictive DVFS controller that predicts the execution time of a job before it executes in order to appropriately set the DVFS level to just meet user response-time deadlines. Our results show 56% energy savings compared to running tasks at the maximum frequency with almost no deadline misses. This is 27% more energy savings than the default Linux interactive power governor, which also shows 2% deadline misses on average.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: Hardware / Software Interfaces

Keywords

DVFS, energy efficiency, run-time prediction

1. INTRODUCTION

Many modern applications on mobile and desktop systems are highly interactive. That is, users will provide inputs and expect a response in a timely manner. For example, games must read in user input, update game state, and display the result within a small time window in order for the game to feel responsive. Previous studies on human-computer interaction have shown that latencies of 100 milliseconds are required in order to maintain a good experience [1, 2, 3] while variations in response time faster than 50 milliseconds are imperceptible [4, 5]. These tasks are effectively soft real-time tasks which have a user response-time deadline. The task must finish by the deadline for good user experience, but finishing faster does not necessarily improve the user experience due to the limits of human perception.

As finishing these tasks faster is not beneficial, we can use power-performance trade-off techniques, such as dynamic voltage and frequency scaling (DVFS), in order to reduce energy usage while maintaining the same utility to the user. By reducing the voltage and frequency, we can run the task slower and with less energy usage. As long as the task still completes by the deadline, then the experience for the user remains the same.

Existing Linux power governors [6] do not take into account these response-time requirements when adjusting DVFS levels. More recent work has looked at using DVFS to minimize energy in the presence of deadlines. However, these approaches are typically reactive, using past histories of task execution time as an estimate of future execution times [7, 8, 9, 10]. However, the execution time of a task can vary greatly depending on its input. Reactive controllers cannot respond fast enough to input-dependent execution time variations. Instead, proactive or predictive control is needed in order to adjust the operating point of a task before it runs, depending on the task inputs. This has been explored for specific applications [11, 12, 5, 13], but these approaches were developed using detailed analysis of the applications of interest. As a result, they are not easily generalizable to other domains.

In this paper, we present an automated and general framework for creating prediction-based DVFS controllers. Given an application, we show how to create a controller to predict the appropriate DVFS level for each task execution, depending on its input and program state values, in order to satisfy response-time requirements. Our approach is based on recognizing that the majority of execution time variation can be explained by differences in control flow. Given a task, we use program slicing to create a minimal code fragment that will calculate control-flow features based on input values and current program state. We train a

MICRO-48 December 05-09, 2015, Waikiki, HI, USA
 ISBN 978-1-4503-4304-2/15/12 $15.00
DOI: http://dx.doi.org/10.1145/2830772.2830776.

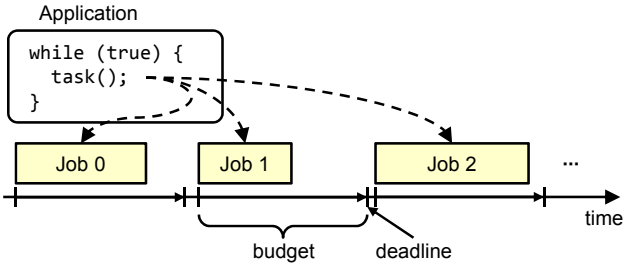

Figure 1: Example of tasks, jobs, and deadlines.

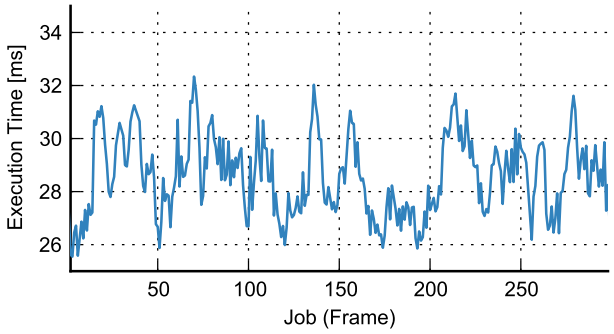

Figure 2: Execution time of jobs (frames) for ldecode (video decoder).

model to predict the execution time for a task given these features. At run-time, we are able to quickly run this code fragment and predict the execution time of a task. With this information, we can appropriately select a DVFS level in order to minimize energy while satisfying response-time requirements. Our main contributions include:

1. An automated approach for generating control flow features.
2. A method for training a model to map features to execution times in such a way as to be conservative and avoid deadline misses.
3. Application of these predictions to DVFS control in order to minimize energy in the presence of deadlines.

We implemented a prototype of this method on an ODROID-XU3 development board. We tested eight different applications, including games, speech recognition, video decoding, and a web browser. Our results show energy savings of 56% over running at maximum frequency with almost no deadline misses. This is 27% energy savings compared to the default Linux interactive governor which shows 2% deadline misses. Compared to a PID-based governor, our approach sees only a 1% improvement in energy savings but the PID-based controller shows 13% deadline misses on average.

The rest of the paper is organized as follows. Section 2 discusses general characteristics of interactive applications and the challenges of designing a DVFS controller to take advantage of execution time variation. Section 3 discusses our method of predicting execution time and using this to inform DVFS control. Section 4 discusses the overall framework and system-level issues. Section 5 shows our evaluation results. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2. EXECUTION TIME VARIATION IN INTERACTIVE APPLICATIONS

2.1 Tasks and Jobs

We define a *task* as a portion of an application that has an associated response-time requirement. We refer to the time period in which it must complete as its *time budget*. For example, games are typically written with a main task that handles reading user input, updating game state, and displaying the output. In order to maintain a certain frame rate (e.g., 30 frames per second), this task must finish within the frame period budget (e.g., 33 milliseconds for 30 frames per second operation).

We define a *job* as a dynamic instance of a task. Figure 1 shows how a task maps to multiple jobs. Each job has a deadline which is the time by which it must finish execution. For example, for a game running at 30 frames-per-second, 30 jobs for the game loop task are run each second. Each of these jobs has a deadline which is 33 milliseconds after the job's start time. These jobs all correspond to the same set of static task code, but their dynamic behavior differs due to different inputs and program state. For example, one job may see that the user has pressed a button and thus execute code to process this button press. Other jobs may see no new user input and skip the need to process user input. As a result, job execution times can vary depending on input and program state.

2.2 Variations in Execution Time

These variations in execution times between jobs can be significant. Figure 2 shows the execution time per job (frame) for a video decoder application (ldecode) running on an ODROID-XU3 development board (see Section 5 for full experimental setup). We can see that there are large variations in execution time from job to job due to differences in input and program state when each job executes. Because of this large variation, setting the appropriate DVFS level is a difficult problem.

Using a single DVFS level based on the average execution time (28.6 milliseconds) will lead to a number of jobs missing their deadline. On the other hand, looking at the worst-case execution time (32.3 milliseconds) implies that the application must be run at its maximum frequency, which means that minimal energy saving from DVFS is possible. The large variations from job to job imply that we need a fine-grained, per-job decision of the DVFS level to use in order to minimize energy usage while avoiding deadline misses.

2.3 Existing DVFS Controllers

There are a large number of existing and proposed

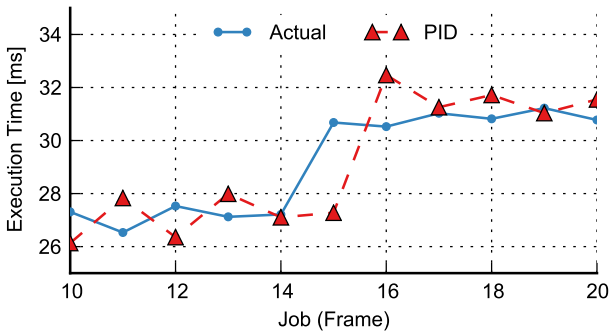

Figure 3: Execution time of jobs (blue, solid) and execution time expected by a PID controller (red, dashed) for ldecode (video decoder).

DVFS controllers. Most controllers, such as the built-in Linux governors [6], adjust DVFS based on CPU utilization. When utilization is high, voltage and frequency are increased, while when utilization is low, voltage and frequency are decreased. This does not explicitly take into account deadlines and can result in high energy usage or deadline misses. For example, high CPU utilization can cause a high voltage and frequency level to be used. However, the time budget for the task may actually be very long and a lower voltage and frequency would be sufficient, resulting in lower energy usage. Similarly, CPU utilization for a job could be low due to memory stalls, causing the controller to lower voltage and frequency levels. However, if the task has a tight time budget, then this can result in a deadline miss, whereas running at higher frequencies may have been able to meet the deadline.

DVFS has been explored in hard real-time systems in order to save energy while guaranteeing that deadlines are met [14, 15, 16, 17]. In order to ensure that deadlines are never missed, the analysis must be strictly conservative which limits the amount of energy that can be saved. That is, a task will always be run at a frequency such that even the slowest jobs will meet the deadline. However, for most applications, there is no need to be this conservative and increased energy savings can be achieved by relaxing these constraints.

Other work has explored using run-time information to inform DVFS control in the presence of deadlines. These approaches are largely reactive and use information about past job execution times to predict future job times [7, 8, 9, 10]. This can capture coarse-grained phase changes in execution time, but cannot capture the fine-grained job-to-job variations in execution times as the adjustment in DVFS level happens too late. For example, Figure 3 shows the expected execution times that a PID-based controller uses for setting the DVFS level and the real execution times of the jobs. As we can see, the PID controller's decision lags the actual execution times of the jobs.

More recently, people have investigated predicting job execution time and setting DVFS in order to meet deadlines for specific applications (e.g., game rendering [11], web browsing [12, 5], web server/Memcached

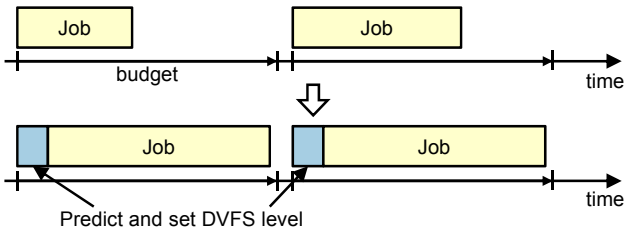

Figure 4: Overview of prediction-based control.

[13]). These approaches involved careful analysis of the application of interest, requiring extensive programmer effort, in order to design the controller. As a result, the resulting controllers cannot be applied to other applications.

2.4 Prediction-Based Control

Our goal in this work is to develop a general and automated framework that will, given a task and its time budget requirement, create a prediction-based DVFS controller that can minimize energy usage without missing deadlines. Figure 4 shows an overview of the operation of our proposed prediction-based controller. The basic idea is to pre-pend tasks with a small segment of code. This segment of code will predict the appropriate DVFS level to use for each of the task's jobs depending on the job's input and current program state.

The main source of execution time variation between jobs is due to different inputs and program state. Thus, the main challenge in creating a prediction-based DVFS controller is determining how to map job input and program state values to the appropriate DVFS frequency level. In general, finding a direct mapping from input values to frequency levels is challenging because the mapping can be irregular and complicated. In addition, this mapping varies from application to application. For example, for one application, pressing the "down" key may correspond to a large increase in execution time while for other applications it may have no effect on execution time. Our solution is to take advantage of the program source to give us hints about how input values and program state will affect execution time. We use the program source to automatically generate a prediction-based DVFS controller.

3. EXECUTION TIME PREDICTION FOR ENERGY-EFFICIENCY

3.1 Overview

The basic intuition behind our prediction methodology is that, to first-order, execution time correlates with the number of instructions run. Variations in the number of instructions run are described by the control flow taken by a specific job. For example, consider the control flow graph for a task shown in Figure 5. Each node is marked with its number of instructions. Taking the left branch instead of the right branch corresponds to nine more instructions being executed. Similarly, each additional loop iteration of the last basic block adds five

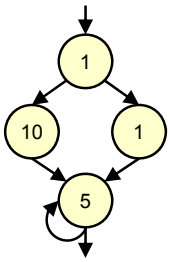

Figure 5: Example control flow graph. Each node is annotated with its number of instructions.

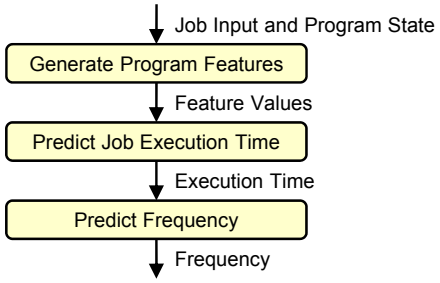

Figure 6: Steps to predict appropriate frequency level from job input and program state.

instructions to the number of instructions executed. By knowing which branch is taken and the number of loop iterations, we can know the number of instructions executed and estimate the execution time. With an estimate of the execution time, we can then estimate the performance impact of DVFS and choose an appropriate frequency and voltage level to run at in order to just meet the deadline.

Figure 6 shows the main steps in our method. We first instrument the task source code and use program slicing to create a code fragment that will calculate control flow features for a job. The code fragment is run before a job executes in order to generate the control flow features (Section 3.2). Next, we use a linear model, which we train off-line, to map control flow features to an execution time estimate for the job (Section 3.3). Finally, we use classical linear models [18, 19] that describe the frequency-performance trade-off of DVFS to select an appropriate frequency (Section 3.4).

3.2 Program Features

The first step needed for our prediction is to generate control flow features. That is, we want to know the control flow of a task when executing with a specific input and program state. For this purpose, we instrument the task source to count these control flow features. We instrument the task to count the following features:

- Number of iterations for each loop
- Number of times each conditional branch is taken
- Address of each function pointer call

Figure 7 shows examples of how these features are instrumented. We focus on control flow features because these explain most of the execution time variation. However, other features, such as variable values or memory

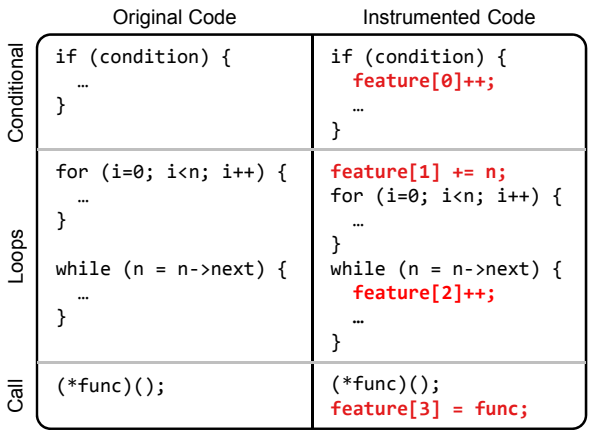

Figure 7: Example of feature counters inserted for conditionals, loops, and function calls.

accesses, could be included to improve the prediction accuracy.

Generating these features using an instrumented version of the task code is not suitable for prediction because the instrumented task will take at least as long as the original task to run. Instead, we need to quickly generate these features before the task execution. In order to minimize the prediction execution time, we use program slicing [20, 21] to produce the minimal code needed to calculate these features. Figure 8 shows a simple example of this flow. By removing the actual computation and only running through the control flow, the execution time can be greatly reduced. Since the information from this slice will ultimately be used to make a heuristic decision on DVFS control, the slice does not need to perfectly calculate the features. Instead, using an approximate slice can reduce the slice's size and execution time. For example, our tool tracks dependences based only on variable names and ignores possible pointer aliasing. As long as inaccuracies in the generated features are low, an approximate slice is adequate for our prediction needs. We refer to the resulting program slice that computes the control flow features as the *prediction slice* or simply as the *slice*.

One problem that arises with running this prediction slice before a task is the issue of side-effects. That is, the slice could write to global variables and break the correctness of the program. In order to prevent this, the slice creates local copies of any global variables that are used. Values for these local copies are updated at the start of the slice and writes are only applied to the local copy. A similar process is applied to any arguments that are passed by reference.

3.3 Execution Time Prediction Model

Next, we need to predict the execution time from the control flow features. This section describes our model that maps features to execution time. Table 1 summarizes the variables and notation that are used in this section. We use a linear model to map features to execution time as this captures the basic correlation. Higher-order or non-polynomial models may provide better ac-

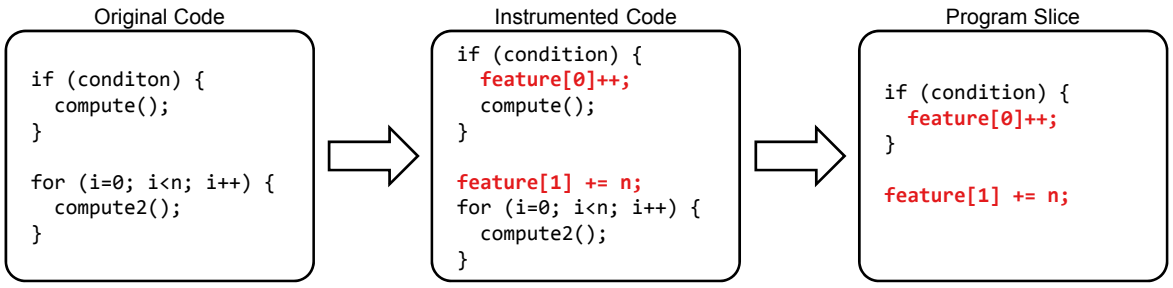

Figure 8: Example of program slicing for control flow features.

Variable	Type	Description
$\bar{y}$	Scalar	Predicted execution time
$\mathbf{x}$	Vector	Feature values
$\boldsymbol{\beta}$	Vector	Model coefficients
$\mathbf{y}$	Vector	Profiled execution times
$\mathbf{X}$	Matrix	Profiled feature values
$\mathbf{X}\boldsymbol{\beta}-\mathbf{y}$	Vector	Prediction errors
α	Scalar	Under-predict penalty weight
γ	Scalar	Number of terms penalty weight
$\|\cdot\|$	Scalar	L2-norm (sum of squares)
$\|\cdot\|_1$	Scalar	L1-norm (sum of absolute values)

Table 1: Variable and notation descriptions.

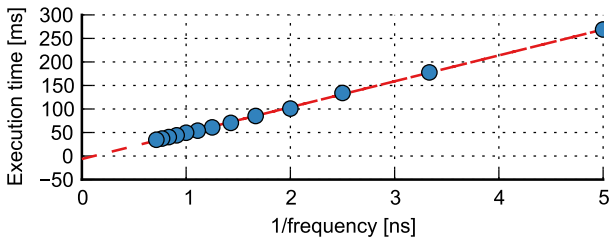

Figure 9: Average execution time of jobs (frames) for ldecode (video decoding) as frequency level varies.

curacy. However, a linear model has the advantage of being both simple to train and fast to evaluate at runtime. In addition, it is always convex which allows us to use convex optimization-based methods to fit the model. Our linear model can be expressed as

$$\bar{y} = \mathbf{x}\boldsymbol{\beta}$$

where $\bar{y}$ is the predicted execution time, $\mathbf{x}$ is a vector of feature values, and $\boldsymbol{\beta}$ are the coefficients that map feature values to execution time. These $\boldsymbol{\beta}$ coefficients are fit using profiling data. We profile the program to produce a set of training data consisting of execution times $\mathbf{y}$ and feature vectors $\mathbf{X}$ (i.e., each row of $\mathbf{X}$ is a vector of features, $\mathbf{x}_i$, for one job). Note that for addresses recorded for function calls, each unique address represents a different control flow which can correlate to a different effect on execution time. In order to properly represent each address as a feature, addresses recorded for function calls are converted to a one-hot encoding indicating whether particular function addresses were called or not.

The most common way to fit a linear model is to use least squares regression. Least squares regression finds the coefficients $\boldsymbol{\beta}$ that minimize the mean square error:

$$\min_{\boldsymbol{\beta}} \quad \|\mathbf{X}\boldsymbol{\beta}-\mathbf{y}\|^2$$

Essentially, this aims to minimize the sum of the absolute errors in the prediction. That is, it weighs negative and positive errors equally. However, these two errors lead to different behaviors on our system. Negative errors (under-prediction) lead to deadline misses since we predict the job to run faster than its actual execution time. On the other hand, positive errors (over-prediction) result in an overly conservative frequency setting which does not save as much energy as possible. In order to maintain a good user experience, we would prefer to avoid deadline misses, possibly at the cost of energy usage. In other words, we should place greater weight on avoiding under-prediction as opposed to over-prediction.

We can place greater weight on under-prediction by modifying our optimization objective:

$$\min_{\boldsymbol{\beta}} \quad \|pos(\mathbf{X}\boldsymbol{\beta}-\mathbf{y})\|^2 + \alpha\|neg(\mathbf{X}\boldsymbol{\beta}-\mathbf{y})\|^2$$

where $pos(x) = max\{x, 0\}$ and $neg(x) = max\{-x, 0\}$ and these functions are applied element-wise to vectors. Thus, $\|pos(\mathbf{X}\boldsymbol{\beta}-\mathbf{y})\|^2$ represents the over-prediction error and $\|neg(\mathbf{X}\boldsymbol{\beta}-\mathbf{y})\|^2$ represents the under-prediction error. α is a weighting factor that allows us to place a greater penalty on under-predictions by setting $\alpha > 1$. Since this objective is convex, we can use existing convex optimization solvers to solve for $\boldsymbol{\beta}$.

Coefficients which are zero imply that the corresponding control flow features do not need to be calculated by the prediction slice. We can use this information to further reduce the size and execution time of the prediction slice. We extend our optimization objective to favor using less features by using the Lasso method [22]:

$$\min_{\boldsymbol{\beta}} \quad \|pos(\mathbf{X}\boldsymbol{\beta}-\mathbf{y})\|^2 + \alpha\|neg(\mathbf{X}\boldsymbol{\beta}-\mathbf{y})\|^2 + \gamma\|\boldsymbol{\beta}\|_1$$

where $\|\cdot\|_1$ is the L1-norm and γ is a weighting factor that allows us to trade-off prediction accuracy with the number of features needed.

3.4 DVFS Model

Given a predicted execution time, we need to estimate how the execution time will change with varying frequency. For this, we use the classical linear model

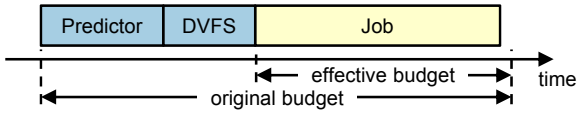

Figure 10: The effective budget decreases due to slice and DVFS execution time.

found in literature [18, 19]:

$$t = T_{mem} + N_{dependent}/f$$

where t is the execution time, T_{mem} is the memory-dependent execution time that does not scale with frequency, $N_{dependent}$ is the number of CPU cycles that do not overlap with memory and scale with frequency, and f is the frequency. In order to verify this linearity assumption, we measured average job execution times as frequency was varied. Figure 9 shows the average job execution time versus $1/f$ for ldecode (video decoder application). We can see that t and $1/f$ do show a linear relationship. We saw similar results for the other applications we tested.

By predicting the execution time at two points, we can determine T_{mem} and $N_{dependent}$ for a job and calculate the minimum frequency f to satisfy a given time budget t_{budget}. More specifically, we predict the execution time $\bar{t}_{fmin}$ at minimum frequency f_{min} and the execution time $\bar{t}_{fmax}$ at maximum frequency f_{max}. Using these two points, we can calculate T_{mem} and $N_{dependent}$ as

$$N_{dependent} = \frac{f_{min} f_{max} (\bar{t}_{fmin} - \bar{t}_{fmax})}{f_{max} - f_{min}}$$

$$T_{mem} = \frac{f_{max} \bar{t}_{fmax} - f_{min} \bar{t}_{fmin}}{f_{max} - f_{min}}$$

For a given budget t_{budget}, we want the minimum frequency f_{budget} that will meet this time. This can be calculated as

$$f_{budget} = \frac{N_{dependent}}{t_{budget} - T_{mem}}$$

Since execution time can vary even with the same job inputs and program state, we add a margin to the predicted execution times used (t_{fmin} and t_{fmax}). In our experiments we used a margin of 10%. A higher margin can decrease deadline misses while a lower margin can improve the energy savings. The resulting predicted frequency is the exact frequency that we expect will just satisfy the time budget. However, DVFS is only supported for a set of discrete frequency levels. Thus, the actual frequency we select is the smallest frequency allowed that is greater than f_{budget}.

The execution of the prediction slice and DVFS switch reduces the amount of time available for a job to execute and still satisfy its budget. Thus, the effective budget when choosing a frequency to run at needs to consider these overheads (see Figure 10). Although the execution time of the prediction slice can be measured, the DVFS switching time must be estimated, as the switch has not been performed yet. This is done by microbenchmarking the DVFS switching time. Figure 11 shows

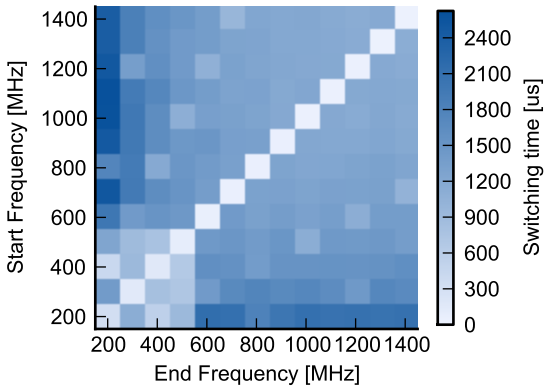

Figure 11: 95th-percentile switching times for DVFS.

the 95th-percentile DVFS switching times for our test platform for each possible start and ending frequency. We use the 95th-percentile switching times in order to be conservative in our estimate of DVFS switching time while omitting rare outliers.

3.5 Extensions for Prediction Models

In this section, we have described our specific prediction strategy for each step in our overall prediction flow shown in Figure 6. However, we note that each step in this prediction flow can be substituted with alternate models as long as it produces the needed prediction for the next step. The most obvious change would be to use more complex prediction models for each step (e.g., more features generated and higher-order, non-linear models). For the benchmarks we evaluated, we saw relatively little gain to be had from improved prediction (see Section 5.3) and thus the increased overheads of more complex models were not justified. Instead, we discuss here some potential extensions.

For feature generation, we have focused on automated generation in order for the approach to be general and limit the need for domain-specific expertise. However, this does not preclude the programmer from manually adding "hints" that they expect would correlate well with a job's execution time. For example, the programmer may be able to extract meta-data from input files and manually provide these as features.

One interesting extension to execution time prediction involves modifying the prediction in order to influence which features need to be generated. Additional constraints could be added to the execution time prediction in order to limit the use of features which require high overhead to generate. Features over some overhead threshold could be explicitly disallowed or the overhead for each feature could be introduced as penalties in the optimization objective.

The last step in our flow focuses on selecting an appropriate frequency level for DVFS control. However, this last step could be substituted to support other performance-energy trade-off mechanisms, such as heterogeneous cores. By using alternate models for how the execution time scales with the performance-energy trade-off mechanism, an appropriate operating point

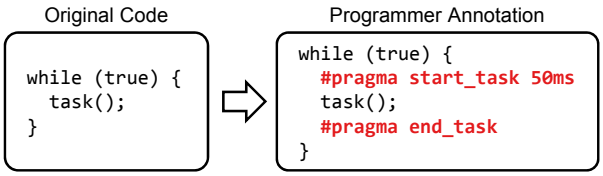

Figure 12: Example of programmer annotation to mark task boundaries and time budgets.

can be selected for the mechanism of interest.

4. SYSTEM-LEVEL FRAMEWORK

This section describes the overall framework and operation of our prediction-based controller.

4.1 Programmer Annotation

In order to apply our framework to an application, we require the individual tasks and their time budgets to be identified. These are identified by programmer annotations. The programmer must annotate the start and the end of a task and the desired response-time requirement. Figure 12 shows an example of this annotation. For ease of analysis and to ensure that tasks that start always end, we require the start and end of a task to be within one function. Arbitrary code paths can be modified to fit this model by using a wrapper function or re-writing the code. Multiple non-overlapping tasks can be supported, though we only considered one task in the applications we tested.

4.2 Off-line Analysis

Figure 13 shows the overall flow of our framework for creating prediction-based DVFS controllers. Given programmer annotation to identify tasks, we can automatically instrument these tasks to record control flow features. Off-line, we profile these tasks in order to collect traces of feature values and job execution times. This is used to train our execution time prediction model, as described in Section 3.3. Since execution time depends on the specific hardware and platform that an application is run on, profiling and model training needs to be done for the platform that the application will be run on. For common platforms, the program developer can perform this profiling and distribute the trained model coefficients with the program. Alternatively, profiling can be done by the user during application installation.

The trained execution time model only requires a subset of all the features to perform prediction. Specifically, features whose model coefficients are zero can be excluded from the prediction slice. Program slicing is used to create a minimal code fragment to calculate only the needed control flow features. Note that since the features needed depends on the training of the execution time prediction model, which is platform-dependent, the features needed could vary across platforms. However, we expect the features that are needed are primarily a function of the task semantics (i.e., execution time variations across control paths) rather than the platform it is run on. In fact, we compared the predictions made for an x86-based (Intel Core i7) platform when using the features selected for an ARM-based ODROID-XU3 platform and for the features selected for the x86 platform itself. For all but three of the benchmarks we tested, the features selected were exactly the same. For one of these three benchmarks, the features selected by the x86 platform were a subset of those selected by the ARM platform and so the predicted times were exactly the same. For the remaining two benchmarks, the predicted times differed by less than 3%. Although we had to re-train the execution time model coefficients, the same prediction slice was applicable across both platforms.

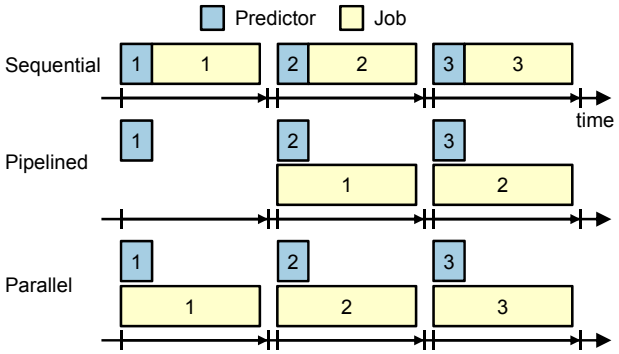

Figure 14: Options for how to run predictor.

4.3 Run-time Prediction

The prediction slice, execution time predictor, and frequency predictor are combined to form the *DVFS predictor* or simply *predictor*. There are several options for how to run the predictor in relation to jobs. Figure 14 shows some of these options. The simplest approach is to run the slice just before the execution of a job. This uses up part of the time budget to perform slicing, as mentioned in Section 3.4. However, if this time is low, then the impact is minimal.

An alternative option would be to run the predictors and jobs in a parallel, pipelined manner such that during job i, the predictor for job $i+1$ is run. This ensures that the DVFS decision is ready at the start of a job with no impact on time budget from the predictor. However, this assumes that information needed by the prediction slice, specifically the job inputs and program state, is ready one job in advance. This is not possible for interactive tasks which depend on real-time user inputs or tasks which are not periodic.

The predictor could also be run in parallel with the task. This avoids the issue of needing input values early. In terms of time budget, this mode of operation still reduces the effective budget by the predictor execution time because an appropriate frequency cannot be selected until after the predictor is run. However, part of the task also executes during the prediction time. By accounting for this, the energy savings may be higher than running in a sequential manner. Running in parallel also avoids the issue of side-effects caused by the prediction slice that was discussed in Section 3.2. However, running in parallel either requires forking off the predictor for each job or sharing data with a dedicated

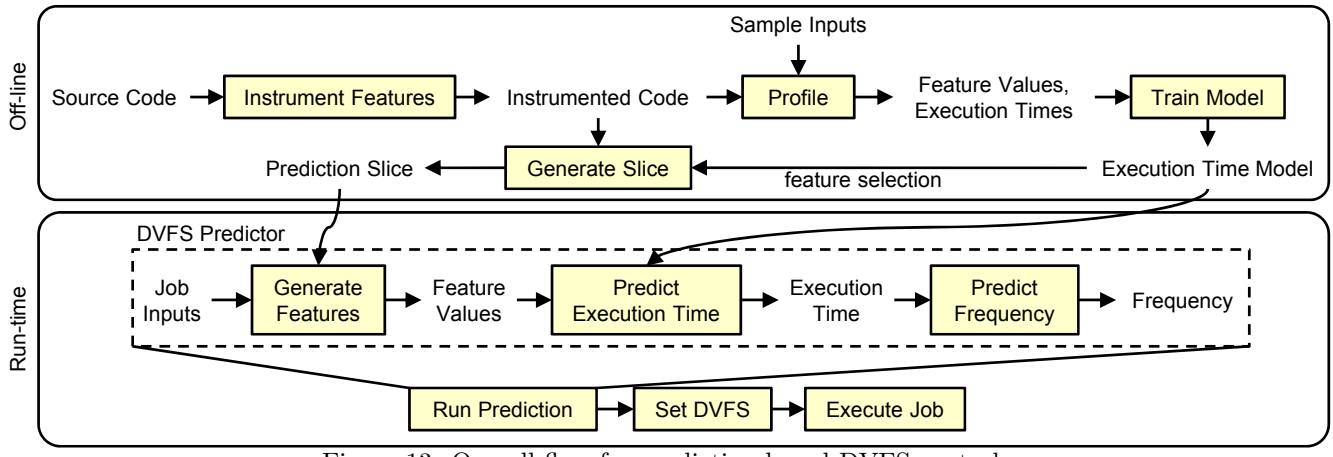

Figure 13: Overall flow for prediction-based DVFS control.

predictor thread, both of which can introduce overhead.

For our target applications, we found that the execution time of the predictor was low. Thus, we decided to run the predictor and task in a sequential manner. For applications which require more complicated predictors, these alternative operation modes may be beneficial.

5. EVALUATION

5.1 Experimental Setup

We applied our framework for prediction-based DVFS control to a set of eight benchmark applications including three games, a web browser, speech recognition, a video decoder and two applications from the MiBench suite [27]. Table 2 lists and describes these benchmarks. It also shows the minimum, average, and maximum job execution times for these benchmarks when run at maximum frequency. User inputs for the games and the web browser were scripted to ensure consistency across runs.

We ran our experiments on an ODROID-XU3 [30] development board running Ubuntu 14.04. The ODROID-XU3 includes a Samsung Exynos5422 SoC with ARM Cortex-A15 and Cortex-A7 cores. We show results here for running on the more power-efficient A7 core but we saw similar trends when running on the A15 core. In order to isolate measurements for the application of interest, we pinned the benchmark to run on the A7 core while using the A15 to run OS and background jobs. We measured power using the on-board power sensors with a sampling rate of 213 samples per second and integrated over time to calculate energy usage.

We compare our proposed prediction-based DVFS controller with existing controllers and previously proposed control schemes. Specifically, we measure results for the following DVFS schemes:

1. **performance**: The Linux performance governor [6] always runs at the maximum frequency. We normalize our energy results to this case.

2. **interactive**: The Linux interactive governor [6] was designed for interactive mobile applications. It samples CPU utilization every 80 milliseconds and changes to maximum frequency if CPU utilization is above 85%.

3. **pid**: The PID-based controller uses previous prediction errors with a PID control algorithm in order to predict the execution time of the next job [7]. The PID parameters are trained offline and are optimized to reduce deadline misses.

4. **prediction**: This is our prediction-based controller as described in this paper.

5.2 Energy Savings and Deadline Misses

Figure 15 compares energy consumption and deadline misses for the different DVFS controllers across our benchmark set. These experiments are run with a time budget of 50 ms per job as running faster than this is not noticeable to a user [4, 5]. pocketsphinx takes at least 100s of milliseconds to run (see Table 2) so we use a 4 second deadline. This corresponds to the time limit that a user is willing to wait for a response [3]. Energy numbers are normalized to the energy usage of the performance governor. Deadline misses are reported as the percentage of jobs that miss their deadline.

We see that, on average, our prediction-based controller saves 56% energy compared to running at max frequency. This is 27% more savings than the interactive governor and 1% more savings than the PID-based controller. For ldecode, pocketsphinx, and rijndael, prediction-based control shows higher energy consumption than PID-based control. However, if we look at deadline misses, we see that PID-based control shows a large number of misses for these benchmarks. On average, the interactive governor shows 2% deadline misses and the PID-based controller shows 13% misses. In contrast, our prediction-based controller shows 0.1% deadline misses for curseofwar and no deadline misses for the other benchmarks tested. Overall, we see that the interactive governor has a low number of deadline misses, but achieves this with high energy consumption. On the other hand, PID-based control shows lower energy usage than the prediction-based controller in some cases, but this comes at the cost of a large number of

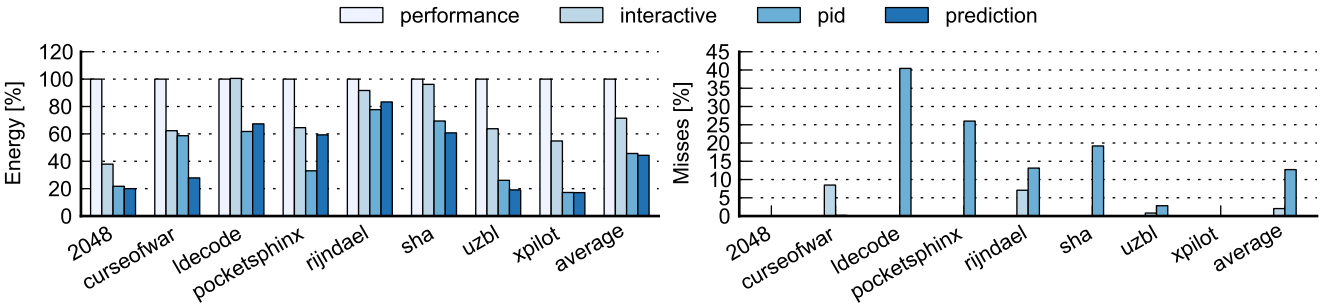

Figure 15: Normalized energy usage and deadline misses.

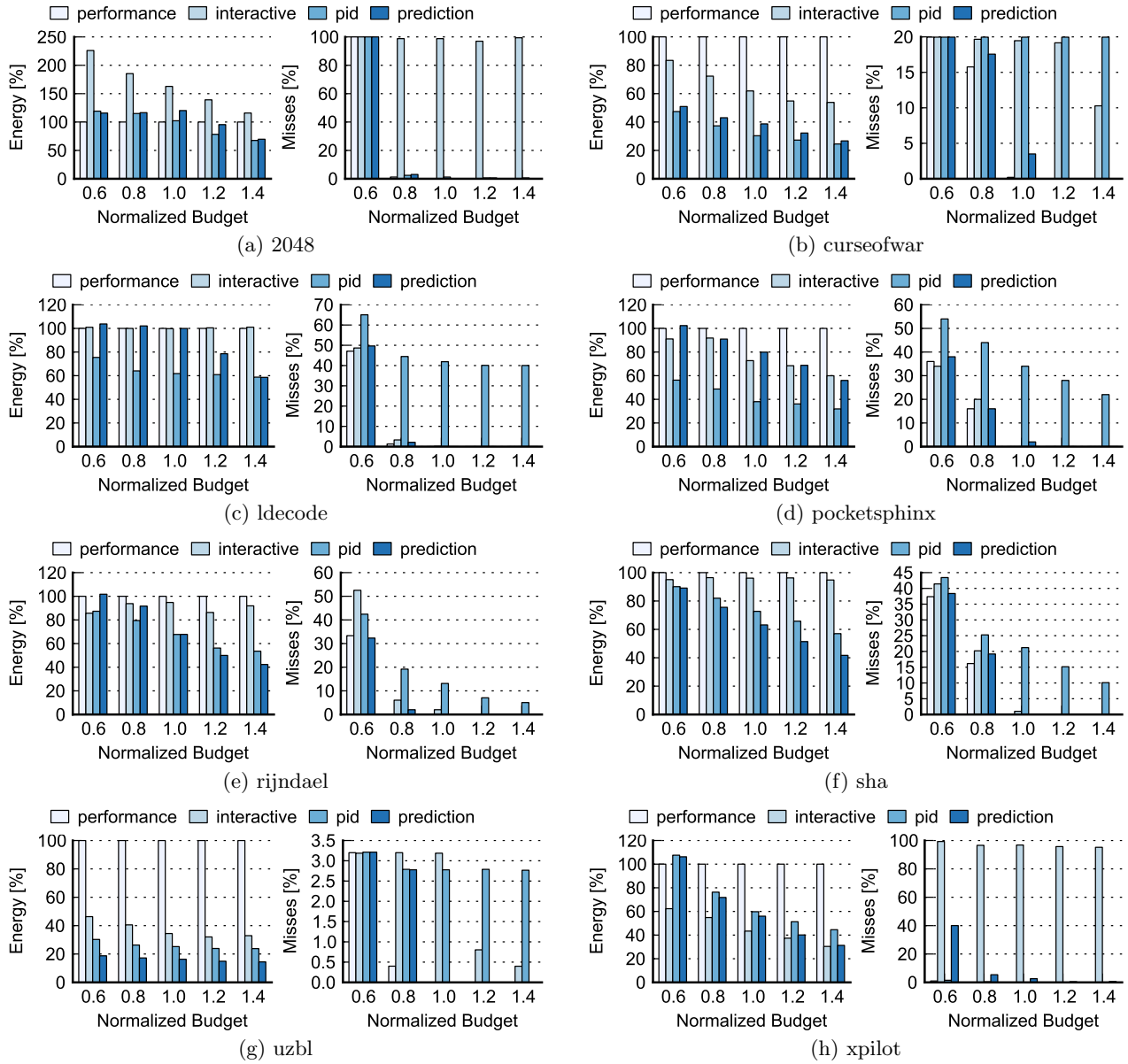

Figure 16: Normalized energy usage and deadline misses as time budget is varied.

Benchmark	Description	Task	Job Times [ms]		
			Min	Avg	Max
2048 [23]	Puzzle game	Update and render one turn	0.52	1.2	2.1
curseofwar [24]	Real-time strategy game	Update and render one game loop iteration	0.02	6.2	37.2
ldecode [25]	H.264 decoder	Decode one frame	6.2	20.4	32.5
pocketsphinx [26]	Speech recognition	Process one speech sample	718	1661	2951
rijndael [27]	Advanced Encryption Standard (AES)	Encrypt one piece of data	14.2	28.5	43.6
sha [27]	Secure Hash Algorithm (SHA)	Hash one piece of data	4.7	25.3	46.0
uzbl [28]	Web browser	Execute one command (e.g., refresh page)	0.04	2.2	35.5
xpilot [29]	2D space game	Update and render one game loop iteration	0.2	1.3	3.1

Table 2: Benchmark descriptions and execution time statistics when running at maximum frequency.

deadline misses. Instead, on average, our prediction-based control is able to achieve both better energy consumption and less deadline misses than the interactive governor and PID-based control.

Since our prediction-based controller takes the time budget into account, it is able to save more energy with longer time budgets. Similarly, with shorter time budgets, it will spend more energy to attempt to meet the tighter deadlines. In order to study this trade-off, we swept the time budget around the point where we expect to start seeing deadline misses. Specifically, we set a normalized budget of 1 to correspond to the maximum execution time seen for the task when running at maximum frequency (see Table 2). This corresponds to the tightest budget such that all jobs are able to meet their deadline. Figure 16 shows the energy usage and deadline misses for the various benchmarks as the normalized budget is swept below and above 1. We see that our prediction-based controller is able to save more energy with longer time budgets and continues to outperform the interactive governor and the PID-based controller. For normalized budgets less than 1, our prediction-based controller shows deadline misses. However, the number of misses is typically close to the number seen with the performance governor. This implies that most of the deadline misses are ones that are impossible to meet at the specified time budget, even with running at the maximum frequency. Note that in some cases, we see normalized energy usage over 100% (i.e., energy usage greater than the performance governor). This occurs because the time to switch DVFS levels also contributes to energy usage. For extremely short deadlines (e.g., a normalized budget of 1 for 2048 corresponds to 2.1 milliseconds), this switching time can have a significant impact on energy usage. As the performance governor never switches DVFS levels, it does not pay this penalty.

5.3 Analysis of Overheads and Error

Figure 17 shows the average times for executing the predictor and for switching DVFS levels. On average, the predictor takes 3.2 ms to execute and DVFS switching takes 0.8 ms. Excluding pocketsphinx, the average total overhead is less than 1 ms which is 2% of a 50 ms time budget. pocketsphinx shows a long execution time for the predictor. However, this time is negligible compared to the execution time of pocketsphinx jobs which are on the order of seconds.

The overheads of executing the predictor and DVFS switching decrease the energy savings achievable. This is due to the energy consumed to perform these operations as well as the decrease in effective budget. Better program slicing and/or feature selection could reduce the predictor execution time. Similarly, faster DVFS switching circuits [31, 32, 33] have shown switching times on the order of tens of nanoseconds. In order to explore the limits of what is achievable, we evaluated our prediction-based control when the overheads of the predictor and DVFS switching are ignored. Figure 18 shows the energy and deadline misses when these overheads are removed. These results are shown for a time budget of 4 s for pocketsphinx and 50 ms for all other benchmarks. On average, we see a 3% decrease in energy consumption when removing the overheads of DVFS switching. Removing the overhead of running the predictor shows negligible improvement past removing the DVFS switching overhead.

In addition to these overheads, the accuracy of our prediction limits the effectiveness of the prediction-based controller. Figure 19 shows box-and-whisker plots of the prediction error. The box indicates the first and third quartiles and the line in the box marks the median

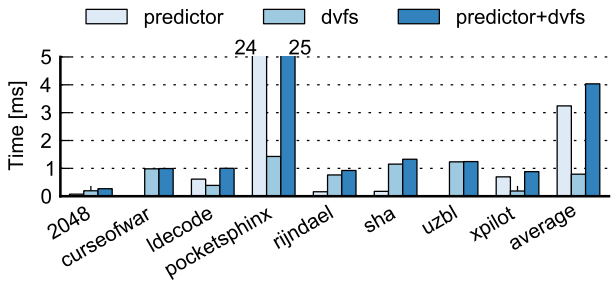

Figure 17: Average time to run prediction slice and switch DVFS levels.

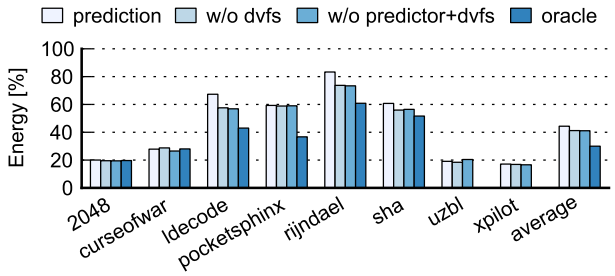

Figure 18: Normalized energy usage with overheads removed and oracle prediction.

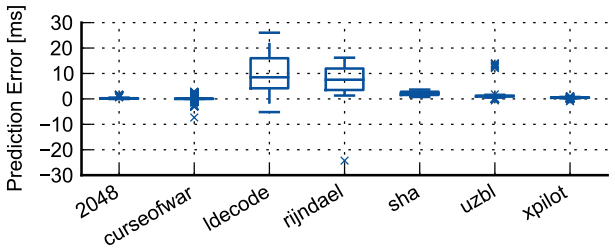

Figure 19: Prediction error where positive/negative numbers correspond to over/under-prediction.

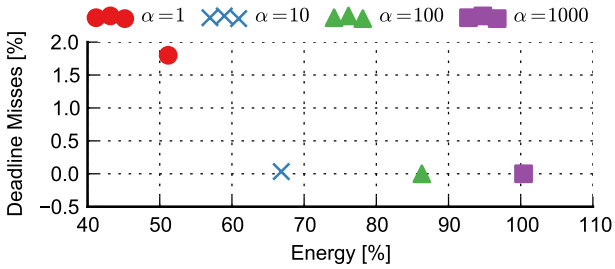

Figure 20: Energy vs. deadline misses for various under-predict penalty weights (α) for ldecode.

value. Outliers (values over 1.5 times the inner quartile range past the closest box end) are marked with an "x" and the non-outlier range is marked by the whiskers. Positive values represent over-prediction and negative-numbers represent under-prediction. We can see that the prediction skews toward over-prediction with average errors greater than 0. Most benchmarks show prediction errors of less than 5 ms, which is only 10% of a 50 ms time budget. ldecode and rijndael show higher prediction errors, which limits the energy savings possible. pocketsphinx (not shown) has errors ranging from 60 ms under-prediction to 2 seconds over-prediction with an average of 880 ms over-prediction. Although these errors are larger in absolute terms than the other benchmarks, they are on the same order of magnitude when compared to the execution time of pocketsphinx jobs.

In order to explore the possible improvements with perfect prediction, we implemented an "oracle" controller that uses recorded job times from a previous run with the same inputs to predict the execution time of jobs. Figure 18 also includes these oracle results. The oracle results include ignoring the overheads of the predictor and DVFS switching. We see that an additional 11% energy savings are achievable with oracle on top of removing the predictor and DVFS switching overheads. Note that we do not have oracle results for uzbl and xpilot as non-deterministic variations in the ordering of jobs across runs made it difficult to implement a good oracle controller.

5.4 Under-prediction Trade-off

In Section 3.3, we discussed how we can vary the penalty weight for under-prediction, α, when we fit our execution time prediction model. Placing greater penalty on under-prediction increases the energy usage but reduces the likelihood of deadline misses. Figure 20 shows the energy and deadline misses for varying under-predict penalty weights for ldecode. We see that as the weight is decreased, energy consumption decreases but deadline misses increase. Reducing the weight from 1000 to 100 keep misses at 0, but reducing the weight to 10 introduces a small number of deadline misses (0.03%). Other benchmarks show similar trends and we found that across the benchmarks we tested, an under-predict penalty weight of 100 provided good energy savings without sacrificing deadline misses. The results in this paper have been shown with a weight of 100.

5.5 Idling Between Jobs

One possible way to further improve energy savings is to decrease the clock frequency and voltage level between jobs to a minimum. We refer to this idea as "idling". Figure 21 shows the normalized energy for each controller with and without idling applied. The energy is normalized to the energy usage of the performance governor without idling. These results are shown for a deadline of 4 seconds for pocketsphinx and 50 milliseconds for all other benchmarks and is an analogous setup to the results shown in Figure 15.

The amount of energy saved depends on the amount of idle time between jobs. The performance governor finishes jobs the fastest, and thus sees the largest additional energy savings by reducing the frequency between jobs. However, for all benchmarks except pocketsphinx, the prediction-based DVFS controller without idling still saves more energy than the performance governor with idling. For pocketsphinx, the performance governor with idling achieves similar energy usage to the prediction-based controller. However, by also reducing the frequency between jobs for the prediction-based controller, it is able to achieve further energy savings and outperform the performance governor with idling. The prediction-based controller with idling saves more energy than the performance and interactive governors for all of the benchmarks we tested. For some cases, the prediction-based controller uses more energy than the PID-based controller. However, as this technique does not affect the number of deadline misses, the PID-based controller is still missing many of its deadlines. On average, with idling enabled, the prediction-based controller uses 35% less energy than the performance and interactive governors and 18% less energy than the PID-based controller.

6. RELATED WORK

6.1 Dynamic Voltage and Frequency Scaling

There have been many designs for DVFS controllers. Most controllers look to decrease frequency when the performance impact is minimal. For example, the built-in Linux governors [6] adjust DVFS based on CPU utilization. However, these controllers do not take into account performance requirements or deadlines.

DVFS has been studied in the context of hard real-time systems [14, 15, 16, 17]. For these systems, dead-

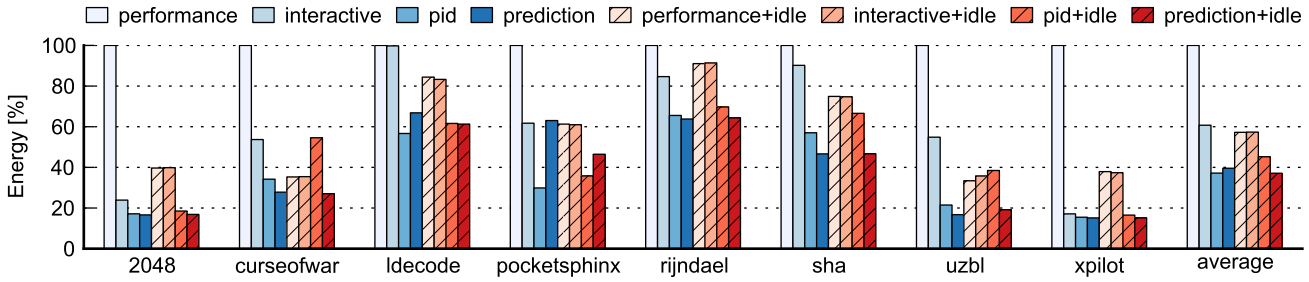

Figure 21: Normalized energy with (+idle) and without running at minimum frequency between jobs

lines are strict requirements that cannot be violated. Thus, lowering frequency must be done in a conservative manner. By relaxing this strict requirement, our prediction-based controller is able to achieve higher energy savings.

A number of reactive DVFS controllers have been proposed that use the past history of job execution times to predict the execution time of future jobs. Choi et al. [8] used moving averages of job execution time history to predict execution times for an MPEG decoder. Similarly, Pegasus [9] used instantaneous and average job execution times to make DVFS decisions. Nachiappan et al. [10] used a moving average to set DVFS for multiple IP cores. Gu and Chakraborty [7] used a PID-based controller to predict execution times of frames in a game. These history-based, reactive controllers are not able to adapt fast enough to job-to-job variations in execution time, resulting in either high energy usage or deadline misses. Our results show that prediction-based control outperforms PID-based control.

Prediction-based approaches have been designed for specific applications. Gu and Chakraborty [11] predicted the rendering time for a game frame based on the number of objects in the scene. Zhu et al. used prediction-based control to select core and DVFS levels for a web browser based on HTML and CSS features [12] and event types [5]. Adrenaline [13] looked to reduce the tail latency of datacenter applications including web search and Memcached by classifying jobs by query type. These predictive approaches required careful analysis of the applications of interest in order to identify features and create predictive models. Our approach presents an automated approach to create these DVFS predictors for a wide range of applications. For example, for the web browser we tested, our approach automatically identifies event types as a feature because of changes in control flow depending on event type.

6.2 Execution Time Prediction

Worst-case execution time analysis is a well-studied problem in hard real-time systems [34]. This analysis looks at the problem of estimating execution time of a program in the worst-case. This can be used as a conservative bound for setting DVFS in order to meet deadlines, but it does not predict the changes in job execution time based on specific inputs or program state.

ATLAS [35] looked at predicting execution time in the context of soft real-time scheduling. Their approach uses programmer-marked features in a linear model in order to predict execution time. Instead, our approach is able to automatically identify features without programmer assistance. Mantis [20] presents an automated approach for predicting execution time, similar to the approach we have presented. However, neither Mantis nor ATLAS looks at execution time prediction in the context of DVFS control. Applying execution time prediction to DVFS allocation with deadlines required creating a prediction method that placed greater penalty on under-prediction and extending the predictor to select a DVFS level.

7. CONCLUSION

In this paper, we presented a framework for prediction-based DVFS control. This controller predicts the appropriate frequency to use for a job in order to minimize energy and just meet the required deadline. Our prediction works by first generating control flow features, then predicting the execution time of the job, and finally predicting the appropriate frequency to use. Our results show 56% average energy savings over running at maximum frequency with almost no deadlines misses. On average, our prediction-based controller outperforms both the Linux interactive governor and a PID-based controller in energy savings and deadline misses.

We note that there are some limitations to our technique, which may serve as directions for future research. First, in this work we have studied only running one application at a time as mobile devices typically run one interactive application at a time. Extending this work to multi-threaded or multi-core architectures will require a way to model and estimate the contention of multiple threads or workloads. Second, we assume that inputs are known at the start of a task in order to calculate control flow features. However, tasks which receive unpredictable input in the middle create new challenges in how to perform prediction. Finally, for time budgets on the order of milliseconds, the overhead of running the predictor and switching DVFS levels will outweigh the energy savings gained. At these time scales, the preditor may need to predict the DVFS level for several jobs at once in order to amortize these overheads.

Acknowledgments

This work was partially supported by the the Office of Naval Research grant N00014-15-1-2175.

8. REFERENCES

[1] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer, "Using Latency to Evaluate Interactive System Performance," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[2] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The Information Visualizer, an Information Workspace," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1991.

[3] R. B. Miller, "Response Time in Man-computer Conversational Transactions," in *Proceedings of the 1968 Fall Joint Computer Conference, Part I*, 1968.

[4] G. Lindegaard, G. Fernandes, C. Dudek, and J. Browñ, "Attention Web Designers: You Have 50 Milliseconds to Make a Good First Impression!," *Behavior & Information Technology*, vol. 25, no. 2, 2006.

[5] Y. Zhu, M. Halpern, and V. J. Reddi, "Event-Based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications," in *Proceedings of the 21st Symposium on High Performance Computer Architecture*, 2015.

[6] D. Brodowski, "CPU Frequency and Voltage Scaling Code in the Linux^{TM}Kernel." `https://android.googlesource.com/kernel/common/+/a7827a2a60218b25f222b54f77ed38f57aebe08b/Documentation/cpu-freq/governors.txt`.

[7] Y. Gu and S. Chakraborty, "Control Theory-based DVS for Interactive 3D Games," in *Proceedings of the 45th Design Automation Conference*, 2008.

[8] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram, "Frame-Based Dynamic Voltage and Frequency Scaling for a MPEG Decoder," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, 2002.

[9] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards Energy Proportionality for Large-scale Latency-critical Workloads," in *Proceeding of the 41st International Symposium on Computer Architecuture*, 2014.

[10] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Domain Knowledge Based Energy Management in Handhelds," in *Proceedings of the 21st Symposium on High Performance Computer Architecture*, 2015.

[11] Y. Gu and S. Chakraborty, "A Hybrid DVS Scheme for Interactive 3D Games," in *Proceedings of the 14th Real-Time and Embedded Technology and Applications Symposium*, 2008.

[12] Y. Zhu and V. J. Reddi, "High-performance and Energy-efficient Mobile Web Browsing on Big/Little Systems," in *Proceedings of the 19th International Symposium on High-Performance Computer Architecture*, 2013.

[13] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting," in *Proceedings of the 21st Symposium on High Performance Computer Architecture*, 2015.

[14] S. Saha and B. Ravindran, "An Experimental Evaluation of Real-Time DVFS Scheduling Algorithms," in *Proceedings of the 5th International Systems and Storage Conference*, 2012.

[15] M. Digalwar, S. Mohan, and B. K. Raveendran, "Energy Aware Real Time Scheduling Algorithm for Mixed Task Set," in *Proceedings of the International Conference on Advanced Electronic Systems*, 2013.

[16] R. Nassiffe, E. Camponogara, G. Lima, and D. Mossé, "Optimizing QoS in Adaptive Real-Time Systems with Energy Constraint Varying CPU Frequency," in *Proceedings of the III Brazilian Symposium on Computing Systems Engineering*, 2013.

[17] G. Chen, K. Huang, and A. Knoll, "Energy Optimization for Real-time Multiprocessor System-on-chip with Optimal DVFS and DPM Combination," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 3s, 2014.

[18] F. Xie, M. Martonosi, and S. Malik, "Compile-time Dynamic Voltage Scaling Settings: Opportunities and Limits," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[19] Q. Wu, V. J. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D. W. Clark, "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance," in *Proceedings of the 38th International Symposium on Microarchitecture*, 2005.

[20] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic Performance Prediction for Smartphone Applications," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, 2013.

[21] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, no. 3, 1995.

[22] R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, 1996.

[23] M. van der Schee, "2048.c." `https://github.com/mevdschee/2048.c`.

[24] A. Nikolaev, "Curse of War – Real Time Strategy Game For Linux." `https://github.com/a-nikolaev/curseofwar`.

[25] K. Sühring, "H.264/AVC Software Coordingation." `http://iphome.hhi.de/suehring/tml/`.

[26] D. Huggins-Daines, M. Kumar, A. Chan, A. W. Black, M. Ravishankar, and A. I. Rudnicky, "Pocketsphinx: A Free, Real-Time Continuous Speech Recognition System for Hand-Held Devices," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 2006.

[27] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the 4th International Workshop on Workload Characterization*, 2001.

[28] D. Plaetinck, "Uzbl – Web Interface Tools Which Adhere to the Unix Philosophy." `http://www.uzbl.org`.

[29] B. Stabell, K. R. Schouten, B. Gÿsbers, and D. Balaska, "XPilot." `http://www.xpilot.org/`.

[30] "ODROID-XU3." `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127`.

[31] T. N. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu, "Booster: Reactive Core Acceleration for Mitigating the Effects of Process Variation and Application Imbalance in Low-Voltage Chips," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, 2012.

[32] N. Pinckney, M. Fojtik, B. Giridhar, D. Sylvester, and D. Blaauw, "Shortstop: An On-Chip Fast Supply Boosting Technique," in *Proceedings of the 2013 Symposium on VLSI Circuits*, 2013.

[33] W. Godycki, C. Torng, I. Bukreyev, A. Apsel, and C. Batten, "Enabling Realistic Fine-Grain Voltage Scaling with Reconfigurable Power Distribution Networks," in *Proceedings of the 47th International Symposium on Microarchitecture*, 2014.

[34] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems*, 2008.

[35] M. Roitzsch, S. Wächtler, and H. Härtig, "ATLAS: Look-Ahead Scheduling Using Workload Metrics," in *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium*, 2013.