Long-term Workload Phases: Duration Predictions and Applications to DVFS

Canturk Isci[†], Margaret Martonosi[*] and Alper Buyuktosunoglu[†]
[*]Department of Electrical Engineering
Princeton University
mrm@princeton.edu
[†]IBM T.J. Watson Research Center
{cantisci,alperb}@us.ibm.com

Abstract

Computer systems increasingly rely on adaptive dynamic management of their operations in order to balance power and performance goals. Such dynamic adjustments rely heavily on the system's ability to observe and predict workload behavior and system responses. In this paper, we characterize the workload behavior of full benchmarks running on server-class systems using hardware performance counters. Based on these characterizations, we develop a set of long-term value, gradient, and duration prediction techniques that can help systems provision resources. Our best duration prediction scheme is able to predict the duration of program phases ranging from 80ms to over 1 second with greater than 90% accuracy across the SPEC benchmarks.

Applying our long-term predictions in a dynamic voltage and frequency scaling (DVFS) framework, we can identify 92% of the low-energy opportunities found by an oracle with simple predictors. Our most aggressive predictors reduce the number of predictions required by 90%, allowing more system autonomy and requiring less application monitoring and interference.

1 Introduction

Repetitive and recognizable phases in software characteristics have been observed by designers and exploited by computer systems for decades [7]. In recent years, application phase behavior has seen growing interest with two main goals. In the first category, researchers seek to identify program phases from simulation traces [5, 14, 15] or runtime power or performance behavior [6, 12, 16] in order to select representative points within a run to study or simulate. In the second category, the goal is to recognize phase shifts dynamically in running systems in order to perform on-the-fly optimizations [1, 2, 3, 8, 10, 13]. These optimizations include a wide range of possible actions such as voltage/frequency scaling, thermal management, dynamic cache reorganizations, and dynamic compiler optimizations of particular code regions.

In this work, we describe a method for employing predictive on-the-fly program phase tracking in real-systems. We use readings from hardware performance counters to guide our analysis. The phase analysis we perform consists of two key parts. The first aspect is *value prediction* of some metric of interest. This could be a simple metric, such as instructions per cycle (IPC), or it could be a compound metric, composing together several counter values to describe execution (e.g. IPC and L2 Cache Misses). The second aspect of our approach is *duration prediction*. That is, for how long do we expect the value prediction to be valid? This duration prediction is important because it helps the system gauge which sorts of adaptations are feasible. For example, if the duration of a phase is predicted to be quite short, then heavy-handed adaptations like voltage scaling or load balancing may not make sense. In this work, we combine duration and value prediction in our final long-term predictors. We make use of gradient information to predict the long-term behavior of tracked metrics. We test the accuracy and efficiency of these prediction methods on 25 benchmarks from SPEC2000 on a high-end server class processor. These lead to around 80% prediction coverages for stable benchmark regions with on average 5% prediction errors for most benchmarks.

[*]Margaret Martonosi was on sabbatical at IBM T.J. Watson during this work.

2 Overview

The basic dimensions of our prediction framework are long-term metric value and duration prediction. Long-term metric prediction differs from local near-term prediction work such as [9]. We also seek to produce long-term value extrapolations based on the gradient trends we see. For example, where the prior work might guess that upcoming IPC samples will be similar to current ones, our gradient prediction allows us to detect upward or downward trends in a metric, and extrapolate them to predict gradual increases or decreases for longer durations.

The second major dimension of the design is duration prediction. That is, for a given value/gradient trend: how long are we willing to bet on this trend continuing? Duration prediction is useful because it allows one to gauge not just the current system status, but also the length of time one can expect that status to continue. Some system adaptations, such as dynamic voltage/frequency scaling, or OS-level load balancing have sizable performance and energy overheads to overcome before they begin to reap benefits. For such adaptations, one wishes to apply them only when the observed trend is likely to last long enough to overcome any transition costs.

Duration prediction for a stable phase also provides confidence in the persistence of the current behavior into the future. Thus, it reduces the need for an adaptive system to continuously perform checks on the system status at every cycle or polling period to detect any change of behavior.

This is very useful in cases where polling itself has a performance penalty.

Our predictor implementations perform predictions only for application regions identified as *stable*. Stability is decided based on a *stability criterion* and *stability threshold*. Once within a stable region, we decide how long the current phase continues with the specified *variation tolerance* of the tracked metric.

- **Stability threshold** helps us decide whether subsequent samples of the workload are stable in behavior or not. If the comparison between any two samples exceeds this threshold, they are considered unstable. In our case, stability threshold requires samples to be within 0.1 absolute IPC difference.

- **Stability criterion** helps identify regions of stability. In our experiments we require a succession of eight consecutive samples each within the stability threshold of each other to consider stability. Only after this condition is met, our predictors make predictions.

- **Variation tolerance** determines whether the current stable phase of the application continues or a phase transition occurs. At each new prediction point, we compare the current metric value to the prior reading. If they are within the specified variation tolerance, then the current stable phase continues. When a new sample exceeds this tolerance, the duration of the current phase ends and a phase transition occurs. In value and duration prediction, we experiment with various variation tolerances from 1% to 50%.

3 Experimental Setup

All the experiments described were performed on an IBM POWER4^{TM} server platform with the AIX5L for POWER V5.1 operating system. The machine includes a dual-core POWER4 processor. The presented results are per-thread behaviors running in multi-user mode on a lightly-loaded machine. The values collected for these results include both PC samples as well as values read from the POWER4's hardware performance counters, with a sampling tool that works on top of the AIX Performance Monitoring API (PMAPI) [11]. The sampler binds counter behavior to a particular thread, including all library calls and system calls performed by that thread. Sampling frequency is the order of OS switching interval, approximately 10ms.

All the experiments are carried out with the SPECCPU 2000 suite with 25 benchmarks (all except `eon`) and reference datasets. All benchmarks are compiled with XLC and XLF90 compilers with the base compiler flags.

4 Near-Term Metric Value Prediction

Although our goal is to provide long term value predictions, here we present a general approach to predict the value of a tracked particular metric (IPC in this case) for the next sample interval. We call this, *transition-guided metric prediction*, and use the results of this approach as our baseline reference. Later, we develop upon this for long-term predictions and also use the observations here to emphasize the differences between short and long term predictions.

Prior work on short-term prediction has explored a range of prediction schemes for distilling past behavior and using it to create a near-future prediction [9]. These methods have spanned from simple statistical methods such as last-value prediction, and exponentially-weighted moving averages (EWMA) to more elaborate history based and cross metric prediction methods. We focus here on single metric predictions for stable application regions.

Our transition guided predictor implementation performs predictions only at stable application regions based on the described stability criterion. The predictions at these stable regions are performed based on a windowed history, guided by the allowed variation tolerance in the window and maximum window size. The predictor starts with an initial window of size 1. If the current reading is within the variation tolerance, the window size is increased by one element, otherwise window shrinks back to 1. History window can expand up to a maximum of 128 entries, afterward it operates like a FIFO queue. That is, when a new sample arrives into the window, the oldest sample is dropped. Larger window sizes above this offer no advantage in our experiments.

The prediction we make is a simple average of the window contents with uniform weighting. This general predictor encompasses several other more common statistical predictor schemes. For example, if the variation tolerance is set to 0, then the window size is never larger than 1, and we have a last-value predictor. If the variation tolerance is set very large and exponential weighting coefficients are applied, the approach becomes EWMA.

We apply our metric value prediction approach to SPEC workloads for several variation tolerances. A fixed size history window (with 100% variation tolerance) shows the worst behavior with 10% mean absolute error. A 10% variation tolerance results in 2% error, with an average history window size of 48 samples. The last-value predictor (with 0 tolerance) leads to best results with 1% error. Thus, under our stability requirement, a simpler last-value approach performs better than the history based statistical predictors. Moreover, *flat* benchmarks, which show no significant metric variations in their stable phases (such as `art` and `crafty`), show similar prediction accuracies across predictors with different tolerances. On the other hand, benchmarks with observable *gradients* in their stable phases (such as `ammp` and `vortex`) consistently do better with last-value prediction.

These observations offer a foundation for building our long term metric value predictions discussed in Section 6. First, as we only make predictions in stable regions, we avoid the large fluctuations of bursty regions. In the stable regions, the inter-sample variation is relatively slow; thus most samples are quite similar to their predecessor. Sec-

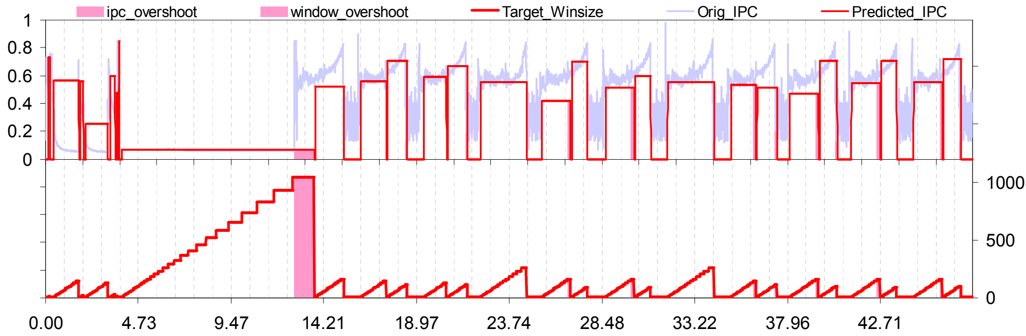

Figure 1. Duration prediction for ammp with FXby8 prediction.

ond, over a long period of stability, the benchmarks can show a trend of increasing or decreasing IPC. In such cases, looking too long back into the history actually worsens the prediction accuracy. For these reasons, our long-term IPC predictions in Section 6 use the last-value, together with the observed inter-sample gradient to predict long durations of IPC behavior.

5 Duration Prediction

As with the near-term value predictions previously discussed, duration prediction boils down to first a decision of *whether* to predict, and second a decision of *what* to predict.

Since our goal is to identify truly long-term program phases suitable for ACPI management, OS load-balancing, and the like, we focus on long-duration predictions (tens of milliseconds or more). Thus, regarding the decision of *whether* to predict, our choice is to avoid duration predictions in periods of instability. We use the same described stability criterion to identify stable regions. Of the measured 25 SPEC benchmarks, 17 spend more than 70% of their runtime in phases that last 200ms to 2 seconds—significantly longer than our stability criterion. Only `equake`, `mgrid` and `bzip2` tend to operate mainly at phase granularities smaller than our stability definition.

The second question for duration prediction is *what* to predict. Here, we discuss several trade-offs, before narrowing in on the possibilities we consider. Duration prediction is distinct from branch outcome prediction or even the value prediction in the previous section, because it has an inequality at the heart of it. That is, the predictor is betting on whether stability will last *at least* N counter samples. For such a prediction, betting $N = 1$ sample is a fairly safe bet, while betting $N = 100,000$ will almost never be correct. The downside to repeatedly betting $N = 1$, however, is that such a short duration may not be long enough to do a major adaptation.

For the results presented here, we consider three simple non-adaptive duration predictors. The first one simply predicts a constant duration. It uses the current and recent counter readings to determine when to predict 8 more samples similar to the current system behavior. In subsequent results, we refer to this predictor as $f(x) = 8$ or *constant8*. This predictor is somewhat conservative, in the sense that some program phases last for seconds (i.e., hundreds of counter samples). For these cases, predicting long phases 8 samples at a time is not as desirable as predicting a long phase with a single aggressive prediction.

In response to the conservatism of this simple constant predictor, we look at two more aggressive predictors. The first of these we refer to as $f(x) = x$ or *FXX*. This predictor counts the number of stable samples it has seen thus far (x), and predicts that the current behavior will continue for at least x more samples into the future. This predictor, thus, behaves as a doubling function. The nice attribute of this approach is that it is relatively cautious for small stable regions, but then grows quickly towards aggressive predictions once longer stability has been shown. The downside to this predictor is that it is prone to significant overshoot when a phase ends.

To try to lessen the overshoot problem, we also look at a third duration prediction function: $f(x) = x/8$, or *FXby8*. This function does not grow as quickly as the x function, but lessens the problems with overshoot, as we show with the following results.

In Figure 1, we show an example of how duration prediction works with the FXby8 dynamic approach on the `ammp` benchmark. In the upper plot, we show the original measured IPC. Superimposed is an IPC value prediction based on last-value prediction that is predicted to be stable for the current duration being predicted. The lower plot shows how the predicted duration grows while FXby8 makes repetitive successful predictions about the current phase. The shaded regions in the lower plot show where the prediction actually performs an overshoot by estimating that the phase will last longer.

While last value prediction is very successful for near-term metric predictions, the flat IPC predictions of Figure 1 show that it is insufficient for long-term prediction. In Section 6, we describe and evaluate a more effective gradient-based method for long-term IPC prediction, where we extrapolate on IPC trends.

5.1 Duration Prediction Evaluation

To evaluate the success of any duration prediction method, we first have to determine metrics for gauging them.

BENCHMARK	DATASET	% STABLE DURATIONS	MEAN STABLE DURATION	ACCURACY			MEAN SAFE PREDN DURATION			PREDICTED STABILITY / TOTAL STABILITY			% RUNTIME IN OVERSHOOT		
				f(x)=8	f(x)=x	f(x)=x/8	f(x)=8	f(x)=x	f(x)=x/8	f(x)=8	f(x)=x	f(x)=x/8	f(x)=8	f(x)=x	f(x)=x/8
ammp	in	82.07	136.73	0.94	0.78	0.96	8.00	34.67	5.43	0.89	0.48	0.85	2.26	24.85	5.74
applu	in	79.99	17.69	0.35	0.32	0.88	8.00	8.10	1.11	0.22	0.16	0.45	20.25	29.07	5.84
apsi	ref	93.84	28.25	0.68	0.58	0.92	8.00	12.34	1.73	0.53	0.34	0.60	14.31	33.09	5.32
art	ref1	91.34	18.71	0.46	0.38	0.88	8.00	10.11	1.37	0.29	0.21	0.49	18.45	29.65	6.84
bzip2	graphic	26.23	9.57	0.07	0.03	0.52	8.00	12.00	1.19	0.02	0.02	0.08	7.08	8.22	1.56
crafty	in	98.98	235.57	0.97	0.74	0.96	8.00	105.74	9.37	0.94	0.49	0.86	1.88	34.39	2.78
equake	in	5.32	44.33	0.78	0.67	0.92	8.00	30.00	2.75	0.63	0.45	0.67	0.50	2.00	0.18
facerec	ref	26.95	67.35	0.87	0.72	0.95	8.00	17.96	2.83	0.77	0.65	0.79	1.54	15.03	1.76
fma3d	ref	88.32	36.48	0.78	0.67	0.92	8.00	18.30	2.65	0.64	0.42	0.67	10.46	32.53	4.86
galgel	in	93.06	33.22	0.73	0.52	0.93	8.00	11.68	1.86	0.57	0.32	0.64	11.62	27.43	5.58
gap	ref	95.58	67.28	0.89	0.70	0.93	8.00	26.77	4.77	0.81	0.45	0.78	6.24	14.55	5.00
gcc	integrate	87.17	39.5	0.78	0.60	0.91	8.00	17.07	2.55	0.62	0.41	0.68	0.90	4.06	0.94
gzip	random	98.4	175.64	0.95	0.73	0.96	8.00	42.25	6.76	0.92	0.49	0.86	2.60	25.73	4.48
lucas	in	96.44	38.25	0.77	0.62	0.93	8.00	15.93	2.04	0.63	0.39	0.70	10.72	38.07	5.70
mcf	inp	99.92	1664.67	1.00	0.90	0.98	8.00	152.94	36.53	0.99	0.52	0.91	0.12	31.33	0.64
mesa	in	97.86	32.18	0.68	0.53	0.92	8.00	11.44	1.71	0.53	0.32	0.64	13.41	32.47	6.42
mgrid	in	0.38	9.5	0.00	0.00	0.60	0.00	0.00	1.00	0.00	0.00	0.16	0.12	0.12	0.04
parser	ref	87.13	23.8	0.64	0.48	0.90	8.00	12.19	1.79	0.45	0.30	0.56	13.12	30.83	5.52
perlbmk	makerand	96.67	145	1.00	1.00	1.00	8.00	30.00	4.44	0.88	0.83	0.83	0.00	0.00	0.00
sixtrack	inp	99.5	226.05	0.97	0.82	0.97	8.00	63.20	8.85	0.94	0.51	0.88	2.00	31.85	5.52
swim	in	93.98	16.37	0.46	0.36	0.85	8.00	8.09	1.26	0.26	0.16	0.41	18.99	22.85	7.30
twolf	ref	99.7	830.5	0.99	0.85	0.98	8.00	97.14	20.42	0.98	0.55	0.89	0.56	12.08	2.14
vortex	bendian3	95.7	106.29	0.93	0.75	0.95	8.00	47.84	5.51	0.87	0.50	0.83	3.94	37.17	4.98
vpr	place	99.94	1665	1.00	0.88	0.98	8.00	203.73	44.69	0.99	0.61	0.94	0.16	12.00	0.60
vpr	route	93.02	26.41	0.66	0.48	0.91	8.00	13.36	1.83	0.49	0.30	0.61	15.05	30.53	5.70
wupwise	ref	93.36	76.49	0.89	0.69	0.95	8.00	17.79	3.42	0.81	0.44	0.79	4.96	14.47	3.46

Table 1. Accuracy, efficiency and overshoot measures for the three duration prediction schemes.

- **Accuracy** or **Safety** define a method's rate of correct predictions *given that it has chosen to make a prediction* [4].

- **Safe Prediction Duration Length** tells how far into the future, on average, a predictor predicts correctly. Although a very *cautious* predictor will have very high accuracy by predicting very small distances into the future, it is equally important to be able to predict long into the future for applicability of the method as well as to reduce the monitoring overhead that accompanies each prediction.

- **Degree of Overshoot** provides a measure of the magnitude of the incorrect duration predictions, as longer predictions have a higher potential to significantly overshoot the actual end of the phase behavior. We present this with the percentage of program runtime spent in overshot predictions.

Table 1 presents these accuracy, efficiency and overshoot results for the three duration predictors discussed. The first two columns show the used benchmarks and their input datasets. Then, the next two columns show each benchmark's *true* stable-phase behavior. One indicates the percentage of application runtime spent in a stable duration. The next gives the mean length (in 10ms samples) of stable durations.

We present *accuracy* as the ratio of safe predictions to the total number of predictions made. *Safe predictions*, is the count of the number of duration predictions in which the stable phase lasted at least as long as predicted.

Overall, FXX shows the worst prediction accuracy, since it tends to overshoot phases often. These overshoots count as "unsafe" predictions. FXby8 shows the best prediction accuracy with an average of 90.6%, because it grows slowly at first and captures short stable regions.

Prediction accuracy is important, but it is only one piece of the puzzle. A second aspect of a predictor is the typical duration it is able to successfully predict. We show this with the *mean safe prediction duration* in Table 1. Predictions that overshot are not included in this average. Next set of columns show what fraction of a benchmark's true stability the predictors were able to successfully capture, i.e. *stable coverage* of a predictor.

By design, the constant8 predictor always has a safe prediction size of 8. FXby8, often makes fairly short predictions, except for applications like `vpr` and `mcf` that have a few very long phases, which allow FXby8 to grow into longer predictions. FXX reaches long intervals more quickly with its aggressive predictions, although this might result in the predictor bypassing some shorter phases via overshoots as in the case of `crafty`, which can impair the effective stable coverage.

In terms of stable coverage, FXby8 predictor is the best for 15 of the 26 benchmarks, offering good predictions for typically 60-94% of a program's stable runtime. For 11 of the benchmarks, however, the constant8 predictor has better coverage than FXby8. These are the cases with longer mean stable phase durations. In these cases, stable coverage of FXby8 drops as we discard the overshot predictions at the end of each phase.

The last figure of merit in designing duration predictors is the degree of overshoot they exhibit. This is given as the *percentage of time spent in overshoot* in Table 1. FXX displays poor performance, with very long overshoots. Between FXby8 and constant8, the distinction is once again more subtle. FXby8 tends to have lower overshoots for the benchmarks that have shorter phases, but it has larger average overshoots for benchmarks with very long mean stable durations—150 samples or more—such as `ammp`, `mcf`, `sixtrack`, `twolf`, and `vpr`. In these cases, the dynamic prediction builds up and overshoots significantly.

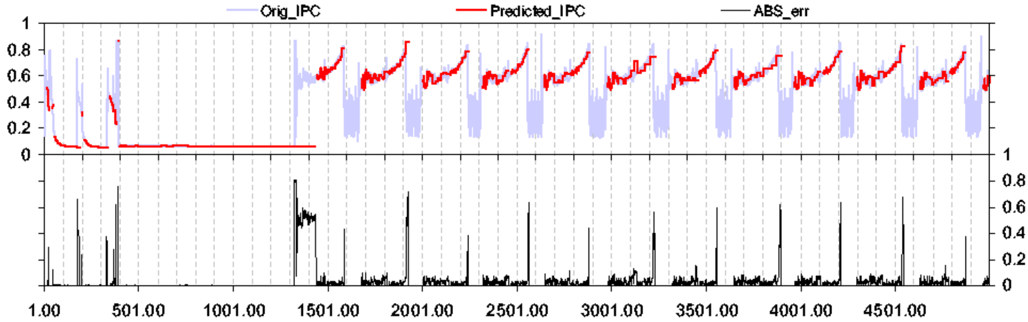

Figure 2. Ammp duration and gradient based metric value prediction for f(x)=x/8.

Overall, duration prediction is a new aspect to phase prediction research with interesting trade-offs. A predictor can be conservative by either guessing infrequently, lowering its stable coverage, or by predicting short intervals, reducing mean predicted safe duration. The FXby8 predictor is better in terms of accuracy, and has low hardware complexity. On the other hand, FXX is dominant in terms of safe prediction durations. In terms of overshoot, constant8 and FXby8 perform better. If a particular application requires very long predictions for its resource planning, then FXX or even FXby8 might be preferable, depending on the trade-off between reaching high predictions quickly and penalty of overshoot. In addition, if the actual cost of monitoring the behavior at each new prediction is significant compared to the penalty of overshoot, then, the more aggressive approach, FXX, can turn out to be appealing with its few prediction checkpoints.

6 Long-term Metric Behavior Prediction

As we have discussed in the previous sections, a simple last-value IPC prediction is good for short-term prediction, but less effective when used in conjunction with duration prediction for longer-term predictions. Here we suggest a simple method to better extrapolate the IPC trend between duration prediction checkpoints. To do this, the value prediction incorporates a *gradient* (slope) by computing the ΔIPC per sampling interval between two prediction checkpoints. With this, it can provide a first-order IPC estimate based on the base IPC and constant gradient, where for each new interval, next predicted IPC equals $current\ prediction + \Delta IPC$. This method relies on the gradients being consistent within predicted durations, which is a reasonable assumption under our stability criterion.

In Figure 2, we show a timeline of duration prediction paired with this value and gradient prediction. We plot the original IPC, the predicted IPC and the difference between them. During stable regions, the results match well. The only points of significant error are where the duration predictor overshoots, leading to higher IPC prediction errors as a phase ends.

We evaluate the accuracy of long-term metric prediction (combined duration and gradient prediction) for the SPEC suite, with the same three prediction functions. Except for the bursty benchmarks `bzip2`, `equake` and `mgrid`, long-term IPC prediction performs quite well. On average the FXby8 achieves an absolute error of 4%. Constant8 and FXX predictors have average errors of 5% and 10% respectively.

7 Applications of Duration Prediction

This section gives an example of a concrete application of duration prediction for energy savings. In particular, we explore its applicability to a dynamic voltage/frequency scaling (DVFS) scenario. For DVFS, the goal is to identify program periods with “slack”, in which slowing down the processor core will save energy with little impact on performance. These periods are typically memory-bound periods in the code [17]. During these periods, we can operate the processor at a lower voltage and frequency, in order to save energy, with little performance impact.

To demonstrate the application of value/duration prediction to DVFS, we focus on a simplified view of the DVFS problem. We consider two modes: *high-energy* mode operates the processor at full performance with full voltage/frequency, and *low-energy* mode operates at low voltage/frequency. Our goal in this work is to correctly predict when to switch to low-energy mode, and gauge how long to remain there before reconsidering a switch back to high-energy mode. We evaluate our success at this goal by considering two conceptual metrics: (i) percentage of time spent in low-energy mode, and its comparison to an oracle, and (ii) number of DVFS switches required, since voltage/frequency adjustments cost both time and energy.

7.1 DVFS Policy

The low-energy setting (slow clock and low voltage) is used for memory-bound portions of the code. The high-energy setting is used at all other times. Our policy is to switch to the low-energy state when we are in a stable phase in which IPC is 25% or less than the maximum IPC value *and* L3 references are greater than 10% of the maximum value. We have looked at several possible metrics to identify slack, such as DTLB and ITLB misses, L3 misses and data table walks. Of them, the rate of L3 references provides the highest confidence (81%) when used with IPC to characterize memory-boundness.

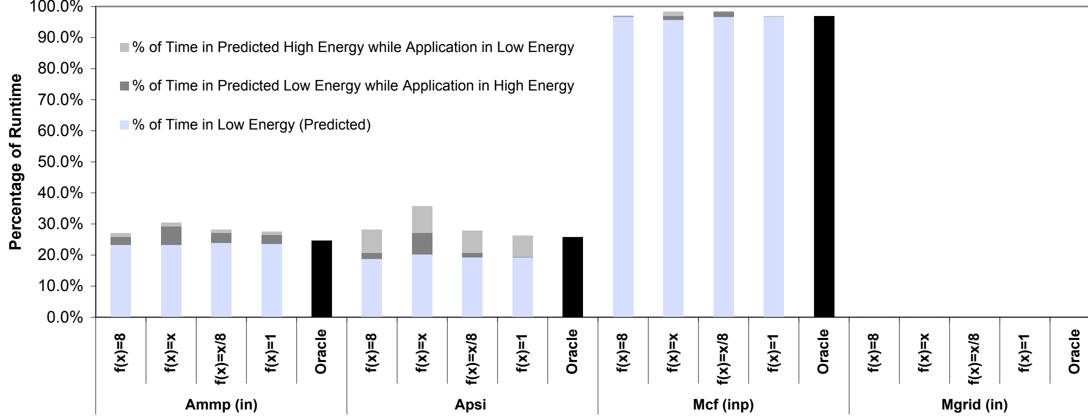

Figure 3. Evaluation of duration and gradient prediction under DVFS.

During stable phases, we can make good predictions about program behavior and apply DVFS accordingly. Duration and gradient prediction is performed as described in the previous section. Layered on top of the long-term metric prediction is the DVFS policy decision. That is, the predictor decides whether to switch to low or high-energy settings for the next predicted duration, based on the long-term IPC and L3 references value predictions.

What remains is to handle unstable regions. At the end of a stable phase, that is when a transition is detected at the new prediction checkpoint, one can either keep the DVFS state as-is, or one can return the DVFS state back to high-energy. Although the former is more efficient in terms of avoiding redundant DVFS costs, it can result in significant performance penalty in quickly-varying benchmarks. For example, `mgrid` has two short stable periods at the benchmark initialization with mean duration of 9.5 samples as we describe in Table 1. The predictor catches one of them and pushes the DVFS state to low at the start of the benchmark. After this point, however, there is only instability for the remainder 99% of the benchmark, and the DVFS state is never returned. As a result, mgrid was incorrectly kept in the low energy state 87% of its runtime. Therefore, we instead use the policy in which unstable regions all revert to the high-energy DVFS state.

7.2 DVFS Results

We show DVFS results for four SPEC benchmarks. The four chosen benchmarks represent different corners of workload behavior. `Ammp` presents a case with repetitive large-scale phases. In contrast, `apsi` has numerous smaller scale phases. `Mgrid` is a very bursty benchmark with almost no stable phases; and `mcf` has a single long stable phase with a significant gradient.

Figure 3 summarizes the effectiveness of DVFS for the four benchmarks, for different duration prediction methods. Again, we show the three predictors: constant8, FXX, FXby8, and a fourth one: $f(x) = 1$ or *constant1*. This fourth predictor predicts every stable sample, and thus never overshoots for more than one sample. We include the constant1 results to demonstrate that our prediction-based techniques achieve results nearly identical (within 1%) to those of a method that monitors counters every sample. The relatively simple predictors we propose offer equal accuracy and a greater degree of autonomy for this sort of system adaptation.

For each application, we show five bars. The rightmost bar shows the runtime spent in low-energy mode with oracle knowledge. The other four bars give the breakdown of DVFS results for different predictors. The lowest portions of the stacked bars show the correctly-predicted low energy regions. Next, we show the percentage of time where our method incorrectly predicts a low-energy region, while in reality the application is in a high-energy region. These represent the prediction overshoots. The top portions represent the opposite: lost opportunity time, where the application is truly in low-energy mode, but our prediction has been for high-energy.

Figure 3 shows, except for `mgrid`, all the benchmarks spend around 20% their stable time in low-energy mode; and our predictors are able to capture most of the available DVFS opportunities. `Mgrid` is too unstable to make DVFS predictions. All of the predictors come within 92% of the oracle approach. `Ammp` and `apsi` with the FXX predictor have a relatively larger incorrect low-energy mode. In `ammp`, this is due to a large overshoot after its huge low-energy phase. In `apsi`, this is due to smaller overshoots accumulated over the large number of small phases. `Apsi` also spends relatively more time in the lost-opportunity mode. This is because of short low-energy regions that are slightly larger than our stability criterion.

Figure 4 shows the number of voltage/frequency adjustments. This is an estimate of DVFS overhead cost. It also gives the number of predictions, as a proxy to the prediction overhead cost. While constant1 makes many predictions and DVFS's, the other predictors are much more stable and lead to significantly less overheads. `Apsi` shows a counter example with FXX, where it leads to more adjustments. Here, predicted gradients in the overshot regions lead to additional false DVFS regions. `Mgrid` and `mcf` show

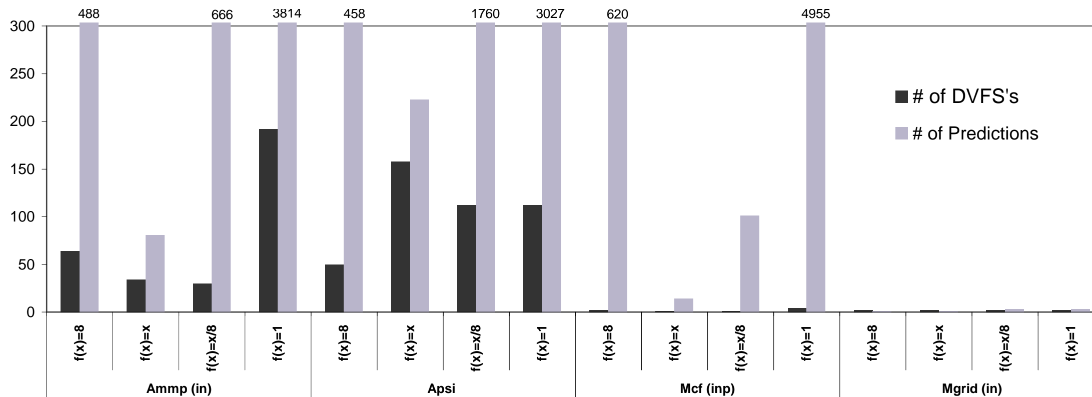

Figure 4. Overheads associated with prediction and DVFS.

very low DVFS transition counts for opposite reasons. In `mcf`, the behavior is very stable, roughly 90% of the time is spent in low-energy mode, and our predictors quickly identify the low-energy opportunity. In `mgrid`, behavior is so unstable that duration prediction does not occur, and thus the program spends all of its time in high-energy mode.

In summary, in all predictable cases the long term predictors perform within 1% of the constant1 case in recovering stable low energy regions. Moreover, unlike the constant1 case that performs continuous monitoring of metrics, they all make substantially less number of predictions, leading to less monitoring overhead.

8 Conclusion

Duration prediction, in conjunction with long-term value prediction is an important area, since many phase directed system level readjustments are only feasible if phases are long enough. In this work we offer first methods for applying duration prediction effectively to energy-saving applications. Our methods achieve prediction accuracies close to 90% of actual stable durations with less than 10% long-term IPC prediction errors in most cases; despite the high variability of phase lengths across the SPEC suite.

With an increasing industry-wide focus on adaptive and autonomous system management, schemes for predicting and responding to very-long-term system behavior become critical for energy efficiency. The work presented here offers practical, low-overhead techniques for such long-range predictions, as well as evaluations of their possible application to important problems in power aware computing.

References

[1] D. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(12):43–51, 2003.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2000.

[3] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03), New Orleans*, Sept. 2003.

[4] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(3):299–316, 2000.

[5] B. Calder, T. Sherwood, E. Perelman, and G. Hamerly. SimPoint web page. http://www.cs.ucsd.edu/simpoint/.

[6] J. Cook, R. L. Oliver, and E. E. Johnson. Examining performance differences in workload execution phases. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[7] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, pages 323–333, May 1968.

[8] A. Dhodapkar and J. Smith. Managing multi-configurable hardware via dynamic working set analysis. In 29th Annual International Symposium on Computer Architecture, 2002.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *IEEE PACT*, pages 220–231, 2003.

[10] C. Hughes, J. Srinivasan, and S. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

[11] IBM. PMAPI structure and function Reference. http://www16.boulder.ibm.com/pseries/en_US/files/aixfiles/pmapi.h.htm.

[12] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-6)*, 2003.

[13] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of Design Automation and Test in Europe, DATE*, Mar. 2001.

[14] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior, 2002. In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[16] R. Todi. Speclite: using representative samples to reduce spec cpu2000 workload. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[17] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002), Grenoble, France,*, Aug. 2002.