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

Computer systems increasingly rely on adaptive dynamic management of their operations 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. The authors characterize the workload behavior of full benchmarks running on server-class systems using hardware performance counters. Based on these characterizations, they developed a set of long-term value, gradient, and duration prediction techniques that can help systems to provision resources.

Canturk Isci
Alper Buyuktosunoglu
IBM T.J. Watson Research Center

Margaret Martonosi
Princeton University

Repetitive and recognizable phases in software characteristics have been observed by designers and exploited by computer systems for decades.[1] Application phase behavior has been the focus of growing interest with two main goals. In the first category, researchers seek to identify program phases from simulation traces,[2-4] runtime power, or performance behavior[5-7] to select representative points within a run to study or simulate. In the second category, the goal is to dynamically recognize phase shifts in running systems to perform on-the-fly optimizations.[8-13] These optimizations include a wide range of possible actions such as voltage or frequency scaling, thermal management, dynamic cache reorganizations, and dynamic compiler optimizations of particular code regions.

We describe a method for employing predictive on-the-fly tracking of program phases in real systems. Readings from hardware performance counters 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, combining several counter values (such as IPC and L2 cache misses) to describe execution.

The second aspect of our approach is *duration prediction,* that is, for how long will the value prediction remain valid? This duration

0272-1732/05/$20.00 © 2005 IEEE Published by the IEEE computer Society

prediction is important because it helps the system gauge which sorts of adaptations are feasible. For example, if the phase has a duration predicted to be quite short, then heavy-handed adaptations like voltage scaling or load balancing might not make sense.

In this work, we combine duration and value prediction in our final long-term predictors. We use gradient information to predict the long-term behavior of tracked metrics, testing 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 percent prediction coverages for stable benchmark regions and an average of 5 percent prediction errors for most benchmarks.

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.[14] We also seek to produce long-term value extrapolations based on gradient trends. For example, where the prior work might guess that upcoming IPC samples will be similar to current ones, our gradient prediction method detects upward or downward trends in a metric and extrapolates 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 lets us gauge not just the current system status, but also the length of time we 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, the goal is 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 based on a stability criterion and stability threshold. Once within a stable region, we decide how long the current phase continues with a specified variation tolerance of the tracked metric.

Stability threshold helps us decide whether subsequent samples of the workload are stable in behavior. If the comparison between any two samples exceeds this threshold, our method considers the sample unstable. In our case, stability threshold requires samples to be within a 0.1 IPC absolute difference.

Stability criterion helps identify regions of stability. In our experiments, we require a succession of eight consecutive samples each within the stability thresholds of the others to consider the current execution region as stable. Only after the region meets this condition do our predictors make predictions. We choose this stability criterion based on the observed phase duration distributions of our experiment's benchmarks, evaluated for the stability threshold. Overall, 70 percent of all observed phases show durations less than eight samples, while constituting less than 5 percent of the actual benchmark execution. With our stability requirement, we avoid unnecessary predictions in these short, bursty regions and focus on the actual, large-scale phase behavior for duration prediction.

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 percent.

Experimental setup

We conducted all our experiments on an IBM POWER4 server platform with the AIX5L for Power, version 5.1 operating system. The machine includes a dual-core

POWER4 processor. The presented results are per-thread behaviors running in multiuser 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).[15] The sampler binds counter behavior to a particular thread, including all library calls and system calls performed by that thread. Sampling period is on the order of OS switching interval, approximately 10 ms. This choice of sampling period is based on the sampling granularity provided by the POWER4 performance monitoring tool.

All the experiments use the SPEC CPU2000 suite with 25 benchmarks (all except eon) and reference data sets. We compiled all the benchmarks with XLC and XLF90 compilers with the base compiler flags.

Short-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 build on this for long-term predictions and 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.[14] 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 makes predictions only at stable application regions identified using the described stability criterion. It makes the predictions 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 size of 1 sample. If the current reading is within the variation tolerance, the predictor increases the window size by one element, otherwise the window shrinks back to 1. A history window can expand up to a maximum of 128 samples; afterward it operates like a FIFO queue. That is, when a new sample arrives in the window, the oldest sample drops out. Larger window sizes offered 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 we set the variation tolerance very high and apply exponential weighting coefficients, the approach becomes an EWMA prediction.

We apply our metric value prediction approach to SPEC workloads for several variation tolerances. A fixed-size history window (with 100 percent variation tolerance) shows the worst behavior with 10 percent mean absolute error. A 10 percent variation tolerance results in 2 percent error, with an average history window size of 48 samples. The last-value predictor (with 0 tolerance) leads to the best results with 1 percent 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. First, because we only make predictions in stable regions, we avoid the large fluctuations of bursty regions. In the stable regions, the intersample variation is relatively slow; thus most samples are quite similar to their predecessor. Second, over a long period of stability, the benchmarks can show a trend of increasing or decreasing IPC. In such cases, looking too far back into the history actually worsens the prediction accuracy. For these rea-

sons, our long-term IPC predictions use the last value together with the observed intersample gradient to predict long durations of IPC behavior.

Duration prediction

As with the short-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 Advanced Configuration and Power Interface (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 stability criterion to identify stable regions. Of 25 SPEC benchmarks, 17 spend more than 70 percent of their runtime in phases that last 200 ms 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 that $N = 100{,}000$ 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 perform a major adaptation.

For the results presented here, we consider three simple nonadaptive duration predictors. The first one simply predicts a constant duration. It uses the current and recent counter readings to determine when to predict eight more samples similar to the current system behavior. We choose eight as the constant prediction duration, so that we predict the stable phase continues at least another stability criterion interval. Thus, the first eight-sample timeout required by our stability criterion triggers the prediction of the next eight samples as the continuing phase duration. This provides a balance between correct predictions and mispredictions that overshoot the whole duration of the current phase. For a stable phase, where we can make at least one correct prediction, the eight-sample predictor covers more than 50 percent of the phase duration, while the final overshoot is less than 50 percent of the total duration. In subsequent results, we refer to this predictor as constant8; $f(x) = 8$.This predictor is somewhat conservative, in the sense that some program phases last for seconds (that is, hundreds of counter samples). For these cases, predicting long phases using eight 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 dynamically growing predictors. The first of these we refer to as FXX; $f(x) = x$. 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 toward aggressive predictions once the region has demonstrated longer stability. The downside to this predictor is that it is prone to significant overshoot when a phase ends.

To lessen the overshoot problem, we also look at a third duration prediction function, which we call FXby8; $f(x) = x/8$. This function does not grow as quickly as the FXX function, but lessens the problems with overshoot, as we show with the following results. We choose FXby8 as our third duration predictor example, because for this set of dynamically growing predictors, FXby8 provides the most timid behavior. Based on our stability criterion of eight samples, FXby8 initially sees eight stable samples and predicts one more sample of stability. Thus, it starts predictions very cautiously and grows to predict further into the future for long stable phases.

In Figure 1, we show an example of how duration prediction works with the FXby8 dynamic approach on the ammp benchmark. For all plotted traces, the x axis shows the execution timeline in seconds. In the Figure 1a,

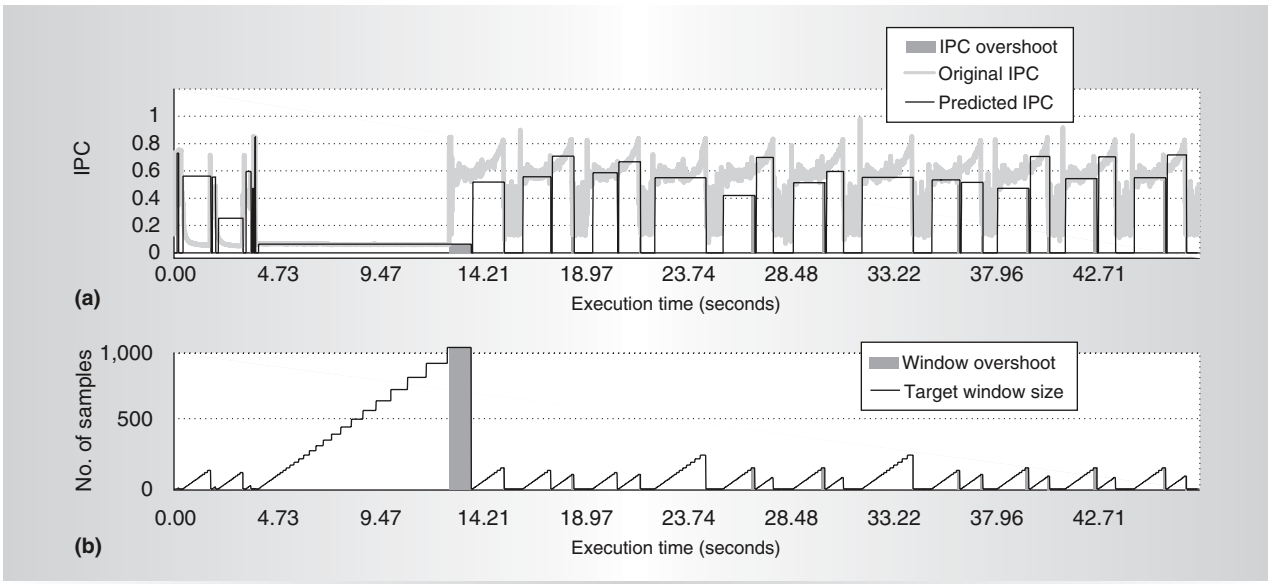


Figure 1. Duration prediction for ammp with FXby8: original and predicted IPC (a) and target window size (b).

we show the original measured IPC and superimpose an IPC value prediction based on last-value predictions for each stable phase. Figure 1b 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.

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

Duration prediction evaluation

To evaluate the success of any duration prediction method, we first have to define the following metrics for gauging them:

- *Accuracy* defines a method's rate of correct predictions given that it has chosen to make a prediction.[16] We present accuracy as the ratio of safe predictions to the total number of predictions made. We define safe predictions as the number of duration predictions in which the stable phase lasted at least as long as predicted.
- *Mean safe prediction duration* tells how far into the future, on average, a predictor makes predictions 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 further into the future for applicability of the method as well as to reduce the monitoring overhead that accompanies each prediction.
- *Overshoot runtime* measures the magnitude of incorrect duration predictions; 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.

Tables 1 to 3 present these accuracy, mean safe prediction duration, and overshoot results for the three duration predictors discussed. The first two columns in the tables show the benchmarks and their input data sets. In Table 1, 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 10 ms samples) of stable durations.

Overall, FXX shows the worst prediction accuracy, since it often tends to overshoot phases. These overshoots count as unsafe predictions. FXby8 shows the best prediction

Table 1. Accuracy for the three duration prediction schemes.

Benchmark	Data set	Stable durations (percentage)	Mean stable duration (no. of 10-ms samples)	Accuracy		
				Constant8	FXX	FXby8
ammp	in	82.07	136.73	0.94	0.78	0.96
applu	in	79.99	17.69	0.35	0.32	0.88
apsi	ref	93.84	28.25	0.68	0.58	0.92
art	ref1	91.34	18.71	0.46	0.38	0.88
bzip2	graphic	26.23	9.57	0.07	0.03	0.52
crafty	in	98.98	235.57	0.97	0.74	0.96
equake	in	5.32	44.33	0.78	0.67	0.92
facerec	ref	26.95	67.35	0.87	0.72	0.95
fma3d	ref	88.32	36.48	0.78	0.67	0.92
galgel	in	93.06	33.22	0.73	0.52	0.93
gap	ref	95.58	67.28	0.89	0.70	0.93
gcc	integrate	87.17	39.5	0.78	0.60	0.91
gzip	random	98.4	175.64	0.95	0.73	0.96
lucas	in	96.44	38.25	0.77	0.62	0.93
mcf	inp	99.92	1664.67	1.00	0.90	0.98
mesa	in	97.86	32.18	0.68	0.53	0.92
mgrid	in	0.38	9.5	0.00	0.00	0.60
parser	ref	87.13	23.8	0.64	0.48	0.90
perlbmk	makerand	96.67	145	1.00	1.00	1.00
sixtrack	inp	99.5	226.05	0.97	0.82	0.97
swim	in	93.98	16.37	0.46	0.36	0.85
twolf	ref	99.7	830.5	0.99	0.85	0.98
vortex	bendian3	95.7	106.29	0.93	0.75	0.95
vpr	place	99.94	1665	1.00	0.88	0.98
vpr	route	93.02	26.41	0.66	0.48	0.91
wupwise	ref	93.36	76.49	0.89	0.69	0.95

accuracy with an average of 90.6 percent 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 2. This mean does not include predictions that overshot. The next set of columns show what fraction of a benchmark's true stability the predictors successfully captured, representing the *stable coverage* of the three predictors.

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, the FXby8 predictor is the best for 15 of the 26 benchmarks, offering good predictions for typically 60 to 94 percent 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; we show this metric in Table 3. FXX displays poor performance, with very long

Table 2. Safe prediction durations and stability for the three duration prediction schemes.

		Mean safe prediction duration		Predicted stability/total stability		
Benchmark	Data set	FXX	FXby8	Constant8	FXX	FXby8
ammp	in	34.67	5.43	0.89	0.48	0.85
applu	in	8.10	1.11	0.22	0.16	0.45
apsi	ref	12.34	1.73	0.53	0.34	0.60
art	ref1	10.11	1.37	0.29	0.21	0.49
bzip2	graphic	12.00	1.19	0.02	0.02	0.08
crafty	in	105.74	9.37	0.94	0.49	0.86
equake	in	30.00	2.75	0.63	0.45	0.67
facerec	ref	17.96	2.83	0.77	0.65	0.79
fma3d	ref	18.30	2.65	0.64	0.42	0.67
galgel	in	11.68	1.86	0.57	0.32	0.64
gap	ref	26.77	4.77	0.81	0.45	0.78
gcc	integrate	17.07	2.55	0.62	0.41	0.68
gzip	random	42.25	6.76	0.92	0.49	0.86
lucas	in	15.93	2.04	0.63	0.39	0.70
mcf	inp	152.94	36.53	0.99	0.52	0.91
mesa	in	11.44	1.71	0.53	0.32	0.64
mgrid	in	0.00	1.00	0.00	0.00	0.16
parser	ref	12.19	1.79	0.45	0.30	0.56
perlbmk	makerand	30.00	4.44	0.88	0.83	0.83
sixtrack	inp	63.20	8.85	0.94	0.51	0.88
swim	in	8.09	1.26	0.26	0.16	0.41
twolf	ref	97.14	20.42	0.98	0.55	0.89
vortex	bendian3	47.84	5.51	0.87	0.50	0.83
vpr	place	203.73	44.69	0.99	0.61	0.94
vpr	route	13.36	1.83	0.49	0.30	0.61
wupwise	ref	17.79	3.42	0.81	0.44	0.79

* Except for mgrid, constant8 has a mean safe prediction duration of 8 samples.

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.

Overall, duration prediction is a new aspect of phase prediction research that has interesting trade-offs. A predictor can be conservative by either guessing infrequently, lowering its stable coverage, or by predicting short intervals, reducing the mean safe prediction durations. The FXby8 predictor is more accurate 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.

The trade-offs that guide the choice of predictors are clear, when comparing FXX and the other two predictors. 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 the overshoot penalty. In addition, if the actual cost of monitoring the behavior at each new prediction is significant—such as reading external power measurement logs or thermal sensors—compared to the overshoot penalty, the more aggressive approach, FXX, turns out to be appealing with its few prediction checkpoints. On the other hand, for lightweight applications with minimal monitoring overhead, more timid predictors con-

Table 3. Overshoot for the three duration prediction schemes.

		Overshoot runtime (percentage)		
Benchmark	**Data set**	**Constant8**	**FXX**	**FXby8**
ammp	in	2.26	24.85	5.74
applu	in	20.25	29.07	5.84
apsi	ref	14.31	33.09	5.32
art	ref1	18.45	29.65	6.84
bzip2	graphic	7.08	8.22	1.56
crafty	in	1.88	34.39	2.78
equake	in	0.50	2.00	0.18
facerec	ref	1.54	15.03	1.76
fma3d	ref	10.46	32.53	4.86
galgel	in	11.62	27.43	5.58
gap	ref	6.24	14.55	5.00
gcc	integrate	0.90	4.06	0.94
gzip	random	2.60	25.73	4.48
lucas	in	10.72	38.07	5.70
mcf	inp	0.12	31.33	0.64
mesa	in	13.41	32.47	6.42
mgrid	in	0.12	0.12	0.04
parser	ref	13.12	30.83	5.52
perlbmk	makerand	0.00	0.00	0.00
sixtrack	inp	2.00	31.85	5.52
swim	in	18.99	22.85	7.30
twolf	ref	0.56	12.08	2.14
vortex	bendian3	3.94	37.17	4.98
vpr	place	0.16	12.00	0.60
vpr	route	15.05	30.53	5.70
wupwise	ref	4.96	14.47	3.46

stant8 and FXby8 significantly improve stable coverage. Additionally, in cases where large overshoots lead to undesired penalties or affect system reliability—such as by causing thermal emergencies under thermal management—FXX (or even FXby8) can deteriorate system performance. In these cases, the constant predictor can be preferable for safe operation under dynamic management.

The trade-offs between FXby8 and constant8 are more subtle and their relations are workload dependent. For the benchmarks we investigated, the ones with mean stable durations of less than 400 ms (40 samples) have better stable coverage and fewer overshoots with FXby8. On the other hand, constant8 predicts approximately 4× longer durations. For mean stable durations between 400 ms to 800 ms, both benchmarks behave similarly in terms of coverage and overshoot. For these cases, constant8 performs better in terms of predicted duration lengths.

For mean stable durations of 800 ms to 2 s, constant8 performs marginally better in terms of coverage, overshoot, and durations. Above 2 seconds, FXby8 demonstrates its ability to predict efficiently for long durations, leading to approximately 4× longer predicted durations than constant8. However, it leads to relatively larger overshoots and lower coverages than constant8, although most such metrics typically have reasonable margins. Thus, in summary, similar trade-offs between stable coverage, number of predictions, prediction lengths, and overshoots, as discussed earlier, guide the choice of predictors between constant8 and FXby8. However, the same trade-offs necessitate different predictors under different workload behavior. Although one set of trade-offs favor FXby8 for benchmarks with comparatively shorter—that is, 400 ms or less—phase durations, the same trade-offs favor constant8 for applications with longer—that is, over 2 second—phases. Consequently, for a dynamic implementation, it is more beneficial to implement a predictor that performs either FXby8 or constant8 prediction, where a separate configurable control logic can choose the right method at runtime based on observed workload phase granularities.

Long-term metric behavior prediction

As we have discussed, 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, a predictor can provide a first-order IPC estimate based on the base IPC and constant gradient and, for each new interval, next-predicted IPC equals the current prediction plus Δ*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 and predict-

Figure 2. Duration prediction (a) and gradient-based metric value (b) for ammp using FXby8.

ed IPCs, 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, FXby8 achieves an absolute error of 4 percent. Constant8 and FXX predictors have average errors of 5 and 10 percent.

Applications of duration prediction

Now we give 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—areas in which slowing down the processor core will save energy with little impact on performance. These periods are typically memory-bound regions in the code.[17] During these periods, the processor can operate at a lower voltage and frequency to save energy; these changes will have little performance impact.

To demonstrate the application of value and 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 and frequency. *Low-energy* mode operates at low voltage and frequency. Our goal in this work is to correctly predict when to switch to low-energy mode and to 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: percentage of time spent in low-energy mode and its comparison to an oracle; and number of DVFS switches required, since voltage and frequency adjustments cost both time and energy.

DVFS policy

We use the low-energy setting (slow clock and low voltage) for memory-bound portions of the code and the high-energy setting 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 percent or less of the maximum IPC value *and* L3 references are greater than 10 percent of the maximum value. We have looked at several possible metrics to identify slack, such as data and instruction translation look-aside buffer misses, L3 misses, and data table walks. Of them, the rate of L3 references provides the highest confidence (81 percent) when used with IPC to characterize memory-bound phases.

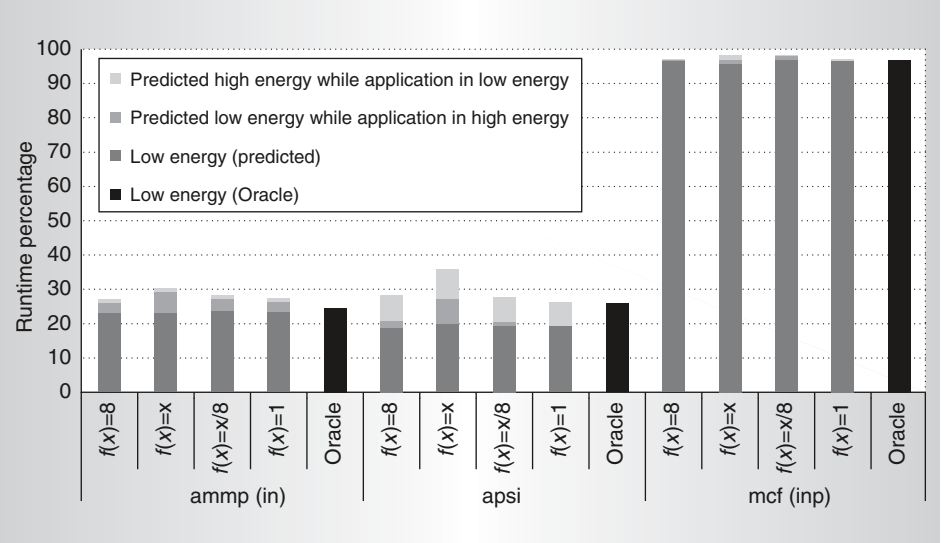


Figure 3. Evaluation of duration and gradient prediction under DVFS.

During stable phases, we can make accurate predictions about program behavior and apply DVFS accordingly. We performed duration and gradient prediction 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 the predictor detects a transition at the new prediction checkpoint—you can either keep the DVFS state as is or return it 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 benchmark initialization; with mean stable 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 percent of the benchmark, and the DVFS state is never returned. As a result, mgrid was incorrectly kept in the low-energy state 87 percent of its runtime. Therefore, we instead use the policy in which unstable regions all revert to the high-energy DVFS state.

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 three benchmarks, for different duration prediction methods. Mgrid is excluded as all five bars are approximately zero for mgrid. Again, we show the three predictors: constant8, FXX, FXby8, and a fourth one: constant1, $f(x) = 1$. 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 percent) 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 pre-

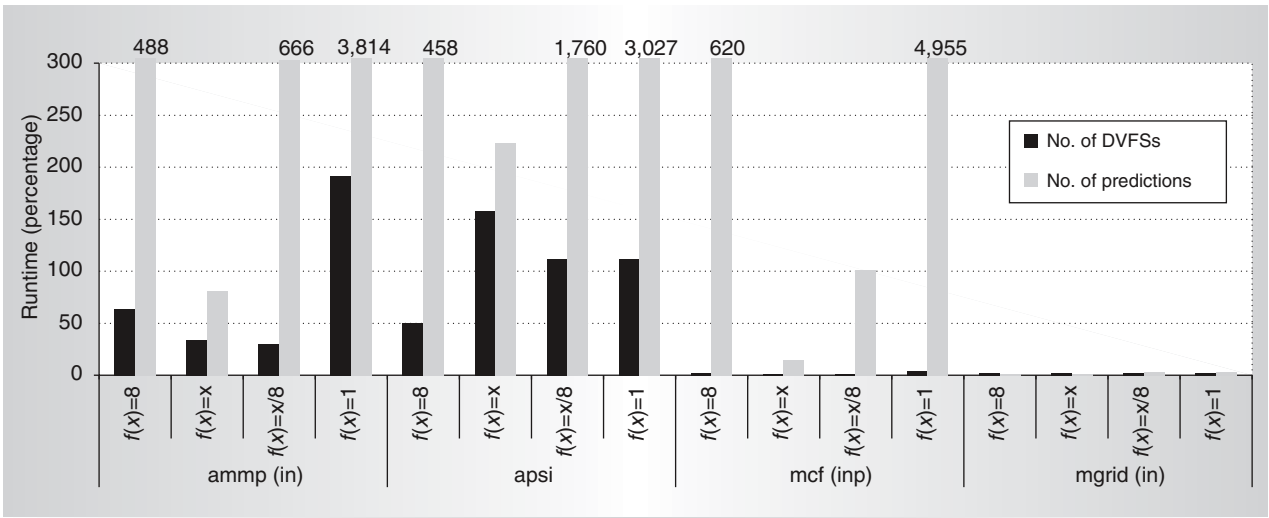


Figure 4. Overheads associated with prediction and DVFS.

dicted 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 that, except for mgrid, all the benchmarks spend around 20 percent of 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 percent of the oracle approach. Ammp and apsi with the FXX predictor have a relatively larger incorrect low-energy mode. In ammp, this is because of a large overshoot after its huge low-energy phase. In apsi, smaller overshoots accumulated over the many small phases. Apsi also spends relatively more time in 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 and frequency adjustments, which gives an estimate of DVFS overhead cost. It also gives the number of predictions, as a proxy to the prediction overhead cost. Although constant1 makes many predictions and frequency and voltage adjustments, the other predictors are much more stable and lead to significantly less overhead. 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 very low DVFS transition counts for opposite reasons. In mcf, the behavior is very stable, roughly 90 percent 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.

Comparing our three predictors, the difference between FXX and the other two is again easily observable. FXX leads to higher overshoots than the others, resulting in performance penalties from keeping the applications incorrectly in low-energy mode and by the spurious voltage and frequency scalings in these overshoot regions. Although FXX performs substantially fewer predictions, it is not the most suitable method under this DVFS scenario, because the cost of DVFS transitions is, in general, significantly higher than those for monitoring performance behavior. Constant8 and FXby8 perform similarly in identifying true low-energy regions and lost DVFS opportunities. Similar to our previous discussion, the choice of predictor between FXby8 and constant8 is workload dependent. For shorter phases, as in the case of apsi, constant8 performs better, having fewer predictions and voltage or

frequency scalings. For longer phases, FXby8 becomes the better alternative, having either fewer scalings or fewer predictions. Therefore, the preferred predictor implementation for such a DVFS scenario requires a dual-mode implementation, which chooses between FXby8 and constant8 based on observed low-energy phase durations.

In summary, in all predictable cases the long-term predictors perform within 1 percent of the constant1 case in recovering stable low-energy regions. Moreover, unlike the constant1 case that continuously monitors metrics, our predictors all make substantially fewer predictions, leading to a lower monitoring overhead. For the demonstrated case, FXby8 performs better for longer phase durations, while constant8 is preferable for shorter phases. Consequently, the optimal solution lies in the combination of the two approaches.

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 first offer methods for effectively applying duration prediction to energy-saving applications. Our methods achieve prediction accuracies close to 90 percent of the actual stable durations with, for most cases, less than a 10 percent error in long-term IPC prediction. Such accuracy is possible 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 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. MICRO

Acknowledgments

This work was done while Canturk Isci was a summer intern and Margaret Martonosi was on sabbatical at IBM T.J. Watson Research Center. The authors wish to thank Pradip Bose, Chen-Yong Cher, and Prabhakar Kudva for many interesting discussions, and Calin Cascaval for his help with the performance monitoring toolset.

References

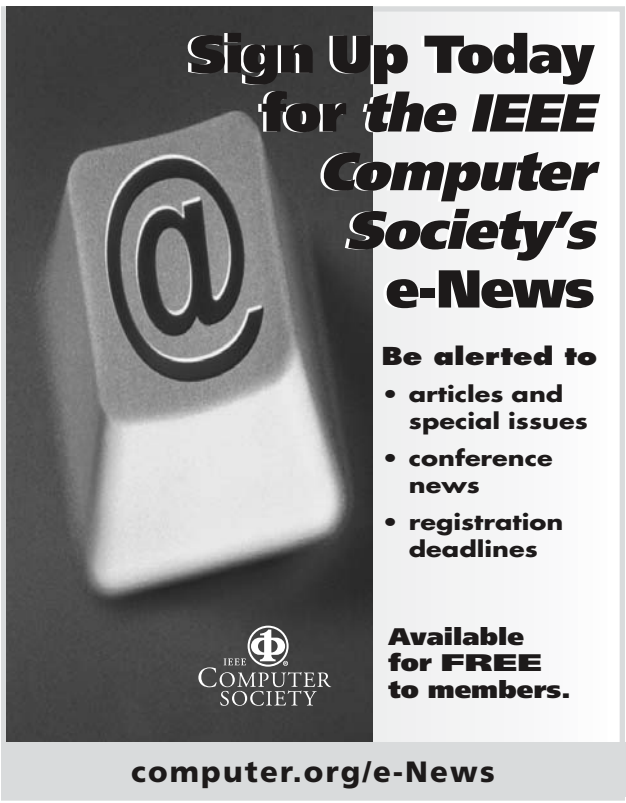
1. P.J. Denning, "The Working Set Model for Program Behavior," *Comm. ACM*, vol. 11, no. 5, May 1968, ACM Press, pp. 323-333.
2. B. Calder et al., "SimPoint 3.0: Faster and More Flexible Program Analysis," http://www.cse.ucsd.edu/~calder/papers/MOBS-05-SimPoint3.pdf.
3. T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques* (PACT 01), IEEE CS Press, 2001, pp. 3-14.
4. T. Sherwood et al., "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS X), ACM Press, 2002, pp. 45-57.
5. J. Cook, R.L. Oliver, and E.E. Johnson, "Examining Performance Differences in Workload Execution Phases," *Proc. IEEE Int'l Workshop Workload Characterization* (WWC-4), IEEE Press. 2001, pp. 82-90.
6. C. Isci and M. Martonosi, "Identifying Program Power Phase Behavior Using Power Vectors," *Proc. IEEE Int'l Workshop Workload Characterization* (WWC-6), IEEE Press, 2003, pp. 108-118.
7. R. Todi, "Speclite: Using Representative Samples to Reduce Spec cpu2000 Workload," *Proc. IEEE Int'l Workshop Workload Characterization* (WWC-4), IEEE Press, 2001, pp. 15-23.
8. D. Albonesi et al., "Dynamically Tuning Processor Resources with Adaptive Processing," *IEEE Computer,* vol. 36, no. 12, Dec. 2003, pp. 49-58.
9. V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 2000), ACM Press, 2000, pp. 1-12.
10. F. Bellosa et al., "Event-Driven Energy Accounting for Dynamic Thermal Management," *Proc. Workshop on Compilers and Operating Systems for Low Power* (COLP 03), 2003, pp. 1-10.
11. A. Dhodapkar and J. Smith, "Managing Multi-Configurable Hardware Via Dynamic Working Set Analysis," *Proc. 29th Ann. Int'l*

Symp. Computer Architecture (ISCA 29), IEEE CS Press, 2002, pp. 233-246.
12. C. Hughes, J. Srinivasan, and S. Adve, "Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications," *Proc. 34th Ann. Int'l Symp. on Microarchitecture* (Micro-34), IEEE CS Press, 2001, pp. 250-261.
13. A. Iyer and D. Marculescu, "Power Aware Microarchitecture Resource Scaling," *Proc. Design Automation and Test in Europe* (DATE 01), IEEE CS Press, 2001, pp. 190-196.
14. E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and Predicting Program Behavior and its Variability," *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques* (PACT 03), IEEE CS Press, 2003, pp. 220-231.
15. IBM, "PMAPI Structure and Function Reference," http://www16.boulder.ibm.com/pseries/en_US/files/aixfiles/pmapi.h.htm.
16. L. Benini, A. Bogliolo, and G.D. Micheli, "A Survey of Design Techniques for System-Level Dynamic POWER Management," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 8, no. 3, 2000, pp. 299-316.
17. A. Weissel and F. Bellosa, "Process Cruise Control: Event-Driven Clock Scaling for Dynamic POWER Management," *Proc. Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems* (CASES 02), ACM Press, 2002, pp. 238-246.

Canturk Isci is a PhD candidate in the electrical engineering department at Princeton University. He was an intern at IBM T.J. Watson Research Center in summer of 2005, and summer and fall of 2004. His research interests include power-aware computing and workload characterizations for dynamic management. Isci has an MSc in VLSI system design from University of Westminster, London, an MA in electrical engineering from Princeton University, and a BS in electrical engineering from Bilkent University, Ankara, Turkey. He is a member of the IEEE.

Alper Buyuktosunoglu is a research staff member at the IBM T.J. Watson Research Center. His research interests include high-performance, low-power computer architectures and digital microelectronic design. Buyuktosunoglu has a PhD and an MS in electrical and computer engineering from the University of Rochester and a BS in electrical engineering from Middle East Technical University, Ankara, Turkey. He is a member of the IEEE.

Margaret Martonosi is a professor of electrical engineering at Princeton University. Her research interests include computer architecture and the hardware-software interface, with particular focus on power-efficient systems and mobile computing. Martonosi has a PhD and an MS in electrical engineering from Stanford University and a BS in electrical engineering from Cornell University. She is a senior member of IEEE and a member of ACM, where she is vice-chair of ACM SIGARCH and recently served two terms on the board of directors for ACM SIGMETRICS.

Direct questions and comments about this article to Alper Buyuktosunoglu, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598; alperb@us.ibm.com.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.