RC23448 (W0411-181) November 19, 2004
Computer Science

IBM Research Report

Dynamic Adaptation in Server-Class Microprocessors: Workload Phase and Duration Predictions with Live Counter Measurements

Canturk Isci, Margaret Martonosi, Alper Buyuktosunoglu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. Ithas been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distributionoutside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g, payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

Dynamic Adaptation in Server-Class Microprocessors: Workload Phase and Duration Predictions with Live Counter Measurements

Canturk Isci, Margaret Martonosi, Alper Buyuktosunoglu
IBM T. J. Watson Research Center

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. While previous application phase analysis has focused on phase behavior at the granularity of small snippets of millions of instructions, our work here looks at coarse-grained workload phases on the order of tens to hundreds of milliseconds.

In this paper, we characterize the workload behavior of full benchmarks running on server-class systems using hardware performance counter measurements. Based on these characterizations, we develop a set of 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. All of our results are measured on live systems in multi-user mode, and thus demonstrate the ability of our predictors to function in the face of real-system variations. These phase prediction techniques can be applied to a range of usages including thread scheduling, planning dynamic voltage and frequency scaling (DVS), and managing system load balancing. In particular, our results demonstrate applying our prediction techniques to DVS. Our simple predictors identify 92% of the low-energy opportunities found by an oracle. Our most aggressive predictors do similarly well, and reduce the number of predictions required, 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 [10]. 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, 21, 22] or runtime power or performance behavior [8, 17, 24] 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, 11, 15, 19]. 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.

Most recent phase analysis work has focused on the first category of uses. That is, researchers using full-length profiling runs to analyze an application and select "representative" snippets from its execution. Simulation studies can then focus on behavior within these chosen intervals, with some assurance that they can extrapolate out to whole-program characteristics of interest.

Such phase analysis benefits from two key details. First, a full-length profiling run gives the phase analysis a comprehensive and forward-looking view of application behavior that is not available to *reactive* on-the-fly phase predictors. Second, much of the existing phase analysis work is based on simulations, rather than real-system measurements. As such, they see very little time and space variability between software runs due to multiprogramming, system call variations, and non-repeatable effects like interrupts.

In this work, we describe a method for employing reactive and predictive on-the-fly program phase analysis in real-life 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, 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, while not previously studied, 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. We propose a step-by-step method for composing these techniques into an application phase recognition system, and we test it on 25 benchmarks from SPEC2000.

The primary contributions of this work are as follows. First, our work characterizes phase behavior of a widely-varied set of benchmarks running on a high-end server system. Second, based on this, we choose a measurement and prediction technique that offers good accuracy and coverage across widely-varying application performance. Third, we offer the first-known duration predictor for on-the-fly metric prediction and evaluate its success.

The remainder of this paper is structured as follows. Section 2 describes our measurement setup and methodology, and presents our power phase analysis approach. Section 3 describes the issues inherent in developing value prediction methods, presents our techniques, and discusses their accuracy. Section 4 then introduces duration prediction as a new aspect of phase prediction, and gives characterizations and results for several duration predictors we have analyzed. In Section 5, we discuss the application of these predictors to a particular problem, planning support for dynamic voltage/frequency scaling. Section 6 discusses related work,

and Section 7 offers our conclusions.

2 Phase Characterization and Prediction: Overview

As mentioned, the basic dimensions of our phase prediction method include *long-term metric value prediction* and *duration prediction*. In addition to these key dimensions, there are a number of other design issues required for on-the-fly phase detection and adaptation; we discuss them here.

Consider Figure 1, which depicts a timeline of how instructions per cycle varies over time for the SPEC benchmark ammp. (As will be discussed in more detail in Section 2.3, the data are collected by reading hardware performance counters at 10ms time granularity on an IBM POWER4^{TM} based server running AIX.) Like most benchmarks, ammp shows strong variations in its IPC levels over time. The overall goal of our research is to explore on-the-fly methods for predicting what values to expect, and how long they are expected to last.

2.1 Trade-offs

Breaking the phase prediction problem into sub-parts allows us to evaluate our progress towards each somewhat independently. Metric value prediction requires us to make fairly local predictions about what a particular metric's value will be in the near future. This work follows on from the local phase prediction work such as that presented in [12]. Moving beyond that work, however, 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 portion of the design is duration prediction. This sub-problem says, for a given value/gradient trend: how long are we willing to bet on this trend continuing? For example, if metric prediction is simply guessing that IPC on the next sample point will be a particular value, duration prediction is the problem of predicting for how many sample points we can expect IPC to be that value.

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 can have fairly sizeable performance and energy overheads to overcome before they begin to reap benefits. For such "heavyweight" adaptations, one cannot apply them at each fluctuation in workload. One instead wishes to apply them only when the observed trend is likely to last long enough to overcome any transition costs. Thus if duration prediction can accurately predict long-periods of low IPC behavior, then OS load rebalancing may be warranted.

From a different perspective, 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 confidence is very useful in cases where the polling itself has a performance penalty, while most of the time the current behavior persists.

With any prediction, issues of accuracy and coverage arise. As we will show in upcoming sections, our goal with both metric value and duration prediction is to have quite high accuracy, with fairly good coverage. We will quantify this as we proceed. In essence, we want the predictions to be right most of the time when they are made, and we want predictions to be made enough of the time to warrant the system support they are given.

2.2 Measurement Infrastructure Support

Having sketched out the rough prediction goals of our work, we now discuss the infrastructure support issues that impact achieving them. These issues include the types of hardware metrics that can be observed, and the overhead incurred in observing them.

Prior work on monitoring has explored a range of possible infrastructures, from software instrumentation, to periodic program counter sampling, to extensive hardware counter support [6, 13, 14, 18, 20, 26, 27]. The choice of metrics to predict is inextricably intertwined with the measurement infrastructure. High-overhead infrastructure, like hardware performance counters, cannot be read on every basic block, nor is it accurate to use coarse-grained scheduler quanta behavior to deduce cycle-level details.

While prior work on phases has focused primarily on fine-grained low-level behavior, deduced via basic-block-level instrumentation and sampling, our work seeks to explore coarse-grained phases. The phases we observe and predict last for milliseconds or seconds of execution. Our methods for collecting data are described in the next subsection.

2.3 Measurement Methodology

All the experiments described here were performed on an IBM POWER4 server platform with the AIX5L for POWER V5.1 operating system. The machine includes a dual-core POWER4 processor. The results presented here 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) [16]. The sampler binds counter behavior to a particular thread, including all library calls and system calls performed by that thread.

In order to minimize the counter-reading overhead, we only examine hardware counters during the context switch interval. Thus our readings are on the 10ms granularity that is the OS switching interval for this system. Our analyses are thus geared to the coarser-grained intervals we see at this granularity.

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.

3 Metric Value Prediction

The first sub-problem we consider is metric value prediction. Our goal is to produce a simple-yet-effective approach that tracks a given metric with good accuracy across a range of workloads. We focus here on a single metric: committed instructions per cycle, although past work has shown that phase behavior across multiple metrics is also visible and useful [12, 17].

Prior work on IPC prediction has explored a range of prediction schemes for distilling past behavior and using it to create a near-future prediction. 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.

In our work, IPC prediction is a starting point for the main focus of stable-phase duration prediction. As such, our primary goal is to predict IPC during regions of relative stability. To accomplish this, we use a scheme with two main components. It has to identify stable versus unstable regions, and within stable regions, it has to choose the window size of points to consider.

We have done fairly extensive studies regarding how to define stability. Due to space constraints in this paper, we focus solely on a simple stability criterion. Namely, we require a succession of eight consecutive samples each within a *stability threshold* of each other before we attempt to predict any behavior. In our case, the stability threshold requires adjacent points to be within 0.1 difference in absolute IPC value of each other. For some benchmarks, as we will show, this stability requirement can reduce the prediction coverage, i.e., the fraction of the application we predict for. It typically improves, however, the prediction accuracy which we value more.

Once within a stable region, we also choose a window of points to consider. Our method for this starts with an averaging window of size 1. As we encounter a sequence of readings about current IPC, we first compare the current counter reading to the most recent prior counter reading. If it is within an *error tolerance* of the prior reading, then the window size is increased by one element, up to a maximum window size of 128. (We experimented with larger maximum window sizes, but found that they offered little further information beyond the 128-entry approach.) If the window size is already at its maximum value of 128 and the readings continue to be stable, the window remains at the maximum of 128 entries, and slides along the timeline with the new entries.

The prediction we make is a simple average of the window contents, although more elaborate weighting schemes or table-lookups are also possible. Another possible variant is to use this method, but insist that IPC predictions are only made when stable, i.e., the last and the current reading differ by less than the stability threshold.

Enlarging the error tolerance increases the likelihood that more fluctuations will be present within the window, leading to higher error rates on the predictions. Conversely, tightening the error tolerance means that the IPC readings in the window are more tightly clustered, and therefore the average error in their predictions also tends to be smaller.

Depending on the setup of the approach, this general predictor encompasses several other more common statistical predictor schemes. For example, if the error tolerance is set to 0, then the window size is never larger than 1, and we have a last-value predictor. If the error tolerance is set very large and weighting coefficients are applied, the approach becomes EWMA.

3.1 IPC Value Prediction Results

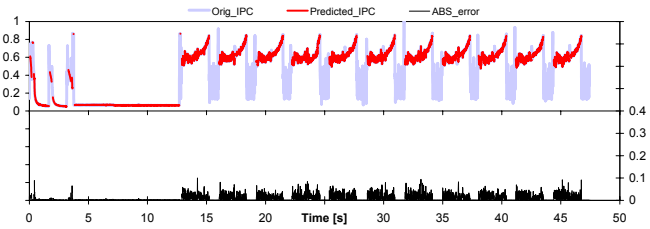
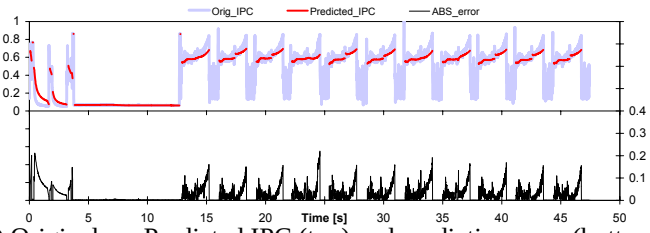
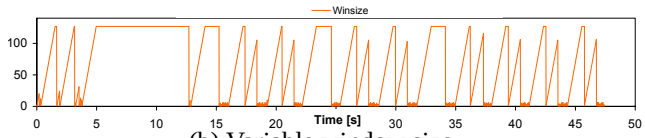


Figure 1. IPC value prediction with 0 tolerance (Last-value Prediction) for stable regions. Original vs. Predicted IPC (top) and prediction error (bottom).



(a) Original vs. Predicted IPC (top) and prediction error (bottom).



(b) Variable window size.

Figure 2. IPC prediction for 10 % error tolerance (variable window).

Figures 1 and 2 show two sets of IPC value prediction results for the SPEC2K ammp benchmark. Figure 1 has three timeline graphs. In the top timeline, we plot the original IPC as collected from the hardware counters directly. (In all the presented traces, we show IPC values normalized to the maximum seen over the trace.) Superimposed on this is the IPC as predicted by our value prediction method. The third graph plots IPC prediction error versus time. Figure 2 has these three graphs as well as a fourth which depicts the prediction window size as a function of time. (For the zero-tolerance case in Figure 1, the window size is by definition always equal to 1.) The predicted IPC line is discontinuous

	MEAN ABSOLUTE ERROR						AVERAGE WINDOW SIZE						% RUNTIME
	Last Value	1%	5%	10%	50%	Fixed Window	Last Value	1%	5%	10%	50%	Fixed Window	COVERAGE
ammp_in	0.01	0.01	0.02	0.03	0.06	0.07	1	26.57	44.03	66.74	112.25	125.40	77.87
applu_in	0.01	0.01	0.01	0.01	0.11	0.11	1	5.91	7.58	8.36	123.65	125.40	48.34
apsi_NONE	0.00	0.00	0.00	0.01	0.11	0.39	1	16.66	19.71	20.67	39.27	125.40	70.59
art_ref1	0.04	0.04	0.03	0.04	0.03	0.03	1	1.16	3.10	12.28	125.35	125.40	57.16
bzip2_graphic	0.03	0.03	0.03	0.04	0.13	0.26	1	1.64	2.82	3.50	15.15	125.40	7.04
crafty_in	0.02	0.02	0.02	0.02	0.02	0.02	1	1.63	26.09	101.37	125.40	125.40	96.04
equake_in	0.01	0.01	0.01	0.01	0.06	0.06	1	1.55	3.93	4.26	125.40	125.40	4.48
facerec_ref	0.00	0.00	0.00	0.00	0.17	0.16	1	8.20	11.45	11.68	59.52	125.40	24.15
fma3d_NONE	0.00	0.01	0.01	0.01	0.15	0.19	1	29.76	36.46	38.95	106.32	125.40	71.39
galgel_in	0.02	0.02	0.02	0.01	0.12	0.16	1	1.99	18.01	22.60	82.21	125.40	73.45
gap_ref	0.01	0.01	0.01	0.01	0.07	0.09	1	12.87	34.54	72.74	115.05	125.40	85.63
gcc_integrate	0.01	0.01	0.01	0.01	0.05	0.26	1	3.83	28.14	30.55	61.49	115.96	71.72
gzip_random	0.01	0.01	0.01	0.01	0.02	0.02	1	8.70	82.46	91.99	116.32	125.40	94.48
lucas_in	0.01	0.01	0.01	0.02	0.11	0.11	1	3.54	15.86	25.24	122.77	125.40	78.79
mcf_inp	0.01	0.01	0.01	0.01	0.01	0.01	1	6.50	122.20	122.35	125.40	125.40	99.50
mesa_in	0.01	0.01	0.01	0.01	0.01	0.01	1	8.98	10.01	21.45	123.07	125.40	76.57
mgrid_in	0.02	0.02	0.02	0.02	0.03	0.15	1	1.78	2.07	2.21	6.66	125.40	0.10
parser_ref	0.03	0.03	0.03	0.05	0.07	0.07	1	1.83	7.65	18.84	118.82	125.40	61.51
perlbmk_makerand	0.00	0.00	0.00	0.00	0.06	0.06	1	65.28	67.81	68.69	73.66	73.66	92.00
sixtrack_inp	0.02	0.02	0.01	0.01	0.02	0.02	1	3.71	41.09	98.85	125.40	125.40	96.42
swim_in	0.00	0.00	0.01	0.01	0.16	0.16	1	8.95	9.39	10.33	125.32	125.40	53.78
twolf_ref	0.00	0.00	0.00	0.00	0.01	0.01	1	48.69	80.72	117.89	122.94	125.40	98.86
vortex_bendian3	0.02	0.02	0.02	0.02	0.05	0.06	1	3.29	19.69	80.28	115.60	125.40	89.40
vpr_place	0.00	0.00	0.00	0.00	0.00	0.00	1	89.19	122.37	123.32	125.32	125.40	99.52
vpr_route	0.03	0.03	0.03	0.05	0.07	0.07	1	1.40	4.67	23.80	125.35	125.40	68.37
wupwise_NONE	0.00	0.00	0.01	0.01	0.08	0.11	1	11.39	50.28	50.85	78.58	125.40	84.81
AVE	**0.01**	**0.01**	**0.01**	**0.02**	**0.07**	**0.10**	**1**	**14.42**	**33.54**	**48.07**	**99.86**	**123.05**	**68.54**

Figure 3. Summary of IPC prediction for SPEC suite. Left half shows the mean absolute error with respect to measured IPC and the right half shows the average size of variable window. The last column is the prediction coverage based on the stability criterion.

because we only predict in stable regions.

First consider Figure 1, where the tolerance is set to 0. This means that two adjacent samples will never be considered stable and the window size is always equal to 1. Qualitatively, the top graph shows that actual and predicted IPC look fairly similar. The third graph plots the prediction error; this is the difference between actual and predicted IPC. Because we only make predictions in the stable regions, the maximum error observed is quite low: less than 10%.

For comparison, we also plot in Figure 2 the same values for the case where window size is increased as long as IPC readings remain within a 10% tolerance of their predecessor. First, fairly large window sizes become commonplace for this tolerance; the average is 66.74 samples for this run. This is an indicator of the repetitive phase behavior in the program. Second, error is slightly increased from the zero tolerance case. This is actually a good indicator, which we base our long term metric value predictions in Section 4.3. 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. However, over a long period of stability, the benchmarks can show a trend of increasing (as in the case of ammp) or decreasing IPC. For this reason, our long-term IPC predictions in Section 4.3 use the last-value together with the observed inter-sample gradient to predict long durations of IPC behavior.

The summary data across the SPEC workloads is shown in Figure 3. The table shows mean absolute error and average window size for a range of prediction possibilities. All have very low errors: less than 0.1 IPC for the worst-performing fixed-window approach, and roughly 0.01 IPC for the best-performing last-value predictor. In general, we see very similar prediction accuracy across predictors for flat benchmarks such as art and crafty. On the other hand, benchmarks with observable gradients such as ammp and vortex consistently do better with last value prediction.

For comparison, we also tried a method in which predictions were made on every sample, regardless of the stability criterion. Here, all predictors, including last value, resulted in higher errors, as bursty behavior confounds both last value and averaging approaches. Under applications that might require prediction over bursty regions, Duesterwald et al. [12] show table based approaches actually perform better. However, in our case, our goal is to predict long stable durations in workload execution, and last value together with gradient information proves to be the right choice for both simplicity and accuracy.

Based on the results presented here, subsequent sections start from last-value prediction, and add on small refinements for gradient-based prediction, which is performed only at prediction checkpoints and for long periods of execution.

4 Duration Prediction

The key lynchpin of our work is that we wish to predict the minimum duration of stable periods in an application's phase behavior. We know of no prior work that has done this, either for simulation or real-system measurements. The benefit of such prediction techniques is that if we can say with confidence that a particular behavior will last for at least 1 second or 100 samples, or some such duration, then we can plan out responses to the behavior that are commensurate with it.

4.1 Duration Prediction Overview

This section introduces the duration prediction methods we have considered. 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 definition as previously described for IPC value prediction. We require 8 consecutive samples of stability to initiate prediction, where stability is defined by the same stability criterion as Section 3.

The second question for duration prediction is *what* to predict. Here, we discuss several tradeoffs, before narrowing in on the possibilities we consider. Duration prediction is distinct from say 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 as 1 cycle 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. Furthermore, such short duration predictions mean that one has to spend much more system time on making predictions, rather than on executing applications.

Our goal with duration prediction is to be able to predict very-long-duration events with good accuracy, so that we can set a response in place and expect some degree of stability for the time duration we imposed. In this section, we look at several prediction possibilities, and evaluate them based on fairly generic metrics concerning their coverage and accuracy. In Section 5 we apply these prediction techniques to particular problems, where they can be evaluated in a more domain-specific manner.

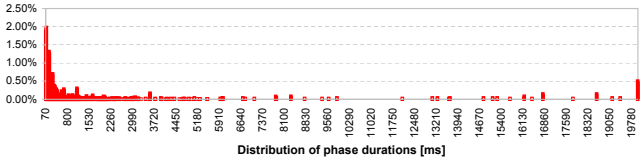


Figure 4. Distribution of stable phase durations over all SPEC reference datasets.

We start with a characterization of duration statistics in the workloads we study. Figure 4 gives a histogram showing the frequency of occurrence for different stable phase lengths, evaluated over more than 40 reference inputs of 25 SPECCPU benchmarks. The distribution is plotted at 10 ms consecutive bins up to 20 seconds. The y-axis shows the percentage distribution of phase durations into those bins. While short phases are definitely the norm, non-negligible amount of phases are quite long. Indeed, applying curve-fitting techniques to the histogram indicates that it displays the heavy-tailed characteristics already documented in other aspects of computer systems behavior [9]. The flat distribution of this heavy tail presents the critical challenge in duration prediction. Most workloads exhibit phase behavior interleaved with short bursty regions. However, the duration of these stable phases are highly variable from application to application and in cases, from phase to phase.

Going back to the stability argument, choosing a loose stability criterion to avoid predicting during unstable regions also misses out on some of the strongest part of the distribution. However, when we weight the distribution information with the amount of time each phase consumes, significantly higher portions of the runtime are spent in the heavy-tail region. Of the measured 25 SPEC benchmarks, 17 spend more than 70% of their runtime in phases that last 200ms to 2 seconds, while only equake, mgrid and bzip2 tend to be primarily operating at phase granularities smaller than our stability definition.

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

In response to the conservatism of this simple constant predictor, we have also looked at two more aggressive predictors. The first of these we refer to as $f(x) = x$ and denote it from now on as 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. When we have seen 8 stable samples, we make our first prediction—that we will see 8 more. At the end of that predicted period, if things have remained stable, it will add up to a total of 16 samples. Thus, our next prediction will be to see 16 further stable cycles into the future. 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 prediction approach is that it is prone to significant overshoot when a phase does end.

To try to lessen the overshoot problem, we have also looked 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 in the following results section.

In Figure 5, 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 on it is an IPC value prediction (as in Section 3) 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. Furthermore, the flat IPC predictions are somewhat inaccurate for a long-term prediction scenario. After presenting our general evaluation for duration, we next describe and evaluate a more effective gradient-based method for long-term IPC prediction, where we extrapolate on IPC trends.

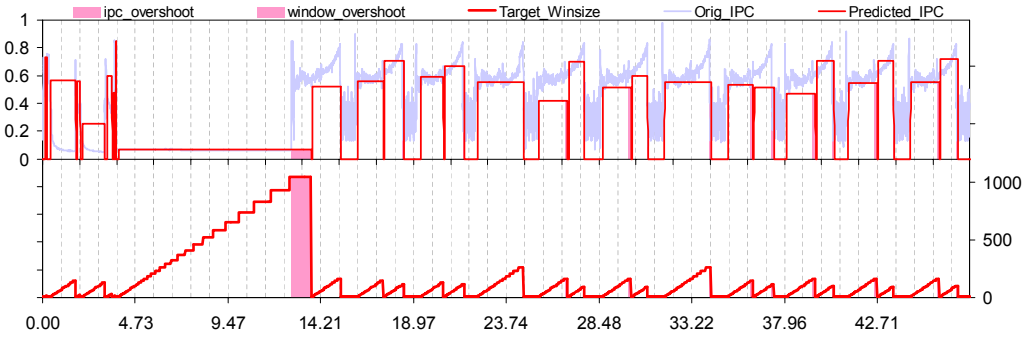


Figure 5. Duration prediction for ammp, for a stability of 8 and FXby8 prediction.

4.2 Duration Prediction Results

To evaluate the success of our duration prediction methods, we first have to determine metrics for gauging them. As previously stated, our goal is to have a predictor that gets good accuracy on predictions when it makes them, and that makes predictions often enough to be worthwhile. Thus, measures of a predictor's success include:

- **Accuracy** or **Safety**: These two terms are used by different communities to refer to a method's rate of correct predictions *given that it has chosen to make a prediction* [4].
- **Safe Duration**: Some duration predictions predict very short durations of stability, while others 'go long' in predicting long stable periods. There is value in predicting long durations, and so we wish to evaluate the methods not just on the frequency at which they predict correctly, but also on the duration of time for which a correct prediction holds. We refer to these durations as safe durations, and we report the mean lengths of these safe duration periods.
- **Degree of Overshoot** The downside to long duration predictions is that they may significantly overshoot the actual end of the phase behavior. Our figures of merit to evaluate this are the mean duration of an overshoot, and the percentage of the program runtime that is spent in overshot predictions.

Figure 6 presents the accuracy results for the three duration predictors discussed. The naming convention in the table is SPEC benchmark_inputfile. All data is collected for full runs of the benchmark with the reference inputs, and for vpr, we present the place and route datasets separately as they have distinctly different behavior. For each benchmark, and for each of the three duration predictors, this table presents three results metrics. The first metric, "safe predictions", is a count of the number of duration predictions in which the stable phase lasted at least as long as predicted. The second metric, "incorrect predictions", refers to overshot predictions. The third metric, accuracy, is the ratio of safe predictions to the total number of predictions.

As one can see, even for the fairly coarse-grained sampling and conservative stability definition (8 samples before thinking about prediction), these benchmarks typically show hundreds of points where duration prediction makes

	SAFE PREDICTIONS			INCORRECT PREDICTIONS			ACCURACY		
	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	456	57	639	28	16	27	0.942	0.781	0.959
applu_in	112	81	1603	209	173	223	0.349	0.319	0.878
apsi_NONE	308	129	1630	142	93	146	0.684	0.581	0.918
art_ref1	164	95	1641	195	158	218	0.457	0.375	0.883
bzip2_graphic	4	2	84	51	56	78	0.073	0.034	0.519
crafty_in	579	23	455	20	8	20	0.967	0.742	0.958
equake_in	21	4	65	6	2	6	0.778	0.667	0.915
facerec_ref	130	49	376	19	19	19	0.872	0.721	0.952
fma3d_NONE	351	101	1123	102	49	95	0.775	0.673	0.922
galgel_in	334	126	1601	126	115	126	0.726	0.523	0.927
gap_ref	482	81	783	57	34	58	0.894	0.704	0.931
gcc_integrate	49	15	168	14	10	16	0.778	0.600	0.913
gzip_random	563	57	623	27	21	27	0.954	0.731	0.958
lucas_in	379	118	1643	113	74	116	0.770	0.615	0.934
mcf_inp	619	17	124	2	2	2	0.997	0.895	0.984
mesa_in	322	135	1820	151	118	151	0.681	0.534	0.923
mgrid_in	0	0	3	1	1	2	0.000	0.000	0.600
parser_ref	245	107	1354	141	117	151	0.635	0.478	0.900
perlbmk_makerand	16	4	27	0	0	0	1.000	1.000	1.000
sixtrack_inp	584	40	492	19	9	18	0.968	0.816	0.965
swim_in	155	91	1541	183	162	272	0.459	0.360	0.850
twolf_ref	612	28	216	5	5	5	0.992	0.848	0.977
vortex_bendian3	520	50	717	39	17	37	0.930	0.746	0.951
vpr_place	619	15	105	2	2	2	0.997	0.882	0.981
vpr_route	284	106	1551	146	115	162	0.660	0.480	0.905
wupwise_NONE	473	116	1073	58	53	57	0.891	0.686	0.950
						AVERAGE	**0.740**	**0.607**	**0.906**

Figure 6. Safety measures for three different duration prediction schemes.

sense. For the simple constant8 predictor, it is able to make an average of 325 safe predictions per application across the benchmarks, while the fast-growing FXX function makes many fewer predictions since its predicted durations are longer. Overall, FXX shows the worst prediction accuracy, since it tends to overshoot phases so often; these overshoots count as incorrect predictions. On the other hand, FXby8 shows the best prediction accuracy, at 90.6%, because it grows slowly at first and does a better job of capturing short and intermediate length stable regions.

Prediction accuracy is important, but it is only one piece of the puzzle. A second aspect of a predictor, as previously mentioned, is the typical durations it is able to successfully predict. These results are shown in Figure 7. The three leftmost columns in the figure characterize each benchmarks *true* stable-phase behavior. The first column indicates the percentage of program runtime spent in a stable duration The second column gives a count of the number of such stable regions in the benchmark's full runlength. The third column gives the mean length (in 10ms samples) of a duration.

After this point, the table presents results on each predictor's ability to capture these stable regions. The constant8

	% STABLE DURATIONS	# STABLE REGIONS	MEAN STABLE DURATION	# PREDICTED STABLE REGIONS			MEAN SAFE PREDN DURATION			Predicted Stability / Total Stability		
				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	30	136.73	24	15	27	8	34.67	5.43	0.89	0.48	0.85
applu_in	79.99	226	17.69	108	80	222	8	8.1	1.11	0.22	0.16	0.45
apsi_NONE	93.84	166	28.25	113	77	143	8	12.34	1.73	0.53	0.34	0.60
art_ref1	91.34	244	18.71	79	74	207	8	10.11	1.37	0.29	0.21	0.49
bzip2_graphic	26.23	137	9.57	1	1	49	8	12	1.19	0.02	0.02	0.08
crafty_in	98.98	21	235.57	18	6	20	8	105.7	9.37	0.94	0.49	0.86
equake_in	5.32	6	44.33	4	1	6	8	30	2.75	0.63	0.45	0.67
facerec_ref	26.95	20	67.35	18	18	19	8	17.96	2.83	0.77	0.65	0.79
fma3d_NONE	88.32	121	36.48	59	47	94	8	18.3	2.65	0.64	0.42	0.67
galgel_in	93.06	140	33.22	105	82	125	8	11.68	1.86	0.57	0.32	0.64
gap_ref	95.58	71	67.28	30	27	58	8	26.77	4.77	0.81	0.45	0.78
gcc_integrate	87.17	16	39.5	9	6	14	8	17.07	2.55	0.62	0.41	0.68
gzip_random	98.4	28	175.64	26	19	27	8	42.25	6.76	0.92	0.49	0.86
lucas_in	96.44	126	38.25	102	55	105	8	15.93	2.04	0.63	0.39	0.70
mcf_inp	99.92	3	1664.67	3	3	3	8	152.9	36.53	0.99	0.52	0.91
mesa_in	97.86	152	32.18	117	89	152	8	11.44	1.71	0.53	0.32	0.64
mgrid_in	0.38	2	9.5	0	0	2	0	0	1	0.00	0.00	0.16
parser_ref	87.13	183	23.8	76	67	137	8	12.19	1.79	0.45	0.30	0.56
perlbmk_makerand	96.67	1	145	1	1	1	8	30	4.44	0.88	0.83	0.83
sixtrack_inp	99.5	22	226.05	19	10	19	8	63.2	8.85	0.94	0.51	0.88
swim_in	93.98	287	16.37	98	90	240	8	8.09	1.26	0.26	0.16	0.41
twolf_ref	99.7	6	830.5	6	6	6	8	97.14	20.42	0.98	0.55	0.89
vortex_bendian3	95.7	45	106.29	33	15	37	8	47.84	5.51	0.87	0.50	0.83
vpr_place	99.94	3	1665	2	2	3	8	203.7	44.69	0.99	0.61	0.94
vpr_route	93.02	176	26.41	82	67	154	8	13.36	1.83	0.49	0.30	0.61
wupwise_NONE	93.36	61	76.49	45	44	57	8	17.79	3.42	0.81	0.44	0.79

Figure 7. Efficiency measures for three different duration prediction schemes.

predictor captures many of the stable regions, but the FXby8 predictor actually captures the most stable regions of the three. This is because its prediction starts out slowly: at the first prediction, x=8, so the first few predictions are always a timid single-cycle duration prediction, and then they grow significantly for longer stable regions. Thus, this function is best able to capture stable phases of durations around 80-160ms. The bzip2 benchmark is the best illustration of this; it has 137 phases with an average stable duration of 9.57 samples. The FXby8 predictor is the only one able to safely capture more than one of them.

The next set of columns depict the average interval size of durations predicted *safely* by each predictor. Predictions that turn out to be incorrect are not counted in this average. By design, the constant8 predictor always has a safe prediction size of 8. The other two columns are more interesting. FXby8, which has thus far been our best predictor, shows that it accomplishes its success by often making fairly short predictions. On the other hand, for applications like vpr and mcf that have a few very long phases, FXby8 is able to eventually work out to long duration predictions, with average interval lengths of 35 or more. FXX is quite good at reaching the long intervals more quickly; its mean safe duration length is typically above ten, and even tops out at 105.7 for crafty, which has 21 long, relatively easy-to-predict regions.

Overall, one good measure of a predictor's success is in what fraction of a program's true stability it is able to successfully capture. This is given in the rightmost set of columns of Figure 7. The FXby8 predictor is the best of the three 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 simple constant8 predictor has better coverage than FXby8, and in many cases they are quite close.

	MEAN OVERSHOOT			% runtime in OVERSHOOT		
	f(x)=8	f(x)=x	f(x)=x/8	f(x)=8	f(x)=x	f(x)=x/8
ammp_in	4.04	77.63	10.63	2.26	24.85	5.74
applu_in	4.84	8.40	1.31	20.25	29.07	5.84
apsi_NONE	5.04	17.79	1.82	14.31	33.09	5.32
art_ref1	4.73	9.38	1.57	18.45	29.65	6.84
bzip2_graphic	6.94	7.34	1.00	7.08	8.22	1.56
crafty_in	4.70	214.88	6.95	1.88	34.39	2.78
equake_in	4.17	50.00	1.50	0.50	2.00	0.18
facerec_ref	4.05	39.53	4.63	1.54	15.03	1.76
fma3d_NONE	5.13	33.18	2.56	10.46	32.53	4.86
galgel_in	4.61	11.92	2.21	11.62	27.43	5.58
gap_ref	5.47	21.38	4.31	6.24	14.55	5.00
gcc_integrate	3.21	20.30	2.94	0.90	4.06	0.94
gzip_random	4.82	61.24	8.30	2.60	25.73	4.48
lucas_in	4.74	25.72	2.46	10.72	38.07	5.70
mcf_inp	3.00	783.00	16.00	0.12	31.33	0.64
mesa_in	4.44	13.75	2.13	13.41	32.47	6.42
mgrid_in	6.00	6.00	1.00	0.12	0.12	0.04
parser_ref	4.65	13.17	1.83	13.12	30.83	5.52
perlbmk_makerand	0.00	0.00	0.00	0.00	0.00	0.00
sixtrack_inp	5.26	176.89	15.33	2.00	31.85	5.52
swim_in	5.19	7.05	1.34	18.99	22.85	7.30
twolf_ref	5.60	120.80	21.40	0.56	12.08	2.14
vortex_bendian3	5.05	109.29	6.73	3.94	37.17	4.98
vpr_place	4.00	300.00	15.00	0.16	12.00	0.60
vpr_route	5.15	13.27	1.76	15.05	30.53	5.70
wupwise_NONE	4.28	13.64	3.04	4.96	14.47	3.46

Figure 8. Overshoot measures for three different duration prediction schemes.

The last figure of merit in designing duration predictors is the degree of overshoot they exhibit. This is given in Figure 8. FXX displays poor performance, with very long

overshoots in some cases, and with a large fraction of total program runtime spent in overshoot. Between FXby8 and constant8, the distinction is once again more subtle. They both have low overshoot and tend to offer quite good performance. FXby8 tends to have lower overshoots for the benchmarks that have shorter phases, but when it is wrong, it can be quite wrong. For example, it has double-digit average overshoots for ammp, mcf, sixtrack, twolf, and vpr. These are all benchmarks with very long mean stable durations—150 samples or more–that lead the dynamic prediction to build up and overshoot significantly.

4.3 Combining Value and Duration Prediction

As we have mentioned briefly at the beginning of this section, the duration prediction method is decoupled from IPC prediction. This section seeks to combine them. As seen in Figure 5, a simple last-value IPC prediction is good for short-term prediction, but less effective as one uses it 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 usually a reasonable assumption.

In Figure 9, 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 are quite good. The only points of significant error are where the duration predictor overshoots, leading to higher IPC prediction error as a phase ends.

In Figures 10 and 11, we show the results of duration and gradient prediction for the whole SPEC suite. We use the same previous three prediction functions to evaluate our results. Except for the very highly variable benchmarks like bzip2, equake and mgrid, the long-term IPC prediction performs quite well, especially with the FXby8 function. The coverage plot, by definition, presents the percentage of runtime the predictor actually made a prediction, including overshoot. Therefore FXX function in general has slightly higher coverage. However, Figure 7 shows it has the least stable coverage of the three. More importantly, Figure 11 provides a direct one to one comparison to the local prediction methods in terms of coverage which is shown in Figure 3. For all benchmarks, our long-term predictions achieve very close coverage to those of local predictions.

4.4 Summary

Overall, duration prediction is a new aspect to phase prediction research. An interesting wrinkle to this sort of prediction is that there are two ways for a predictor to be conservative: it can either not guess at all, lowering its coverage of stable regions, or it can guess very short intervals. The FXby8 predictor is much better than the rest in terms of accuracy, and its hardware complexity, while higher than constant8, is not excessive. On the other hand, it is roughly a toss-up with constant8 in terms of figures of merit other than accuracy, such as length of predicted durations and degree of overshoot. If a particular application requires very long predictions for its resource planning, then FXby8 or even FXX might be preferable, depending on the trade-off between reaching high predictions quickly and penalty of overshoot. In addition, if the actual cost of checkpointing the behavior is significant compared to the penalty of overshoot, then the more aggressive approach, FXX can turn out to be appealing with its few checkpoints.

5 Applications of Duration Prediction

This section gives an example of a concrete application of duration prediction. In particular, we explore its applicability to a dynamic voltage/frequency scaling (DVS) scenario. For DVS, the goal is to identify program periods with "slack", in which slowing down the processor core will save energy with little impact on performance [25]. These periods are typically memory-bound periods in the code. During these periods, we can operate the processor at a lower voltage and frequency, in order to save energy. Since memory latencies are decoupled from processor clock rate, one can often operate at the low-frequency mode with little to no performance impact—*if* sufficiently memory-bound regions are selected.

DVS has been heavily studied, and so our results here are mainly to demonstrate an application of value/duration prediction. For this reason, we focus on a somewhat stylized view of the DVS 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. Practical processors use much more than two settings typically, but we focus first on the two-setting case. 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 DVS switches required, since these voltage/frequency adjustments cost both time and energy.

5.1 DVS Policy

We looked at several possible metrics to predict and exploit slack in different ways. Possibilities we considered include DTLB misses, ITLB misses, off-chip L3 references, and L3 misses. In addition, the counter devoted to data table walks (DTWs) indicates cycles spent on behalf of OS page table activity. We ruled out the ITLB count fairly quickly since across the SPEC benchmarks we found that instruction references have little impact in performance due to higher locality and high predictability. Other metrics, like DTLB misses, were ruled out because they occur so sporadically that it is difficult to predict based on them.

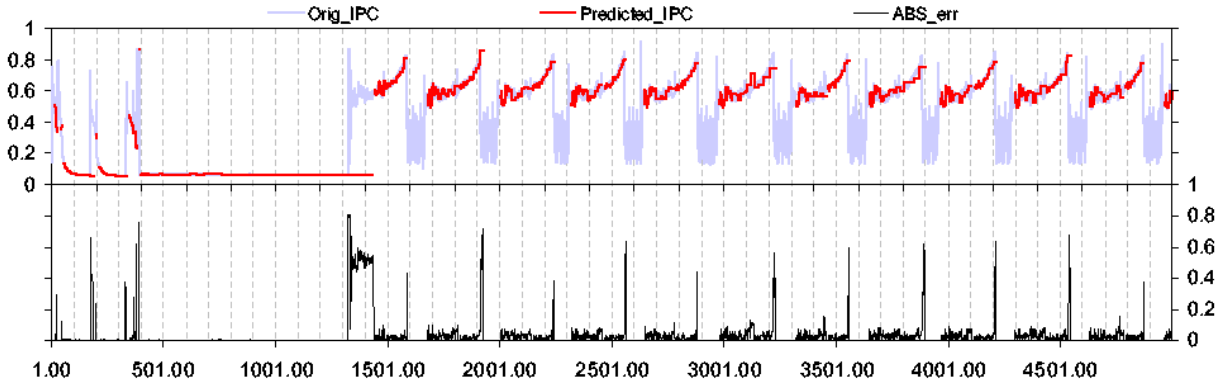


Figure 9. Ammp duration and gradient based metric value prediction for f(x)=x/8.

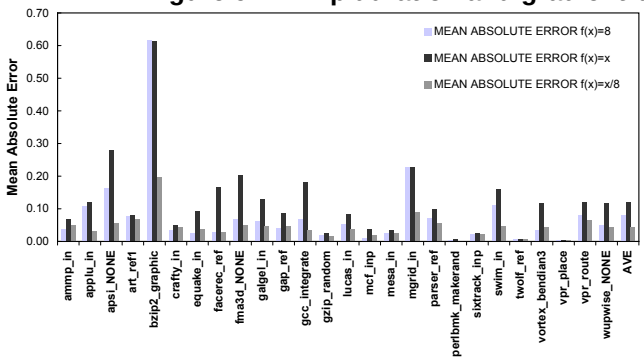


Figure 10. Duration and gradient based long term future IPC prediction results for SPECCPU benchmarks.

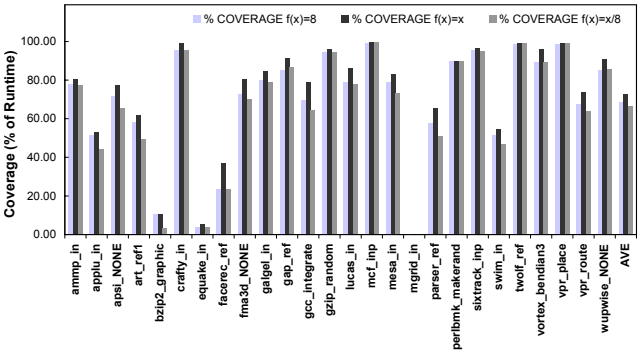


Figure 11. % Coverage of long term IPC and duration prediction.

For the remaining metrics under consideration (DTWs, L3 references) we used scatter plots with IPC (not shown due to space constraints) to make our policy choice. Our goal is to find metrics of significance, and to use as few counters as possible. Clear breakpoints were observed across the SPEC benchmarks. As a result of these breakpoints, we refer to a *low IPC* region when the IPC values are less than 25% of maximum observed IPC. Similar boundaries for DTWs and L3 references are 6% and 10% respectively.

When we analyze the statistics of these metrics over the whole SPEC suite, the probability of encountering high L3 references given that we observe low IPC is 81%, while the same probability for DTWs is only 58%. This is because there is no significant case where DTWs are high while L3 references are low, while the converse is possible– i.e. art, which has very high L3 references with no significant DTWs. This means that if we only have two counters available to us, IPC and L3 references offer the best significance. If we have three counters available to us, we can do a joint IPC/L3/DTW prediction: the probability that either L3 references or DTWs are high given that IPC is low is 89%. So, there is an additional 8% prediction that can be extracted with the help of DTWs. However, since the POWER4 counter architecture precludes reading both L3 references and DTWs in the same measurement, we focus from here forward on DVS policies based on IPC and L3 references.

Having decided on the metrics to use for our DVS policy, we turn now to the other specific aspects of our approach. Our goal is to explore performance-oriented DVS policies; if we are in an unstable region where predictions are uncertain, we use the highest-possible clock frequency to avoid harming performance. Duration prediction is performed (with gradient prediction) as described in the previous section, and our definition of region stability also holds. Layered on top of the metric prediction is a DVS policy decision.

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 high (greater than 10% of the maximum value).

During stable phases, we can make good predictions about program behavior and apply DVS accordingly. 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 DVS state as-is, or one can return the DVS state back to high-energy/high-performance. In early experiments, we used the more conservative technique of retaining the DVS state as-is. However, although this works quite efficiently for very stable

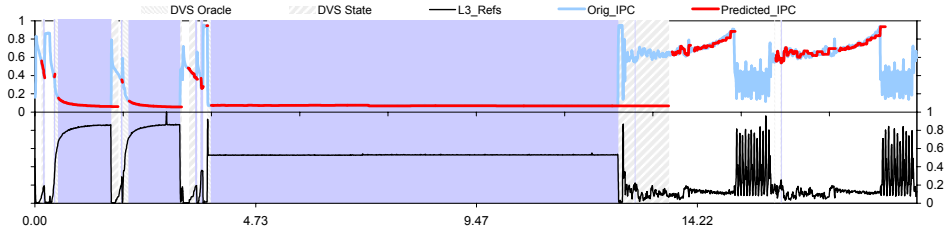


Figure 12. Duration and long term value prediction applied to DVS for ammp.

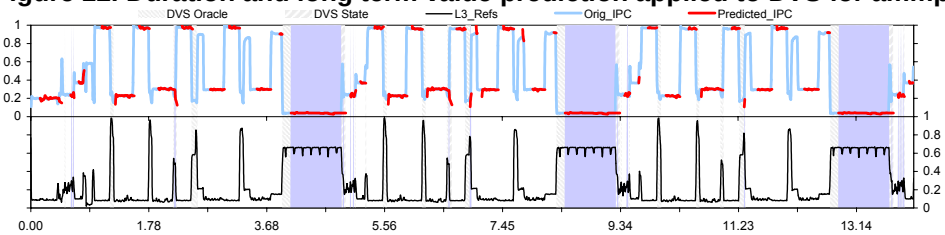


Figure 13. Duration and long term value prediction applied to DVS for apsi.

benchmarks, it can result in significant performance penalty in highly-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 Figure 7. The predictor catches one of them and pushes the DVS 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 DVS state is never returned. As a result, mgrid was incorrectly kept in the low energy state 87% of its runtime.

Based on our experiences with unstable benchmarks like mgrid, and based on our goal of valuing performance as a primary metric, we instead use a policy in which unstable regions all revert to the high-energy DVS state.

5.2 DVS Results

We will show DVS results for four SPEC benchmarks The four chosen benchmarks represent different corners of workload behavior. Ammp presents a case with repetitive large-scale phases. On the other hand, apsi has numerous smaller scale phases. Mgrid is a very bursty benchmark with almost no stable phase. And mcf has a long stable phase with a significant gradient.

Figures 12 and 13 show DVS for two of the benchmarks. (The other two are not shown as timelines due to space constraints, but their statistics are given in the summary graphs and tables to follow.) In this figure, the duration prediction uses the FXby8 predictor and the value prediction uses the long-term gradient method. The IPC and L3 reference rates are normalized. In the figures, the thin striped "DVS oracle" regions correspond to portions of execution that would be DVS'd using our criterion but with oracle knowledge, i.e., zero-error duration and value prediction. The thick striped "DVS state" shows the regions where DVS is actually applied based on our duration predictor and the solid shaded regions show where the two overlap. In general, there is a performance penalty for points where our predictor is in low-energy mode but the oracle is not. Likewise, there is an energy penalty associated with regions where the oracle is in low-energy mode but our predictor is not. We include these timelines to give a qualitative feel about the style and quantity of DVS opportunities that exist. More quantitative results are in the graphs and tables that follow.

Figure 14 summarizes the effectiveness of DVS for four benchmarks, for different duration prediction methods. For each application, we show five bars. The rightmost bar shows the percentage of runtime that would be spent in low-energy DVS mode with oracle knowledge of system behavior. The other four bars are stacked bars giving a DVS results breakdown for different predictors. The predictors we show here are the three familiar ones: constant8, FXX, FXby8, and a fourth one: constant1. This fourth predictor predicts every stable cycle, and thus never overshoots for more than one sample.

For each function, the lowest portion of the stacked bar shows 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. This represents the overshoots our predicted duration will cause. The top portions of the stack then represent the converse: lost opportunity time. That is, they plot the amount of time where the application is truly in low-energy mode, but our prediction has been for high-energy.

These stacked bars show first, that significant DVS opportunities exist, and second, that most of the predictors are able to capture them. Except for mgrid, all the benchmarks spend 20% or more of their time in low-energy mode. Mgrid's behavior is distinct, as it is too unstable to make DVS predictions. All of the predictors come within 92% of the oracle approach. Ammp and apsi with the FXX predictor have a somewhat larger incorrect low-energy mode. In case of ammp, this is due to a single large overshoot after its first huge low-energy phase. In apsi, this is due to smaller overshoots accumulated over the large number of smaller phases. On the other hand, apsi spends relatively more time in the lost-opportunity high-energy mode. This is because of short low-energy regions that are slightly larger than the stability criterion. In this case the initial fixed sta-

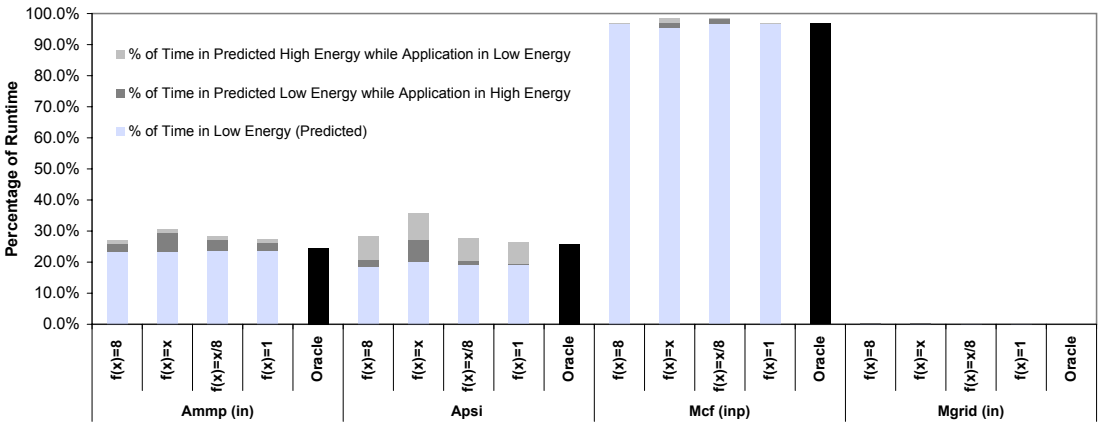


Figure 14. Evaluation of duration and gradient prediction under DVS.

bility timeout of 8 samples becomes comparable to the actual predicted durations. Consequently, even the most timid prediction scheme incurs a lost-opportunity compared to an oracle predictor. We include the constant1 results to demonstrate that our prediction-based techniques achieve results nearly identical (within 1%) to those of a method that reads counters on every cycle. The relatively simple predictors we propose offer equal accuracy and a greater degree of autonomy for this sort of system adaptation.

In Figure 15, we show the number of DVS's made as a proxy to DVS overhead cost and number of predictions made as a proxy to the prediction overhead cost. For both metrics, lower quantities are better. For all prediction methods and benchmarks, the number of DVS's required is quite low. While constant1 makes many predictions and DVS's, the other predictors are much more stable. The apsi benchmark shows some weakness in FXX, where its more aggressive behavior leads to more DVS's. The predicted gradients in the overshot regions lead to additional false DVS regions. Mgrid and mcf show very low DVS transition counts, but 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 and exploit it. 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.

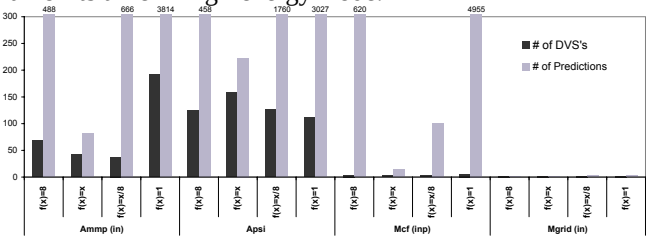


Figure 15. Overheads associated with prediction and DVS.

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–which is equivalent number of counter reads for metric updates.

6 Related Work

Prior phase characterization and prediction work operates at various abstraction levels with different endgoals. One group of research particularly focuses on offline simulator or counter analysis to select representative execution points [17, 18, 21, 22, 24]. While this work is largely phase *characterization* research, our work is distinct in that it focuses on *predictions* guided by online phase change detection.

A second class of work explores architectural dynamic optimizations guided by fine-grained phase information [11, 14, 23]. These are fine grained techniques requiring detailed execution space information, while our work operates at large scale phases. Furthermore, our work uses transition information only at prediction boundaries, thus removing monitoring from the main loop and allowing long durations of time without system probes or adjustments.

At the operating system level, some application-oriented work has been done to read hardware counters and use their information for techniques like DVS or thermal-aware OS scheduling [3, 7, 25]. All of these are reactive, while our presented approach shows the ability to predict and to proactively adjust system behavior. Furthermore, these techniques all rely on fixed-interval system probes, while our techniques perform duration prediction so as to reduce the number of system probes required during the (not uncommon) very long phases.

Last, Duesterwald et al. [12] recently present the performance of various coarse-grained (also millisecond granularity) prediction schemes for performance counter metrics on two different IBM POWER platforms. They show the possibility of cross metric predictions and mention potential applications of such predictive techniques. Their schemes, however, focus on near-term sample-to-sample predictions. Our work has developed longer-term approaches incorporating duration prediction with gradient prediction schemes.

7 Conclusion

Duration prediction, in conjunction with future 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 the first published methods for applying duration and long-term value prediction effectively. Our methods achieve prediction accuracies close to 90% of actual stable durations, despite the high variability of phase lengths across the SPEC suite. A key to our approach is pairing duration prediction with gradient-based long-term value prediction for stable regions. This allows several duration predictors to achieve low errors across the SPEC suite. For example, our FXby8 predictor achieves an average IPC error of 5% over the whole SPEC suite, including highly variant bzip2 and mgrid. These predictors provide similar prediction coverage (66.5% on average over SPEC) as sample-by-sample predictors (68.5%), while avoiding constant counter polling. Over the SPEC suite, even our less agressive predictors monitor performance behavior 10X less frequently than a constant-polling system.

The contributions of this work are: (i) a comprehensive analysis of different prediction scenarios, both near-term and long-term, over the whole SPEC suite; (ii) a simple-yet-effective duration prediction method, which provides accurate long-range information without the need for constant polling; (iii) a gradient-based, long-term value prediction method which aids in extrapolate application phase behavior over long stable durations; and (iv) description and conceptual evaluation of a dynamic management application (DVS) that can benefit from this method.

With an increasing industry-wide focus on adaptive and autonomous system management, schemes for predicting and responding to very-long-term system behavior (tens of milliseconds up to seconds) become critical. The work presented here offers practical, low-overhead techniques for such long-range prediction, as well as evaluations of their possible application to important concrete problems.

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] F. Chang, K. Farkas, and P. Ranganathan. Energy driven statistical profiling: Detecting software hotspots. In *Proceedings of the Proceedings of the Workshop on Computer Systems*, 2002.

[7] K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004.

[8] 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.

[9] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE /ACM Transactions on Networking*, 5(6):835–846, 1997.

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

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

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

[13] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, Feb. 1999.

[14] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the International Symp. on Computer Architecture*, 2003.

[15] 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.

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

[17] 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.

[18] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the 36th International Symp. on Microarchitecture*, Dec. 2003.

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

[20] I. Kadayif, T. Chinoda, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. vEC: virtual energy counters. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 28–31, 2001.

[21] 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.

[22] 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/.

[23] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-30)*, June 2003.

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

[25] 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.

[26] A. Weissel, B. Beutel, and F. Bellosa. Cooperative i/o-a novel i/o semantics for energy-aware applications. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation OSDI'02*, 2002.

[27] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Oct. 2002.