Characterizing and Predicting Program Behavior and its Variability

Evelyn Duesterwald
IBM T.J. Watson Research
Yorktown Heights, NY
duester@us.ibm.com

Călin Caşcaval
IBM T.J. Watson Research
Yorktown Heights, NY
cascaval@us.ibm.com

Sandhya Dwarkadas
University of Rochester
Rochester, NY
sandhya@cs.rochester.edu

Abstract

To reach the next level of performance and energy efficiency, optimizations are increasingly applied in a dynamic and adaptive manner. Current adaptive systems are typically reactive and optimize hardware or software in response to detecting a shift in program behavior. We argue that program behavior variability requires adaptive systems to be predictive rather than reactive. In order to be effective, systems need to adapt according to future rather than most recent past behavior.

In this paper we explore the potential of incorporating prediction into adaptive systems. We study the time-varying behavior of programs using metrics derived from hardware counters on two different micro-architectures. Our evaluation shows that programs do indeed exhibit significant behavior variation even at a granularity of millions of instructions. In addition, while the actual behavior across metrics may be different, periodicity in the behavior is shared across metrics. We exploit these characteristics in the design of on-line statistical and table-based predictors. We introduce a new class of predictors, cross-metric predictors, that use one metric to predict another, thus making possible an efficient coupling of multiple predictors. We evaluate these predictors on the SPECcpu2000 benchmark suite and show that table-based predictors outperform statistical predictors by as much as 69% on benchmarks with high variability.

1. Introduction

Workload characterization plays an important role in the design of efficient hardware and software systems. By understanding workload characteristics, modern architectures, both hardware and software, can allocate and tailor resources to best suit the needs of the application. Consequently, there have been numerous studies that evaluated the performance of workloads and attempted to characterize them along dimensions of interest, such as common operations for instruction set design, instruction-level parallelism for micro-architecture design, and cache behavior for memory system evaluation and compiler optimizations. Most of these studies have focused on the characterization of the aggregate behavior of entire applications.

More recently, there has been a focus on dynamic optimization and reconfiguration during program execution both in hardware and in software. Several micro-architectural designs [3, 5, 6, 8, 11, 13, 15, 18, 21, 31] have been proposed that allow the rapid low-cost adaptation of processor components to match hardware complexity and speed with an application's requirements. There have also been several approaches to dynamically adapt the software, such as application code, data placement, or runtime resource allocation. One such optimization is to adapt dynamic memory allocation to more efficiently meet the application's memory access pattern requirements. Operating systems may employ the use of variable memory page sizes [27, 30] to adapt to the memory access patterns of the application. Another example is memory management in Java Virtual Machines (JVM), where the garbage collector's activities are adapted to the application's behavior [7, 23]. Just-In-Time compilers, such as the ones found in JVMs [1, 2], and dynamic optimization frameworks [4, 19] target adaptation of the executing code itself in order to exploit dynamic program characteristics.

Currently, both hardware and software adaptation techniques typically operate in a reactive manner. An adaptive change is triggered *after* a variation in program behavior has been observed. A reactive algorithm uses immediate past behavior as representative of future behavior for on-the-fly adaptation. Reactive adaptation works well as long as programs execute in a sequence of more or less stable phases with relatively few transitions. However, if a program exhibits significant variability in its behavior, a reactive approach suffers by continuously lagging behind. Results in [6] demonstrate that a reactive system may need to increase the sampling interval so that the aggregate behavior captured remains stable for a sufficient amount of time, thus missing opportunities for more fine-grained adaptation. Al-

ternatively, profiling has been used to predict performance for future executions of the program [15, 18]. There have also been studies that attempt to characterize and classify program behavior into phases, where phases are defined as regions of a program with relatively stable behavior [8, 26]. The goal in these papers is the dynamic identification of phases in order to tune resources to meet the application needs.

In this paper, we explore the predictability of program behavior across various dimensions. We first characterize the time-varying behavior of programs with respect to several metrics – instruction mix, branch prediction accuracy, data cache miss rates, and instructions per cycle (IPC). We demonstrate that programs indeed exhibit significant variations in their behavior and these variations persist even at coarse granularity, when aggregating over millions of instructions. We also illustrate the presence of strong frequency components in the analyzed metrics indicating high rates of repeatability in program behavior. The source of periodicity is the looping structure in the program, which manifests itself across all metrics. Thus, while the behavior of different metrics for an application may not be correlated, the periodicity is typically shared: the behavior repeats at the same rate across all metrics.

Based on these observations, we explore the performance of several on-line behavior predictors. Periodicity in program behavior motivates the use of table-based predictors that use behavior in past intervals as an index into the table to predict behavior in the next interval. We also consider statistical models that use interpolation with varying levels of sophistication to predict the behavior in the next interval. In addition, the fact that periodicity of behavior tends to be shared across metrics motivates the use of *cross-metric* predictors that track the behavior of one metric to predict the behavior of others. Cross-metric predictors are interesting because they allow the efficient coupling of multiple predictors using a single source metric. We evaluate these predictors on the entire SPECcpu2000 benchmark suite and show that table-based predictors outperform statistical predictors by as much as 69%, especially on benchmarks with high variability. One of the statistical predictors considered is a simple last value predictor, which models the behavior of a reactive system. The performance advantages of table-based predictors over last value prediction suggests that adaptive optimization needs to be *predictive* rather than reactive.

We conducted our experiments on two different micro-architectures — the IBM Power3 [22] and Power4 [28]. We have built the on-line predictors entirely in software using the hardware performance monitors (PMAPI) available on the AIX operating system. Our predictors sample the performance monitors at a granularity of 10 ms. 10 ms coincides with the operating system's context switch interval allowing the efficient incorporation of the predictors into the kernel. To investigate the pollution potential of context switching activity, we analyze the sensitivity of our predictors to multiprogramming effects.

While a similar analysis can be applied at finer granularities and/or in hardware, a software solution allows the immediate use of the predictors in resource allocation on existing hardware. For example, we are currently investigating the use of the on-line predictors to schedule processes on the two cores of an IBM Power4. The scheduling objective is to optimize the utilization of the shared L2 cache based on predicted memory behavior. Another advantage of a software prediction infrastructure is its flexibility. The software predictors can be easily adjusted to provide prediction at varying levels of granularity, thereby serving a larger class of applications.

We first discuss related work in Section 2. In Section 3, we provide a detailed analysis of the behavior of the SPECcpu2000 [12] benchmarks. In Section 4, we describe and evaluate our on-line behavior predictors and include a sensitivity analysis. Finally, we summarize our findings and discuss potential applications of the predictors in Section 5.

2. Related Work

Prior work can be categorized into methods that define and use metrics to dynamically identify phases for adaptive optimization [5, 6, 8, 13, 26], and into techniques that identify appropriate simulation points for the desired workloads [10, 16, 17, 24, 25].

Balasubramonian *et al.* [5, 6] use interval-based exploration mechanisms in order to dynamically reconfigure the underlying micro-architecture to meet the need of an application. They identify a change in program behavior based on instruction mix and/or memory behavior variations using an adjustable-length sampling interval [6], and an adjustable behavior difference threshold to trigger an exploration [5]. Dhodapkar and Smith [8] extend the algorithm in [5] to use instruction working set signatures to characterize application intervals. They use a table-based mechanism to store configurations from previous executions of intervals with the same signature, thereby reducing exploration overhead. All of the above work is reactive in that it assumes that future behavior will be similar to that just seen.

On the software side, phase changes have been exploited in dynamic optimization systems, such as Dynamo [4]. Dynamo detects hot code fragments, optimizes them, and stores them in a code cache. Phase shifts are detected as changes in the program's fragment working set by tracking the fragment creation rate.

Eeckhout *et al.* [10] identify program-input pairs with similar behavior in order to reduce the evaluation space for a design point. They use several whole program character-

istics as input to a principal component analysis (PCA) tool in order to determine a reduced set of orthogonal characteristics. The reduced set is passed to a hierarchical clustering analysis to determine which of the program-input pairs have similar behavior. Sherwood *et al.* [24, 25] use clustering at a finer grain to classify regions of program execution into phases, and use basic block vectors as a unique signature to characterize the phases. Algorithms in [5, 13] use subroutines to identify program phases and propose a hardware-based call stack to identify major program subroutines. As our analysis will show, a simpler characterization derived from existing hardware counters is sufficient for the purpose of on-line program behavior prediction. We avoid the need for dedicated hardware to capture basic blocks by using existing processor hardware counters to capture current execution behavior. This gives us the ability to use our predictions for both hardware and software optimizations.

Voldman *et al.* [29] provided an early study on the periodicity and other dynamic characteristics of cache miss behavior in complex workloads. Sherwood *et al.* [24] also showed that the periodicity has a direct relationship to the code being executed. They also explore the use of a run length encoded Markov model to predict the next phase in [26]. Rather than attempting to predict the next *phase*, we directly predict the next *value* for the metric of interest. As we will show in Sections 3 and 4, this helps us take into account the variation in behavior of each individual metric yielding high prediction accuracy.

3. Program Behavior Characterization

In this section, we characterize program behavior and its variability to set the stage for understanding and developing on-line predictors. Our infrastructure is based on actual program execution, allowing analysis of the entire run of the program. We use the Dynamic Probe Class Library (DPCL [9]) to insert a timer-based interrupt into the program to periodically collect hardware performance counter samples. The hardware counters are read using PMAPI (Performance Monitoring Application Programming Interface [20]), a low-level API provided as a kernel extension to AIX. The granularity at which the counters are read is 10 milliseconds (ms).

We present data on two machines with different micro-architectures — the IBM Power3 and the IBM Power4. They represent processors from different generations, providing program characterization across design points as well as different interval granularities. A 10 ms interval captures the aggregate behavior of many more instructions on Power4 than on Power3.

The Power3-based machine is a 200 MHz 64-bit out-of-order superscalar PowerPC architecture. It has a fetch and execute width of up to 8 instructions, a dispatch and completion width of up to 4 instructions, and up to 32 in-flight instructions. The memory hierarchy consists of a 32 KB on-chip L1 instruction cache, 64 KB on-chip L1 data cache, and a 16 MB L2 cache. Both instruction and data L1 caches are 128-way set associative with 128 byte lines.

The Power4-based machine is a Regatta, with a 1.3 GHz 64-bit out-of-order superscalar PowerPC architecture. Each chip consists of 2 processor cores, with a shared L2 cache. For our experiments, we use only a single core to run the applications. Each core has a fetch and execute width of up to 8 instructions, a dispatch and completion width of up to 5 instructions, and up to 200 in-flight instructions. The memory hierarchy consists of three levels of cache — a direct-mapped, 128 byte line, 64 KB L1 instruction cache and a 2-way, 128 byte line, 32 KB L1 data cache per core, a shared 8-way, 128 byte line, 1.5 MB L2 cache, and an off-chip 8-way, 512 byte line, 32 MB L3 cache.

We analyze the behavior of the entire SPECcpu2000 [12] benchmark suite (integer and floating point) using the *ref* data sets. Table 1 lists the programs, along with the average and standard deviation of their instructions per cycle (IPC) on both Power3 and Power4. We begin by examining the time-varying behavior of several metrics that can be derived from hardware counters — IPC, L1 D-cache misses per instruction, branches per instruction, and branch mispredict rate (mispredicts per branch). Figure 1 plots their values over time for the compression program *bzip2* on the Power3 architecture. The left column shows the time series over the entire execution (54,639 samples). The plots show two distinct macro phases that repeat across all metrics. These macro phases represent a sequence of compress and decompress operations on the input files at two different blocking levels. The right column in Figure 1 zooms into a specific interval of 10 secs (1000 samples) from the first macro phase.

The first observation that can be made from the plots is that the behavior of programs varies significantly with respect to each metric. For example, the average IPC in *bzip2* is 1.18 but the instantaneous IPC ranges from 0.2 to 3.4. While *bzip2* shows the same variation in the zoomed in plots, there are phases of relatively stable behavior between variations.

Figure 2(a) shows the behavior of *art*, a neural networks program used for image recognition. The figure demonstrates less of a visually identifiable pattern at a large time scale. In addition, *art* shows dramatically different behavior from *bzip2* at a small time scale. Figure 2(b) zooms into a one second interval. The behavior varies widely with a 0.21 average difference in IPC values from one sample to the next. This is the highest observed difference across all programs in our benchmark suite and is even higher than the standard deviation of the IPC (0.146).

Standard deviation has been used to characterize work-

SPECint	Power3		Power4		SPECfp	Power3		Power4	
	Avg IPC	Std IPC	Avg IPC	Std IPC		Avg IPC	Std IPC	Avg IPC	Std IPC
perlbmk	1.11	0.28	0.86	0.15	sixtrack	1.54	0.06	0.97	0.05
twolf	0.94	0.06	0.63	0.02	lucas	0.58	0.09	0.54	0.18
gap	1.35	0.26	0.83	0.19	swim	0.43	0.42	0.26	0.24
crafty	1.39	0.04	0.98	0.03	mesa	1.14	0.08	0.69	0.03
eon	1.20	0.05	0.76	0.03	apsi	0.85	0.70	0.62	0.43
mcf	0.34	0.23	0.22	0.13	applu	0.95	0.29	0.53	0.15
bzip2	1.18	0.56	1.10	0.44	wupwise	1.53	0.20	1.12	0.26
vpr	0.65	0.15	0.48	0.11	fma3d	0.99	0.35	0.69	0.23
parser	1.09	0.17	0.82	0.14	ammp	0.73	0.16	0.49	0.16
gcc	1.28	0.49	0.95	0.31	mgrid	1.74	0.36	1.21	0.23
vortex	1.58	0.31	1.30	0.17	facerec	1.03	0.41	0.81	0.11
gzip	1.13	0.26	0.83	0.12	galgel	0.80	0.51	0.68	0.26
					equake	1.30	0.35	0.73	0.08
					art	0.45	0.15	0.25	0.03
Overall mean IPC	Power3: 1.05				Power4: 0.74				

Table 1. Average and standard deviation of IPCs for the SPECcpu2000 benchmarks on Power3 and Power4.

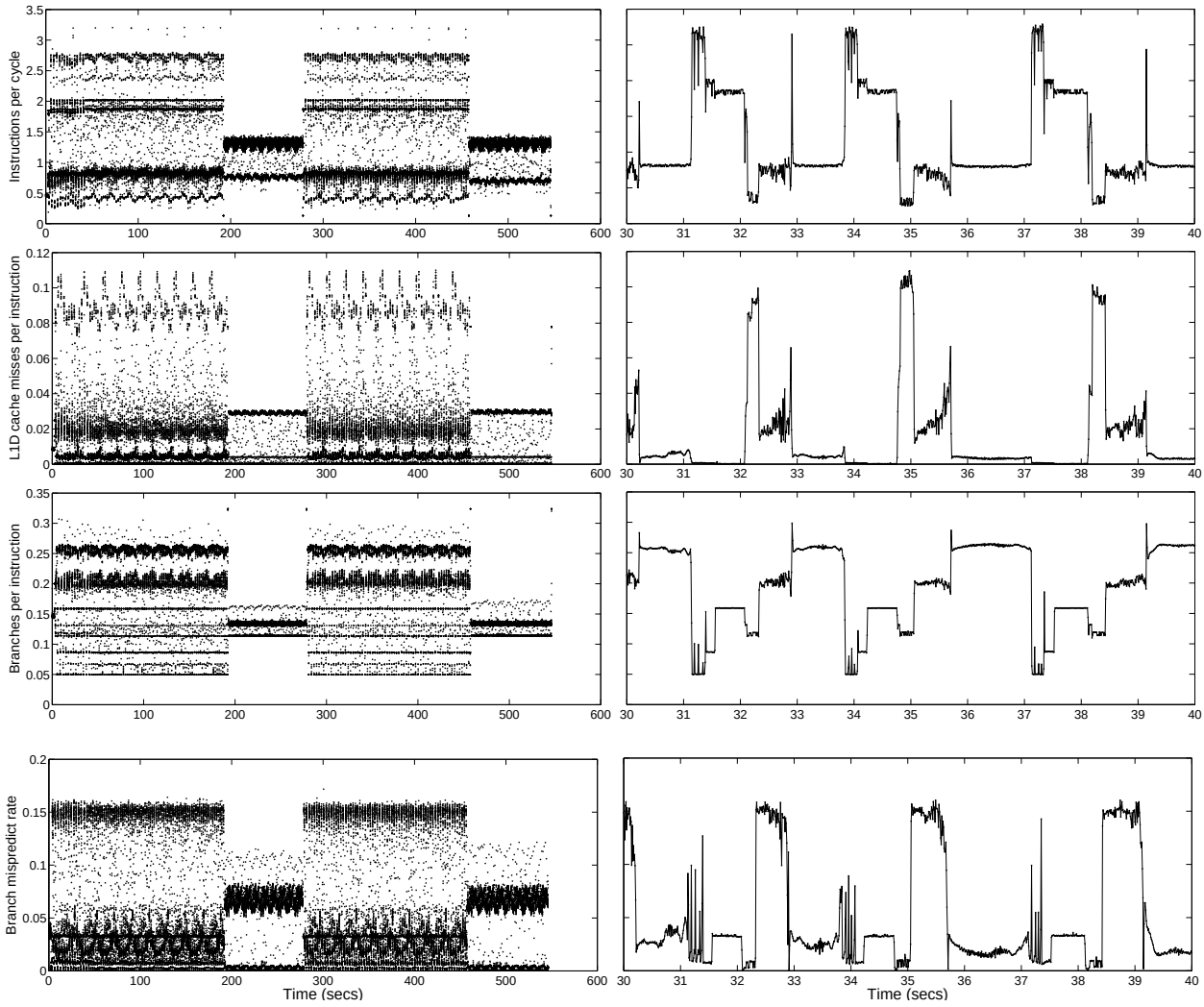
loads [14]. However, standard deviation is insensitive to the time of variation and thus does not reveal much about the rate of variability. We obtain a better measure of variability by considering the average absolute distance between adjacent points (mean deltas) in the time series. Figure 3 shows the mean deltas in IPC values between adjacent time intervals for each of the programs and for both the Power3 and the Power4 [1]. While we present the deltas only for IPC, similar behavior variability is seen across all metrics. Figure 3 shows the SPECint benchmarks followed by the SPECfp benchmarks, each in order of increasing IPC delta measured on the Power3. The figure illustrates the significant differences that exist in the variability of IPC behavior ranging from fairly flat behavior in *perlbmk* to widely varying behavior in *equake* and *art*. Note that Power3 and Power4 impose different orderings with respect to the mean IPC delta. *art*, which shows the highest mean delta on the Power3, has a much lower mean delta on the Power4. Conversely, *bzip2* shows a larger mean delta on the Power4. This behavior is primarily a result of averaging over a much larger number of instructions on the Power4 (For example, the execution time of *bzip2* is roughly 550 and 85 seconds, respectively, on the Power3 and Power4, for the same number of total instructions completed.). The change toward a coarse granularity causes a slight smoothing of the noisy behavior for *art* (i.e., lower delta), while the relatively stable behavior of *bzip2* on the Power3 transforms to more variable behavior on the Power4 as the program moves more quickly through regions of execution (i.e., higher delta).

Returning to the plots in Figure 1, our second observation is that program behavior is highly periodic. There is a clear, repeating pattern of behavior with a period of roughly 3 seconds. To analyze the periodicity of the program behavior, we applied Fourier analysis to compute the periodograms of the metrics across the entire execution of the program. Figure 4 shows the periodograms for the IPC time series in *bzip2* (left) and in *art* (right). Given the 10 ms sampling rate for the hardware counters, the range of frequency discernible is 0 to 50 cycles/sec (the Nyquist rate). For *bzip2* we plot only the periodogram from 0 to 5 cycles/sec since the rest of the periodogram is flat. The relative heights of the peaks in the periodogram represent the relative strength (power) of the corresponding frequency across the entire execution of the program. Note that this power is also a function of the fraction of the entire execution time for which the particular periodicity behavior was seen.

The period of three seconds observable in the *bzip2* plots in Figure 1(b) corresponds to the isolated peak in *bzip2*'s periodogram at 0.33 cycles/second. Similar periodic behavior, but at different frequencies, can be observed in each of the other macro phases in *bzip2*. In fact, the top four peaks in the frequency spectrum (after accounting for harmonics) can be attributed to each of the macro phases. Compared to *bzip2*, *art* has a very different periodogram. As shown in Figure 4, there is a cluster of peaks at much higher frequencies corresponding to the much shorter period depicted in Figure 2. We computed the periodograms for all benchmarks and found significant frequency peaks, although at varying frequencies, across the benchmarks.

While we can expect to find periodic behavior from the looping structure of programs, the fact that periodicity is still visually obvious at a granularity of millions of instructions is less intuitive. The period in *bzip2* of 3 seconds corresponds to several millions of instructions, and manifests

[1] In this paper, we use the arithmetic mean since we focus on the accuracy of performance prediction rather than on absolute performance



(a) (b)

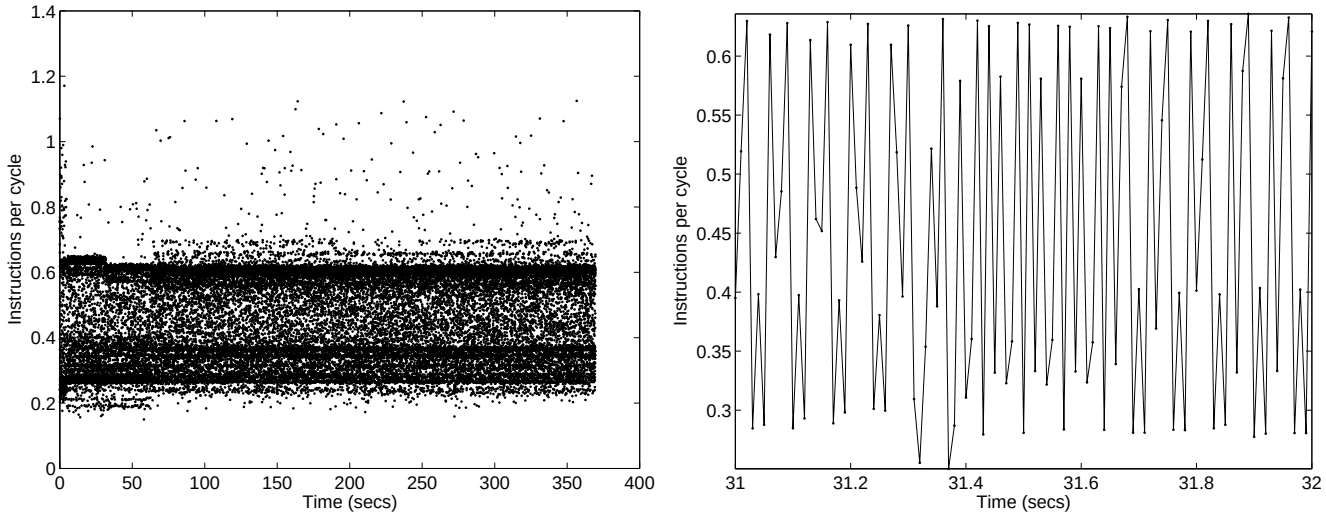
Figure 1. IPC, L1D cache miss rate, branch frequency, and branch mispredict rate for bzip2 (a) over the entire execution and (b) zoomed into a 10 second interval (30–40 secs).

itself with equal strength across all metrics.

This leads us to our third observation: the periodicity of metric behavior is shared across all metrics. Going back to Figure 1(b), we can clearly see the same 3 second period in each plot. However, the behavior within a period is widely different. During intervals where the behavior is almost flat in one metric (e.g., L1 D-cache misses) there are significant variations in others (e.g., branch mispredict rate). Thus, what makes the various plots look alike in Figure 1(a) is not the similarity in the metric behavior itself but the similarity in the underlying periodicity of the behavior.

To investigate this observation further, we compared the periodograms for the different metrics for each program. Subjectively, the periodograms for the same program across metrics look alike. In order to provide a more objective measure of their similarity, we matched the top 10 peak frequencies for each metric periodogram with all the peak frequencies in the IPC periodogram and vice versa. This way we obtain the percentage of successful matches for each program indicating how strongly the periodicity is related across metrics. While the relative amplitudes of the peaks (strength of the period) varied across metrics, the match across all metrics was on average 86%. For *art* and *bzip2*, in particular, they were 96% and 93%, respectively. The high matching rate indicates that the periodicity of behavior is shared across various metrics.

While we have presented detailed results on the Power3, the Power4 runs show similar trends in behavior, with some differences due to the fact that the sampling interval represents a much larger number of instructions.



(a) (b)

Figure 2. IPC for art (a) over the entire execution and (b) zoomed into a one second interval of 100 samples.

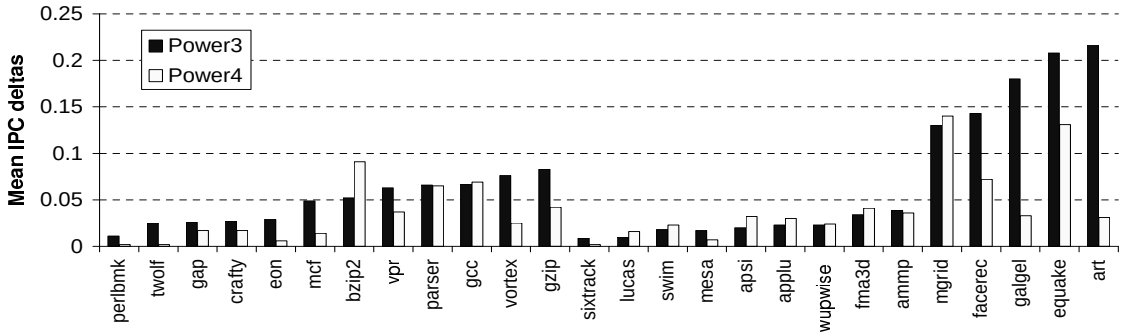


Figure 3. Average differences in IPC values between adjacent time intervals on Power3 and Power4.

In summary, our characterization found that program behavior has significant variability even at a coarse granularity (millions of instructions). We showed that program behavior tends to be periodic and that the periodicity of program behavior is shared across metrics. The high variability in the behavior for some programs implies that the problem of behavior prediction is non-trivial. Periodic behavior indicates that it should be possible to build effective predictors based on history. Finally, the similarity of periodicity across metrics suggests that a predictor may use the histories of one metric to predict the other metrics.

4. Program Behavior Prediction

In this section, we consider the design of several on-line predictors of program behavior. The predictors sample the program's behavior along various performance metrics and make predictions about the metric values for the next sampling interval. As in the previous section, we use a fixed sampling interval of 10 ms.

4.1. Simple Statistical Predictors

The most basic predictor we consider is a *Last Value predictor*. A Last Value predictor assumes that the program executes in a stable phase and that the current behavior will repeat. The prediction for the next interval is simply the last measured value. Last Value predictors provide an interesting baseline in that they reflect the behavior of reactive adaptive optimization systems. Thus, improvements over Last Value predictors provide an indication of the potential of incorporating prediction into adaptive optimization systems.

More sophisticated predictors use histories of metric val-

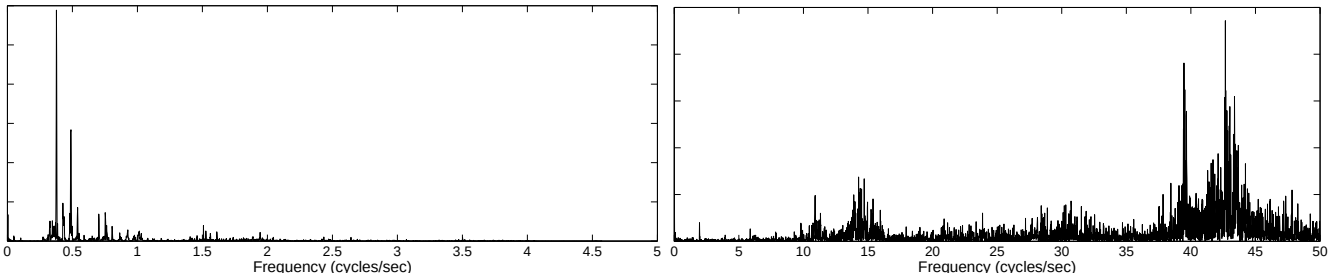


Figure 4. Periodogram for bzip2 (left) and art (right).

ues to smooth out noise and isolated peaks. For example, an *Average(N)* predictor chooses the average over the last *N* values and a *Mode(N)* predictor chooses the most frequently occurring value among the last *N* values. An exponentially weighted moving average (EWMA) predictor places more emphasis on the most recent data. An EWMA prediction for the $(n+1)$th value is computed as:

$$(1-\alpha)x_n + \alpha\,\overline{x}_{n-1},$$

where x_n is the n-th metric value, $\overline{x}_{n-1}$ is the EWMA over the first $n-1$ values, and $\alpha \leq 1$ is the filter constant that determines the relative weight of older values compared to more recent ones.

4.2. Table-Based History Predictors

We showed that programs exhibit varying degrees of periodicity, indicating that behavior patterns repeat over time. *Table-based history predictors* are designed to exploit this repeatability. A table-based predictor uses an encoding of past behavior history as an index into a prediction table. The prediction stored in the table is a value that was observed to immediately follow the behavior history in the past.

When exposed to a new workload, the predictors go through an initial learning phase to populate the prediction table with history patterns. To provide predictions during the learning phase, a fall-back prediction scheme, such as *Last Value*, may be used. The predictors adapt to changes in learned behavior by updating the table entries. However, care must be taken to avoid updating the table with spurious values from isolated noise peaks. We use a voting mechanism similar to the one used in [26]. In addition to the prediction value, each table entry stores the last actual value that was observed to follow the history. After the result of a prediction is known we have three values: the prediction and the last actual value stored in the table and the new actual value. The table is updated by setting the prediction value to the most frequent among the three values.

Table-based predictors vary in the encoding and length of the history used to index into the table. We consider two types of history encodings: fixed-size and run-length encoded history. A fixed-size history predictor (*History(N)*) uses the history of the past *N* values. Run-length encoded history predictors (*Run-length(N)*) use a history that is composed of the last *N* values, where each value is followed by the number of contiguous occurrences. Run-length predictors yield more efficient history encodings for programs with successive intervals of stable behavior. The reverse is true when considering highly variable programs, which are more efficiently captured using a History(N) predictor. A Run-length(1) predictor can be viewed as a hybrid between a Last Value predictor and a History(N) predictor.

The performance of a history predictor is highly sensitive to both the history length and the precision at which metric values are included in the history. The precision of history values determines the amount of aliasing in the prediction table. If the precision is too coarse, the predictor is insensitive to fine-grained behavior variations and will make the same predictions for behavior patterns that may vary significantly at finer granularities. Conversely, a predictor that is overly sensitive (too fine a precision) loses the ability to recognize repeating patterns in the presence of slight noise. Thus, choosing the appropriate history precision involves both an assessment of the amount of noise and of the desired level of prediction accuracy.

A similar trade-off applies to the history size parameter. For perfectly periodic behavior the ideal history size is the length of the period. Smaller history sizes may lead to less accurate prediction and larger history sizes require much longer learning periods. Realistic data will not be perfectly periodic and will include some amount of noise, such that a single behavior pattern may have many noise-induced history variations, each requiring a separate table entry. Larger history sizes not only require more table space, but also lead to many more entries being created, thus reducing the utilization of the table. We experimented with the various parameters and report the results in the next section.

4.3. Cross-Metric Predictors

We can exploit the fact that periodicity tends to be shared across metrics by using the history of one metric to predict another. We call a predictor that predicts values along one

metric (the target metric) using the history of a different metric (the source metric) a *cross-metric predictor*. A predictor with the same source and target metrics is called a *same-metric* predictor. An example of a cross-metric predictor is a predictor for IPC that uses the cache miss rate as the source metric.

The appeal of cross-metric predictors is the ability to efficiently combine multiple predictors. We can build a multi-predictor that uses the same history index to produce predictions along multiple metrics. To expand a cross-metric predictor into a multi-predictor merely requires additional prediction fields each table entry.

Care must be taken when choosing the source metric for a given target prediction metric because not all metrics are equally effective as a source predictor. Ideally, the source and target metrics should have similar variability. Choosing a flat source metric (e.g., L2 cache misses on a small SPECint program) to predict a highly variable one (e.g., IPC) is likely to result in poor prediction quality. On the other hand, a source metric that is much more variable than the target may require a longer than necessary learning interval.

We also consider a cross-metric predictor that uses the instruction mix (I-mix) as a microarchitecturally independent source metric. I-mix combines branch frequency and load/store frequency histories to capture the major changes in control and data instruction frequency during program execution. To produce an *I-mix(M/N)* index we concatenate M values from the branch frequency history with N values from the load/store instruction history. An I-mix predictor is particularly attractive when the predictions are used for optimizations that affect performance or power related behavior since the history used to index into the table is unaffected.

4.4. Experimental Parameters

We present results for the following five predictors:
– Last Value
– EWMA filter (α=0.2)
– Run-length(1) encoded table-based predictor
– History(4) predictor
– Cross-metric I-mix(2/2) predictor

We selected the above predictors and their parameters because they demonstrated the best performance among predictors in their class. We do not show data for *Average* or *Mode* predictors since the EWMA predictor performed consistently better. After experimenting with histories of various sizes, we found that a history of size 4 provided sufficient pattern sensitivity while still tolerating noise. We encode IPC, L1 D-cache misses, and branch mispredict rate values in the histories at a precision of .01 (i.e., up to two digits after the decimal point). We experimentally verified that this precision provides the best trade-off between sensitivity to noise and prediction accuracy. Since a combined history using multiple sources provides more history variability than a single-source history, we found that shorter histories for the I-mix predictors were more effective at striking a balance among noise tolerance, learning period, and prediction accuracy.

4.5 Evaluation

We evaluate the predictors by measuring the mean absolute prediction error, that is, the mean absolute distance between the predicted and the actual value. Whether a certain mean error is considered high or low depends entirely on the application that uses the predictor. For an application that uses the predictor to trigger mostly short-lived adaptations, a low absolute error is important. On the other hand, an application that performs more long-lasting adaptations may be satisfied with a much looser prediction accuracy. To evaluate the predictors independent of an application we use the mean absolute prediction error primarily as a relative measure and give no further meaning to it other than re-stating it as a percentage of the mean value of the metric being predicted.

IBM Power3 IPC Prediction Accuracy: We first consider IPC as the target prediction metric on a Power3. Figure 5 shows the mean absolute prediction error for the five predictors. The mean IPC value across all benchmarks is 1.05. Thus, an absolute prediction error of 0.05 corresponds to 5% of the mean IPC. In order to expose the challenge in the prediction problem, we show the SPECint benchmarks followed by the SPECfp benchmarks, each in order of increasing mean IPC deltas. The mean deltas are shown again below each program name.

As expected, when considering programs with low variability, such as *perlbmk* through *eon* in SPECint, and *sixtrack* through *ammp* in SPECfp, all predictors perform very well with only negligible performance differences. However, when considering programs with larger variability, significant differences in prediction performance emerge. For *mcf* and *gzip* in SPECint and for *facerec*, *galgel*, *equake*, and *art* in SPECfp, the table-based history predictors, History(4) and I-mix(2/2), clearly outperform the other predictors. They have a significantly lower mean error than the simple statistical predictors. The prediction error is reduced by up to 69% compared to last value predictors. The Run-length(1) table-based predictor, while beating the simpler statistical predictors, gains only a fraction of the benefits of fixed-size history predictors. Run-length encoding becomes less effective when there is high variability since there is little stability in behavior values. A remarkable result of Figure 5 is that using the cross-metric I-mix history predictor is nearly as effective as using a same-metric IPC predictor.

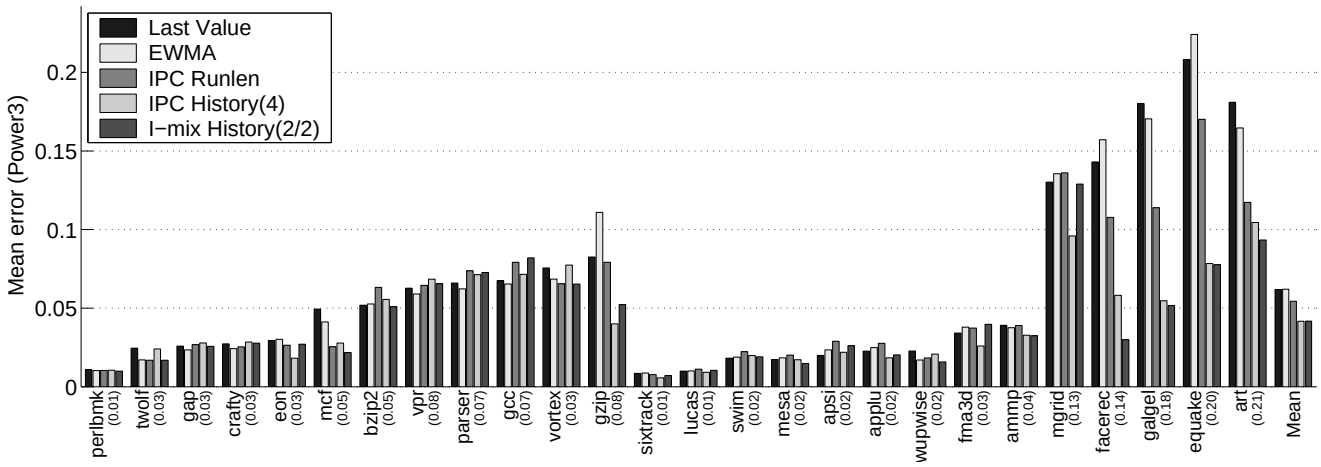


Figure 5. Mean absolute error for predicting IPC on a Power3. A mean error of .05 corresponds to 5% of the mean IPC across all benchmarks. IPC deltas are shown below each program name.

The benchmark *art* shows an absolute prediction error that is higher than that of most other programs even for the table-based predictors. Figure 2(b) shows that the periodicity is close to the sampling rate. Thus, any statistical variation may result in an incorrect prediction. Prediction accuracy may improve by sampling at a finer granularity but we were limited to 10 ms by our infrastructure. It remains to be seen if matching the sampling rate to the periodicity in the application affects the accuracy of the table-based predictors.

IBM Power3 Prediction Accuracy for Other Metrics: We evaluate the predictors for predicting other metrics: L1 D-cache misses per instruction and branch mispredict rate. The branch mispredict rate results are similar to the IPC results. Therefore, we only discuss the L1 D-cache miss prediction results. Figure 6 shows the mean absolute prediction error for L1 D-cache misses per instruction. We again show the performance of Last Value and EWMA predictors and include two same-metric table-based predictors: a Run-length(1) predictor and a History(4) predictor. We replaced the cross-metric I-mix predictor with a different cross-metric predictor: an IPC-based History(4) predictor.

The programs that have the highest L1 D-cache miss rate variability are different from the ones with the highest IPC variability: *mcf*, *parser*, *gcc*, *gzip*, *facerec*, and *art* are the ones with the highest deltas. While the table-based L1 D-cache History(4) predictor follows the previous trend of outperforming the other predictors in *mcf*, *gzip*, *facerec*, and *art*, it actually performs worse in *gcc*, and slightly worse in a few others. We inspected the time-varying L1 D-cache miss behavior plots for these programs and found that they are considerably more noisy and irregular than for IPC. Noise makes the periodic patterns in the metric much harder to capture. Noise is also the primary reason why the cross-metric IPC History(4) predictor lags behind the performance of its same-metric counterpart (L1 D-cache History(4)) in several programs. In spite of these noise effects, the results of L1 D-cache miss rate prediction follow previous trends when considering the mean across all benchmarks.

IBM Power4 IPC Prediction Accuracy: We collected results for IPC prediction on a Power4 in order to determine whether the relative accuracy of the different predictors holds across microarchitectures and across different sampling intervals. Results for cache misses per instruction and branch mispredict rates are similar to the Power3 data and are not shown.

As pointed out in Section 3, the time varying behavior, while generally flatter than on the Power3, still shows significant variability on the Power4. Also, the relative degree of IPC variability across the benchmarks is not the same as for the Power3. Figure 7 shows the prediction error obtained for predicting IPC. The predictors in Figure 7 follow the same trend as for the Power3: the table-based predictors tend to outperform the others when IPC variability is high, most notably for *mgrid* and *equake*, the programs with the highest variability.

Effect of Finite Table Sizes: The results presented above are based on prediction tables implemented as hash tables of unlimited size. To determine the loss of prediction accuracy when limiting the table size, we repeated the IPC prediction experiment from Figure 5 and compared the mean prediction error across the entire benchmark suite for the predictors with unlimited table size to the mean prediction error when using various table size limits. Limiting the table to 4000 and 1000 direct-mapped entries increases the mean IPC prediction error by 10% and 25%, respectively. The range of possible values in IPC tends to be larger com-

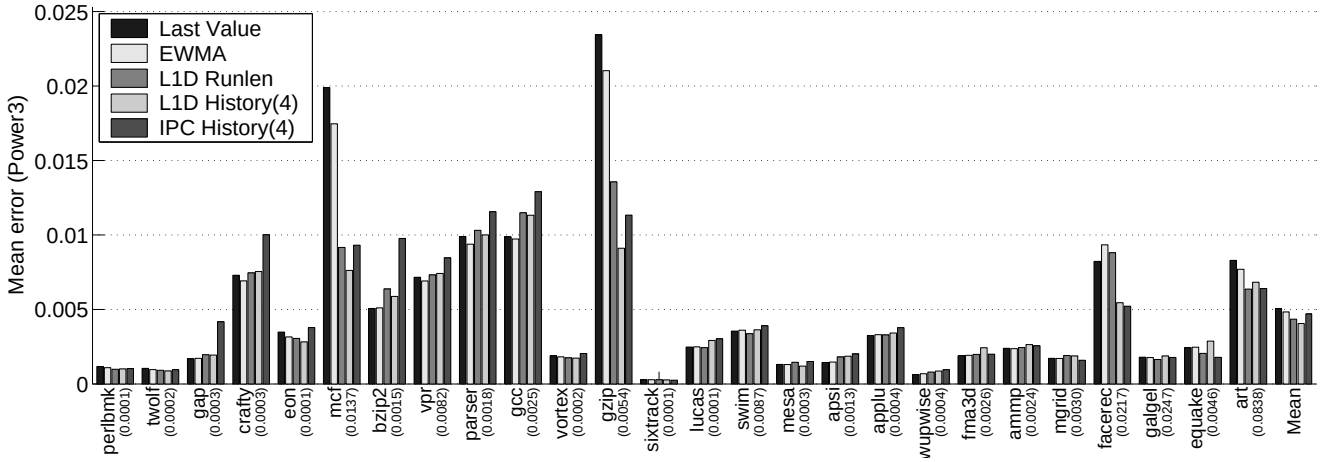


Figure 6. Mean absolute error for predicting L1 D-cache misses per instruction on an IBM Power3. The mean L1 D-cache misses per instruction across all benchmarks is 0.04 so that a mean error of 0.005 corresponds to 12.5% of mean. The variability of L1 D-cache misses is shown next to each benchmark name.

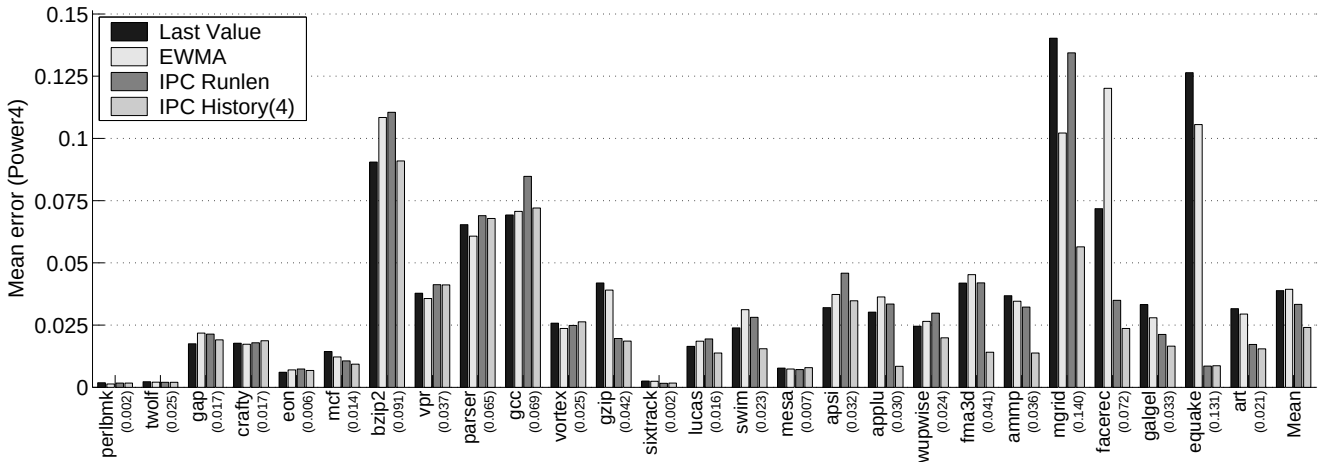


Figure 7. Mean absolute error for predicting IPC on an IBM Power4. An absolute mean error of .025 corresponds to 3% of the IPC mean across all benchmarks. IPC deltas are shown below each program name.

pared to the value range in L1 D-cache misses or branch mispredict rate. Correspondingly, the number of distinct histories in L1 D-cache miss and branch mispredict rate is much smaller resulting in less table size pressure. For L1 D-cache miss prediction, limiting the table to 1000 entries increases the mean prediction error by less than 15%. For predicting the branch mispredict rate, the table can be limited to only 512 entries with less than 2% increase in prediction error. Generally, as the table size is reduced further, the history predictors gradually deteriorate to the performance of a Last Value predictor.

In summary, the experiments highlight a number of important results:

– Program behavior variability along various performance metrics is highly predictable to within a few percent of the mean metric value.

– In programs with non-trivial variability, table-based fixed history predictors outperform any of the other predictors, while providing similar prediction accuracy for programs with relatively flat metric behavior. For IPC prediction on a Power3, the per application mean error is within 10% of mean IPC, compared to 20% for the Last Value predictor. On Power4, the per application error is within 9% of the mean IPC compared to 14% for the Last Value predictor.

– Cross-metric table-based predictors are nearly as effective as their same-metric counterparts, making it possible to predict multiple metrics through a single predictor.

– Microarchitecture-independent metrics built using ex-

isting hardware counters work well as a source metric, while providing stable behavior in the table even when the predicted target metric changes due to dynamic optimizations.

4.6. Sensitivity to Measurement Noise

All the predictors discussed are built entirely in software although a hardware implementation is feasible. Using a software infrastructure enables us to collect realistic behavior measurements of complete runs of the application. However, unlike in a simulation environment, actual measurements are necessarily exposed to a certain amount of sampling noise. In order to assess the degree of noise in our execution environment, we conducted two experiments.

In our first experiment, we investigated the pollution effects of context switching in a multiprogrammed environment. The performance monitor interface (PMAPI) virtualizes the hardware counters and provides per process counters. We expect that context switching, which occurs at exactly our sampling interval (10 ms), causes initial perturbation at the beginning of each sampling interval. To determine the effect of multiprogramming, we forced high context switching activity by running two programs simultaneously with equal priorities. We examined the IPC History(4) predictor on the Power3 and selected two of the most difficult to predict programs: *art* and *equake*. We ran *art* and *equake* simultaneously and compared the prediction results to the original undisturbed runs. Multiprogramming perturbed execution behavior as expected, causing an increase in the number of samples by 8% (*art*) and 10% (*equake*). We also observed changes in the mean IPC and its standard deviation by up to 9%. When comparing the prediction results from the multiprogramming runs to the original results from Figure 5, we noticed a slight increase in the mean error: .03 in *art* and .02 in *equake*, which correspond to 6% of *art*'s and 1.5% of *equake*'s average IPC value, respectively. Thus, while multiprogramming does cause disturbances in the sampled metric data, the loss in prediction accuracy is small.

The second experiment examined the effect of system daemons on the collected performance monitor data. Although we specify each sampling interval to be of equal length, the activity of system daemons causes variations in the number of cycles that are captured in each sampling interval. Variation in cycle count across samples makes it impossible to compare data collected from different runs at a sample-by-sample granularity. Since we convert the measured counter values into rates, we do not expect variations in cycle count to disturb the data within one run. To verify this assumption, we investigated cycle count variation in more detail. The mean cycle count per sample across all applications on the Power3 is 1.35 million cycles with a large standard deviation of 50%. We filtered out any sample with a cycle count that differed from the mean by more than 15%. We then compared the prediction results of the filtered data with the original data and found a difference in mean error of less than 0.5%, indicating that even significant cycle count variation does not cause noticeable pollution of the sampled metric data.

5. Conclusions and Future Work

We have presented a characterization of program behavior and its predictability on two different processor generations and micro-architecture design points. Our analysis demonstrates significant behavior variation across metrics, as well as periodicity even at a granularity of millions of instructions. We exploit these observations in the design of on-line behavior predictors, and use them to explain the relative performance of simple statistical as well as table-based predictors. Our evaluation shows that table-based predictors can cope with behavior variability much better than the simple statistical predictors. When using the table-based predictors the per application prediction error is within 10% and 9% of the mean IPC across all benchmarks in the SPECcpu2000 suite on Power3 and Power4, respectively. The performance advantages of table-based predictors over a simple Last Value predictor point to the potential of incorporating prediction into adaptive optimization systems.

We also demonstrate that different metrics show similar periodic behavior. Based on this result we introduced the use of cross-metric predictors that use behavior histories of one metric to predict others, making it possible to efficiently couple multiple predictions into a single predictor. Finally, we explored the use of a cross-metric predictor that uses an estimation of the program's instruction mix to make predictions. The use of such an architecture-independent metric for prediction provides a stable reference even when the underlying micro-architecture or operating environment is reconfigured.

We are currently investigating the use of the on-line predictors for resource-aware scheduling. One application is to optimize the kernel-level scheduling of processes on the two cores of an IBM Power4. The scheduling objective is to optimize the utilization of the shared L2 cache. We can use the L1 miss rates to predict L2 cache traffic and capacity requirements. Traffic predictions can be used to avoid scheduling processes with high traffic in the same interval.

Other applications of the predictor that we plan to investigate include the dynamic management of configurable hardware for both performance and power efficiency, and dynamic voltage and frequency scaling of a multiple clock domain processor for power efficiency.

Acknowledgements

We would like to thank Bonnie Ray for numerous discussions and Liudvikas Bukys for his support.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Prog., Sys., Lang., and Apps., OOPSLA'00*, Oct. 2000.

[2] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Prog., Sys., Lang., and Apps., OOPSLA'02*, Nov. 2002.

[3] I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proc. of the 28th Int. Symp. on Computer Arch., ISCA-28*, Jul. 2001.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. In *Proc. of the ACM SIGPLAN Conf. on Prog. Design and Impl., PLDI*, Jun. 2000.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proc. of the 33rd Int. Symp. on Microarchitecture, MICRO-33*, Dec. 2000.

[6] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proc. of the 30th Int. Symp. on Computer Arch., ISCA-30*, Jun. 2003.

[7] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proc. of the 1st ACM Int. Symp. on Memory Management, ISMM'98*, Jun. 1998.

[8] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proc. of the 29th Int. Symp. on Computer Arch., ISCA-29*, May 2002.

[9] *The Dynamic Probe Class Library (DPCL)*. http://oss.software.ibm.com/developerworks/opensource/dpcl/doc.

[10] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: Selecting representative program-input pairs. In *Int. Conf. on Parallel Arch. and Compilation Tech., PACT'02*, Sep. 2002.

[11] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proc. of the 28th Int. Symp. on Computer Arch., ISCA-28*, Jul. 2001.

[12] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE COMPUTER*, July 2000. http://www.spec.org/cpu2000.

[13] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proc. of the 30th Int. Symp. on Computer Arch., ISCA-30*, Jun. 2003.

[14] C. Hughes, P. Kaul, S. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architectures. In *Proc. of the 28th Int. Symp. on Computer Arch., ISCA'01*, Jun. 2001.

[15] C. Hughes, J. Srinivasan, and S. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proc. of the 34th Int. Symp. on Microarchitecture, MICRO-34*, Dec. 2001.

[16] J. J.W. Haskins and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *Proc. of the 2003 IEEE Int. Symp. on Performance Analysis of Sys. and Software*, Mar. 2003.

[17] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications*. Kluwer Academic Publishers, Sep. 2000.

[18] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain processor. In *Proc. of the 30th Int. Symp. on Computer Arch., ISCA-30*, Jun. 2003.

[19] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and W. Hwu. An architectural framework for run-time optimization. *IEEE Trans. on Computers*, 50(6), Jun. 2001.

[20] *Performance Monitor Application Programming Interface*. http://www.cs.utk.edu/ browne/dod/cewes/pmapi.html.

[21] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proc. of the 34th Int. Symp. on Microarchitecture, MICRO-34*, Dec. 2001.

[22] POWER3: Next generation 64-bit PowerPC processor design. http://www.ibm.com/servers/eserver/pseries/hardware/whitepapers/power3wp.html.

[23] F. Qian and L. Hendren. An adpative, region-based allocator for Java. In *Proc. of the ACM Int. Symp. on Memory Management, ISMM'02*, Jun. 2002.

[24] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Int. Conf. on Parallel Arch. and Compilation Tech., PACT'01*, Sep. 2001.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys., ASPLOS-X*, 2002.

[26] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Int. Symp. on Computer Arch., ISCA-30*, Jun. 2003.

[27] I. Subramania, C. Mather, K. Peterson, and B. Raghunath. Implementation of multiple pagesize support in HP-UX. In *Proc. of the 1998 USENIX Technical Conf.*, Jun. 1998.

[28] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Res. and Dev.*, 46(1), 2002.

[29] J. Voldman and L. W. Hoevel. The software-cache connection. *IBM J. Res. and Dev.*, 25(6), Nov. 1981.

[30] S. Winwood, Y. Shuf, and H. Franke. Mulitple page size support in the Linux kernel. In *Proc. of the 4th Annual Ottawa Linux Symp., OLS 2002*, Jun. 2002.

[31] S. Yang, M. Powell, B. Falsafi, K. Roy, and T. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep Submicron High-Performance I-Caches. In *Proc. of the 7th Int. Symp. on High-Perf. Computer Arch., HPCA-7*, Jan 2001.