Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management

Canturk Isci, Gilberto Contreras and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{canturk,gcontrer,mrm}@princeton.edu

Abstract

Computer architecture has experienced a major paradigm shift from focusing only on raw performance to considering power-performance efficiency as the defining factor of the emerging systems. Along with this shift has come increased interest in workload characterization. This interest fuels two closely related areas of research. First, various studies explore the properties of workload variations and develop methods to identify and track different execution behavior, commonly referred to as "phase analysis". Second, a large complementary set of research studies dynamic, on-the-fly system management techniques that can adaptively respond to these differences in application behavior. Both of these lines of work have produced very interesting and widely useful results. Thus far, however, there exists only a weak link between these conceptually related areas, especially for real-system studies.

Our work aims to strengthen this link by demonstrating a real-system implementation of a runtime phase predictor that works cooperatively with on-the-fly dynamic management. We describe a fully-functional deployed system that performs accurate phase predictions on running applications. The key insight of our approach is to draw from prior branch predictor designs to create a phase history table that guides predictions. To demonstrate the value of our approach, we implement a prototype system that uses it to guide dynamic voltage and frequency scaling. Our runtime phase prediction methodology achieves above 90% prediction accuracies for many of the experimented benchmarks. For highly variable applications, our approach can reduce mispredictions by more than 6X over commonly-used statistical approaches. Dynamic frequency and voltage scaling, when guided by our runtime phase predictor, achieves energy-delay product improvements as high as 34% for benchmarks with non-negligible variability, on average 7% better than previous methods and 18% better than a baseline unmanaged system.

1 Introduction

The increasing complexity and power demand of processors mandate aggressive dynamic power management techniques that can adaptively tune processor execution to the needs of running applications. These techniques extensively benefit from application "phase" information that can pinpoint execution regions with different characteristics. Recognizing these phases on-the-fly enables various dynamic optimizations such as hardware reconfigurations, dynamic voltage and frequency scaling (DVFS), thermal management and dynamic hotcode optimizations [1, 4, 5, 7, 11, 15, 25, 28].

In recent years, studies have demonstrated different approaches to track application phase behavior, relying on execution properties such as control flow [7, 10, 15, 20, 22, 23] or power/performance characteristics [3, 5, 6, 12, 13, 26, 27]. While such studies provide useful insights to application behavior, most of these mainly focus on characterizations of application behavior, summarizing application execution or detecting repetitive execution phases. Few of these studies [8, 14, 21, 24, 30] also seek to predict future application behavior. However, to be able to utilize different phase characterizations effectively on a running system, there is need for a general dynamic phase prediction framework that can be seamlessly employed on-the-fly during workload execution. Moreover, it is essential to provide a useful, readily-available binding between application phase monitoring and prediction, and dynamic, adaptive management opportunities, especially on real system implementations.

In this work, we describe a fully-automated, dynamic phase prediction infrastructure deployed on a running Pentium-M based system. We show that a *Global Phase History Table* (GPHT) predictor, leveraged from a common branch predictor technique, achieves superior prediction accuracies compared to other approaches. Our GPHT predictor performs accurate on-the-fly phase predictions for running applications without any offline profiling information or any static or dynamic modifications to application execution flow, and with no visible overheads.

We demonstrate how our runtime phase predictors can effectively guide dynamic, on-the-fly processor power management using DVFS as the underlying example dynamic power management technique [9]. Our dynamic phase predictor efficiently cooperates with a DVFS interface to adjust processor execution on-the-fly for improved power/performance efficiency. This GPHT-based dynamic power management can improve the energy-delay product (EDP) in our deployed experimental system by more than 15%. We also demonstrate that our methodology can be used under different phase definitions that can be aimed at serving different purposes such as bounding execution with performance degradation limits. We evaluate our methods on the SPEC2000 benchmark suite, with runtime monitoring using performance monitoring counters (PMCs), and real power measurements with a data acquisition (DAQ) unit.

There are three primary contributions of this work. First, we present and evaluate a live, runtime phase prediction methodology that can seamlessly operate on a real system with no observable overheads. Second, we demonstrate a complete real-system implementation on a deployed system. Our implementation can autonomously function during native operation of the processor, without any profiling or static instrumentation of applications. Nor does it require any underlying virtual machine or dynamic compilation support. Third, we demon-

strate the application of our phase prediction infrastructure to dynamic power management using DVFS as an example technique. Although we present our work with specific phase definitions and power management techniques, our runtime phase prediction is a general framework. It can be applied to any feasible definition of application phases and to other dynamic management techniques, such as dynamic thermal management or bounding power consumption.

The rest of the paper is organized as follows. Section 2 describes our phase definitions. Section 3 presents our runtime phase prediction methodology and prediction results. Section 4 discusses the dependence of phase characterizations to dynamic management actions. Section 5 describes our prototype real-system implementation. Section 6 provides our dynamic management results. Section 7 summarizes related work and Section 8 offers our conclusions.

2 Defining Phases

The key motivation for our work is to develop a phase prediction technique that can be accurately applied at runtime application execution to guide dynamic power management. We describe and evaluate our proposed prediction technique in detail in Section 3. However, first, in this section, we explain our phase classification methodology, which we also use in the evaluations of Section 3.

The fundamental purpose of phase characterization is to classify application execution into similar regions of operation. This classification can be done via various features, depending on the ease of monitoring and the goal of the applied phase analysis. Similarly, how the observed features are classified into different phases depends on the target application.

In our work, to track application behavior, we rely on hardware performance monitoring counters (PMCs), which can be configured to monitor execution without disrupting execution flow. For system-level dynamic management, we define relatively coarse grained phases, on the order of millions of instructions. This guarantees that monitoring of application behavior—and dynamic management responses—do not lead to any observable overheads. Our phase classifications are constrained by two factors. First, our experimental platform, described in greater detail in Section 5 supports simultaneous monitoring of 2 PMCs. Therefore, our classifications of application behavior can be based on only two configured counters. Second, we monitor PMCs from within a performance monitoring interrupt (PMI) routine. Therefore, we need a simple classification method to avoid violating interrupt timing constraints as well as to have negligible performance overheads. In addition, one of the counters has to be dedicated to monitor *micro-ops (Uops) retired*, to trigger the PMI at specified instruction granularities.

We refer to prior work for our choice of monitored PMC events. Wu et al. [28] make use of event counter information to assign application routines to different DVFS settings under a dynamic instrumentation framework [19]. They define the ratio of *memory bus transactions* to *Uops retired* as the measure of the "memory-boundedness" of an execution region, and use the ratio of *Uops retired* to *instructions retired* as a proxy to represent available "concurrent execution" in the same region. These two metrics then determine the available "CPU slack" in the application, which guides different DVFS settings. For our experiments, we configure the remaining independent counter to track memory bus transactions. Thus, the ratio of the memory bus transactions to our Uop granularity represents the memory-boundedness of each observed phase. We refer to this measure as "Mem/Uop" in the rest of the paper.

In addition to Mem/Uop, the two configured counters, together with the time stamp counter (TSC), also enable simultaneous monitoring of Uops per cycle (UPC), which can provide additional information on application behavior. These two metrics have already been used cooperatively in other previous studies to guide dynamic power management [27]. However, for phase prediction to perform robustly under dynamic management, our phase classifications have to be resilient to the effects of dynamic management actions. As we demonstrate in Section 4, while Mem/Uop behavior is virtually invariant to the responses of our dynamic management technique, UPC can fluctuate strongly. Therefore, for a simple, yet robust phase classification that is largely invariant under dynamic power management, we use Mem/Uop to define application phases.

We classify Mem/Uop into different phases by observing how different Mem/Uop rates are assigned to different DVFS settings in Wu et al.'s description [28]. That work examines memory access rates and concurrency of different applications on a similar experimental platform. Then, it calculates the DVFS settings for different application regions based on a performance loss formulation. For our phase definitions, we convert these statistics to Mem/Uop rates and available concurrency ranges for each DVFS setting. As we do not have the concurrency measure available for our runtime monitoring and prediction, we base our phase classifications to the derived Mem/Uop ranges for the common lowest observed concurrency—i.e. $Uops\ retired/instructions\ retired \approx 1$. Based on this classification, we define 6 phase categories as shown in Table 1. Conceptually, category 1 corresponds to a highly CPU-bound execution pattern that should be run as fast as possible, and category 6 corresponds to a highly memory-bound phase, where the application can be significantly slowed down to exploit available slack.

Mem/Uop	Phase #
< 0.005	1 (highly cpu-bound)
[0.005,0.010)	2
[0.010,0.015)	3
[0.015,0.020)	4
[0.020,0.030)	5
> 0.030	6 (highly memory-bound)

Table 1. Definition of phases based on Mem/Uop rates.

3 Predicting Phases with a Global Phase History Table Predictor

In this section, we first discuss different prediction options and describe our chosen technique. Afterwards, we present our phase prediction evaluations. For a phase prediction technique that can perform well on all corners of benchmark behavior, we propose a *Global Phase History Table* (GPHT) predictor. There exist other prior history based predictors that

also target at estimating application performance characteristics [8, 14]. However, predictors that simply rely on the statistics of past behavior cannot perform well for highly variable benchmarks. To demonstrate this comparatively, we also consider some of the simple statistical predictors in our evaluations.

The simplest statistical predictor is the *last value* predictor. In this predictor, the next sample behavior of an application is assumed to be identical to its last seen behavior. In this case, predicted phase in the next interval can be expressed as $Phase[t+1] = Phase[t]$. This can be extended to encompass longer past histories by considering a *fixed history window* predictor, where the predictions are based on the last *window size* observations. In this case, the next phase prediction can be phrased as $Phase[t+1] = f(Phase[t], Phase[t-1], ..., Phase[t-(winsize-1)])$. The function $f()$ can be a simple averaging function, an exponential moving average or a selector, based on population counts. Another approach, similar to fixed history window is a *variable history window* predictor. In this case, the history can be shrunk in case of a phase transition, where previous history becomes obsolete for the following phase predictions.

In contrast, our *Global Phase History Table* (GPHT) predictor observes the *patterns* from previous samples to deduce the next phase behavior. In such an approach, it relies on the widely acknowledged repetitive execution behavior of applications. Structurally, the GPHT predictor, depicted in Figure 1, is similar to a global branch history predictor [29]. Unlike hardware branch predictors, however, the GPHT is a software technique, implemented in the operating system for high-level, dynamic phase prediction.

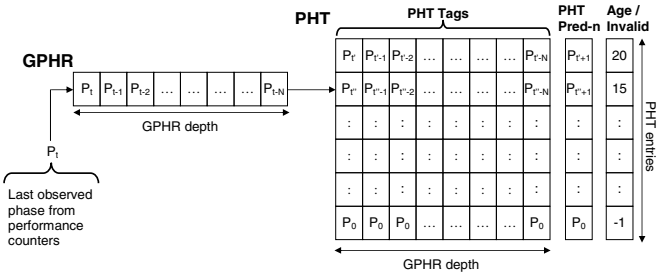

Figure 1. GPHT predictor structure.

Similar to a global branch predictor, a GPHT predictor consists of a global shift register, called the *Global Phase History Register* (GPHR), that tracks the last few observed phases. The length of the history is specified by *GPHR depth*. At each sampling period, the GPHR is updated with the last seen phase, as observed from PMCs. This updated GPHR content is used to index a *Pattern History Table* (PHT). The PHT holds several previously observed phase patterns, with their corresponding "next phase" predictions based on previous experience. These phase predictions are shown as the *PHT Pred-n* vector in the PHT. The GPHR index is associatively compared to the stored valid PHT tags, and if a match is found, the corresponding PHT prediction is used as the final prediction. An *Age / Invalid* entry is kept for each tag to track the ages of different PHT tags for a least recently used (LRU) replacement policy when the PHT is full. A -1 entry denotes the corresponding tag contents and prediction are not valid. The number of entries in the PHT is specified by *PHT entries*. In the case of a *mismatch* between GPHR and PHT tags, the last observed phase, stored in GPHR[0], is predicted as the next phase. After a mismatch, the current GPHR contents are added to the PHT by either replacing the oldest entry or by occupying an available invalid entry. In the case of a match, a PHT prediction entry is updated in the next sampling period based on the actual observed phase for the corresponding tag.

By observing the phase patterns in application execution, the GPHT predictor can perform reliable predictions even for highly variable benchmarks. Inevitably, for a hypothetical application with no evident recurrent behavior, no predictor can perform good predictions. In such cases there is no matching pattern in PHT and we revert to a last value predictor, thus guaranteeing to meet the accuracy of previous methods under worst case scenarios. Most applications exhibit some amount of repetitive patterns, however, due to the common loop-oriented and procedural execution style.

We demonstrate an example to how GPHT accurately captures varying application behavior with the `applu` benchmark. `Applu` shows a highly varying behavior, with distinctive repetitive phases throughout its execution. In Figure 2, we show the variation in Mem/Uop for `applu` and its corresponding phases (shown as $\square$) from a sample execution region. We show the performed phase predictions with both GPHT (shown as $\circ$) and last value (shown as $\diamond$) predictors. For GPHT, we choose a GPHR depth of 8 and 1024 PHT entries. This example shows, even for this highly variable application, GPHT predictions almost perfectly match the actual observed phases. On the other hand, last value mispredicts more than one third of the phases due to `applu's` rapidly varying phases. In Figure 2, we highlight two regions, showing the repetitive phase behavior and how GPHT can easily capture this behavior. In addition we show two distinct cases, where GPHT first mispredicts next phase at point labeled *"A"*, and later can correctly predict similar behavior at point *"B"* by learning from previous pattern history. This example shows the clear strength of pattern based phase prediction with GPHT over statistical approaches.

3.1 Predictability and Power Saving Potentials of Different Applications

To assess the quality of a phase prediction scheme for an application, it is imperative to understand the predictability characteristics of the application. For example, for a very stable application, with very few changes in its phase behavior, a simple predictor that assumes the last observed behavior will continue, will be highly accurate. However, benchmarks with high variability, where the observed phases rapidly change, will experience many mispredictions with such an approach. Therefore, before evaluating our phase prediction method, here we first discuss the predictability of different benchmarks based on their stability characteristics.

In Figure 3, we show the characteristics of different benchmarks in two dimensions. In the *y* dimension we show the variability of benchmarks, based on the observed variation in Mem/Uop. We represent this as the percentage of time Mem/Uop changes more than 0.005 between two samples for a 100 million instruction sampling granularity. Thus, this dimension shows how "unstable" the benchmark is. Benchmarks higher along the y axis represent cases with temporally varying behavior, which cannot be predicted in a straightforward manner, simply by assuming the benchmark will pre-

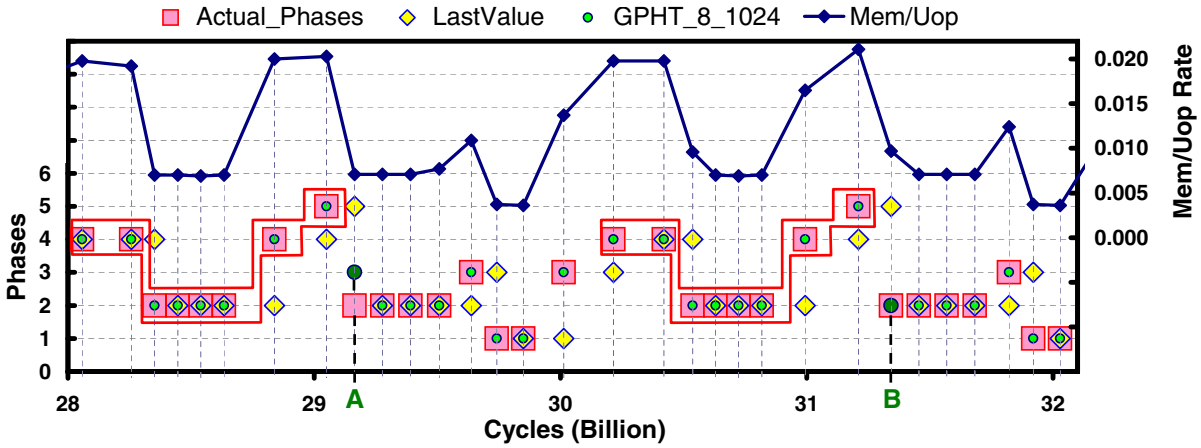

Figure 2. Actual and predicted phases for applu benchmark.

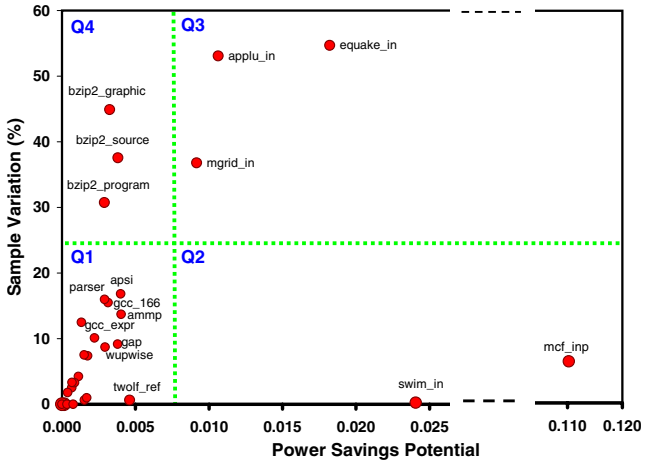

Figure 3. Benchmark categories based on stability and power saving potentials.

serve its last observed behavior. On the other hand, benchmarks close to the x axis show almost completely "flat" execution behavior, where the application rarely changes its execution properties. In these cases, simply assuming the previous observed characteristics will prevail performs as well as any other method. In addition to these variability characteristics, in the *x* dimension of the figure, we also show the average Mem/Uop rate for our applications. This shows how much potential exists to slow down the CPU frequency for each application. Thus, benchmarks further to the right exhibit higher power savings potentials. Note that many of the SPEC applications lie very close to the origin, showing small variations and power saving opportunities. We do not label these in the figure to avoid cluttering the image.

Based on these observed properties we categorize the experimented benchmarks into four quadrants. $Q1$ benchmarks, which include many of the SPEC applications, are very stable and show little power saving opportunities. $Q2$ benchmarks show higher power saving potentials and little variability. These two categories are easily predictable with simple phase predictors. $Q3$ benchmarks—that also include our `applu` example—are the most interesting applications for our research. These have both highly varying phase behavior and high power saving potentials. $Q4$ benchmarks also show high variability, but show relatively smaller power saving opportunities. Both $Q3$ and $Q4$ applications are not expected to perform well under a simple phase prediction strategy that assumes the next phase behavior will match the previously observed one.

3.2 Phase Prediction Results

In Figure 4, we show the achieved prediction accuracies for all discussed prediction methods and all experimented SPEC applications. Specifically, we show the results with last value prediction, fixed window prediction with window sizes of 8 and 128, variable window with 128 entry window and phase transition thresholds of 0.005 and 0.030, and GPHT with GPHR depth of 8 and 1024 PHT entries. The benchmarks sorted in the order of decreasing prediction accuracy with last value prediction. For most of the $Q1$ and $Q2$ benchmarks, almost all approaches perform very well, achieving above 80% prediction accuracies. For these mostly stable applications, last value and GPHT perform almost equivalently. However, the benefits of GPHT are immediately observed with the last 6 benchmarks, which constitute the $Q3$ and $Q4$ applications. In these more variable benchmarks, the statistical approaches experience significant drops in prediction accuracies, while GPHT can still sustain higher prediction accuracies by observing repetitive phase patterns. For our `applu` example, the last value predictor—the best among statistical predictors for this application—results in more than 53% mispredictions. In comparison, GPHT achieves less than 8% mispredictions, improving phase mispredictions by more than $6X$. On average, for the $Q3$ and $Q4$ benchmarks, our GPHT predictor leads to $2.4X$ less mispredictions than the experimented statistical predictors.

This evaluation clearly demonstrates that our proposed GPHT predictor performs effectively in all corners of our benchmark categories. In the remainder of our work, we build our dynamic power management framework upon this phase prediction methodology. However, for a real-system implementation, holding and and associatively searching through a 1024 entry PHT may be undesirable. Therefore, in Figure 5, we show how GPHT prediction accuracy changes with different number of PHT entries. As the figure shows, down to 128 entries, GPHT predictor performs almost identically to the 1024 entry predictor. However, observable degradations in accuracy are seen with a 64 entry PHT. As PHT entries are reduced down to 1, the accuracy of the GPHT predictor con-

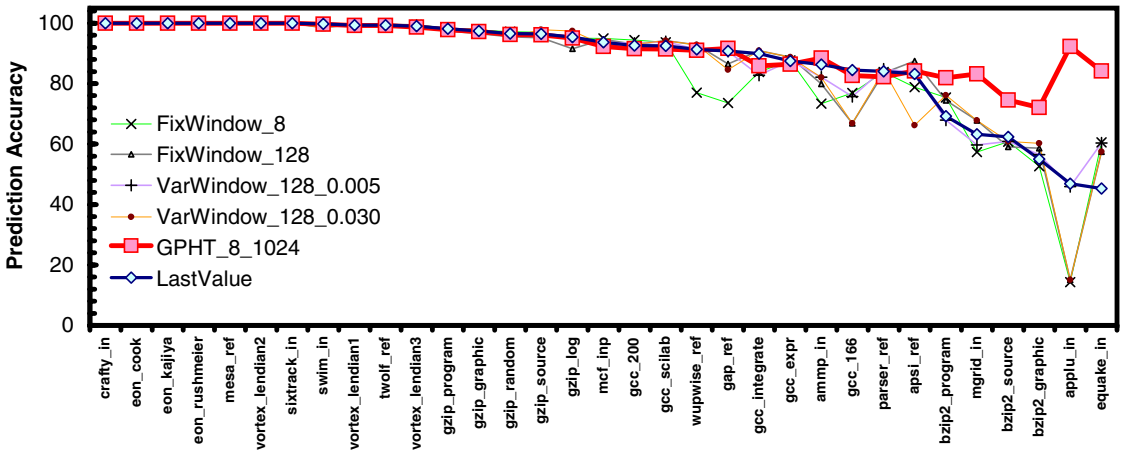

Figure 4. Phase prediction accuracies for experimented prediction techniques.

verges to last value, as we realize almost 100% tag mismatches and so the next phase is continuously predicted as the last seen phase from GPHR[0]. Therefore, we conclude that a 128 entry PHT is sufficient for our GPHT implementation. In our deployed real system, described in the following sections, we use this configuration for our final GPHT predictor implementation.

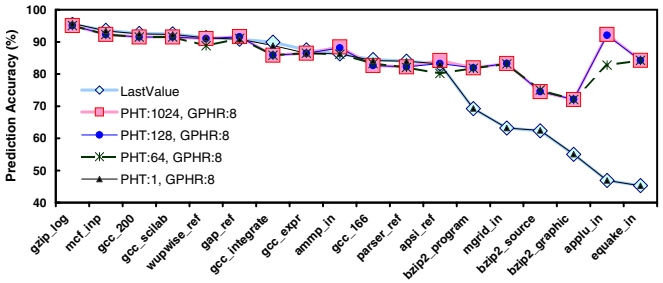

Figure 5. GPHT prediction accuracy for different number of PHT entries.

4 Dependence of Phases to Dynamic Management Actions

For our phase prediction methodology to be useful in a dynamic management framework, phase patterns must not be altered by the dynamic management actions that respond to them. Such action-dependent phases both conceal actual phase patterns, impairing predictability of application behavior and also lead to incorrect management decisions. Previously, we have mentioned that our phase definition based on memory bus transactions per micro-op (Mem/Uop) is resilient to changes in processor voltage and frequency settings. In this section we justify this claim with detailed measurements.

For our verifications, we consider the two metrics obtainable with our choice of monitored PMC events, Mem/Uop and Uops per cycle (UPC). To guarantee our conclusions are valid on all possible corners of execution, we profile all our experimented applications with performance counters (PMCs) and record different observed $(UPC, Mem/Uop)$ pairs for our two dimensional execution behavior space. In Figure 6, we show the corresponding exploration space for all acquired $(UPC, Mem/Uop)$ sample pairs for all set of applications with the lighter data points. These show the wide range of operating points that are covered by these applications. In addition, a boundary is observed as the maximum achievable UPC for each Mem/Uop level, depicted with the "SPEC Boundary" curve. This is an expected effect, as high memory latencies slow down dependent execution. Consequently, more memory bound applications can retire less instructions per cycle. To evaluate how the UPC and Mem/Uop metrics change under different DVFS settings, we develop a suite of configurable applications that can pinpoint specific $(UPC, Mem/Uop)$ coordinates in our two dimensional exploration space. We call these applications the *"IPCxMEM suite"*. The grid points, denoted as "IPCxMEM Grid" in Figure 6, show how we configure IPCxMEM suite to cover the whole exploration space. With these applications, we evaluate the behavior of our tracked metrics at all possible corners of execution and evaluate how these are affected by DVFS actions.

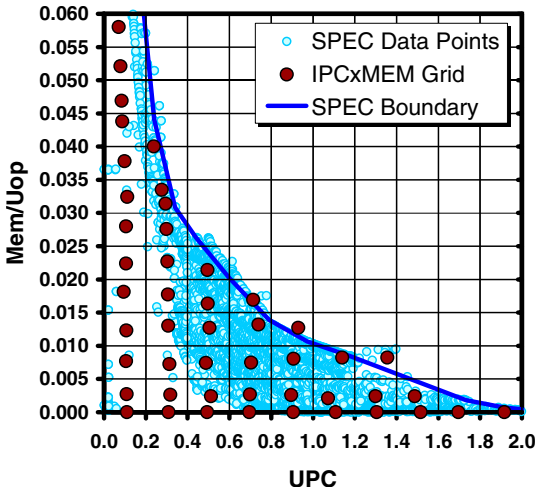

Figure 6. Observed (UPC,Mem/Uop) pairs for all experimented applications and grid of points covered by our IPCxMEM suite.

For our evaluations, we run the IPCxMEM suite in approximately 50 $(UPC, Mem/Uop)$ configurations, covering the exploration space grid. We run all these configurations in all the available frequency settings of our experimental platform. These are 1500MHz, 1400MHz, 1200MHz, 1000MHz, 800MHz and 600MHz. We monitor UPC and Mem/Uop via

PMCs in these frequency settings and demonstrate their frequency dependence for few chosen configurations in Figure 7. In Figure 7, each curve corresponds to a specific IPCxMEM suite application—run at all frequency settings—configured to target a specific UPC and Mem/Uop at the highest frequency. These target values, referenced in the legend, correspond to the specific points of the IPCxMEM grid in Figure 6.

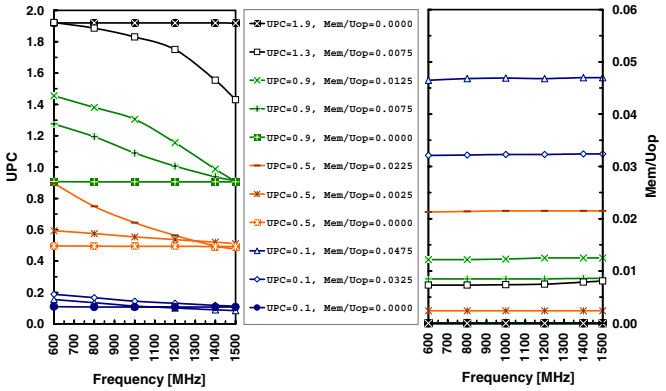

Figure 7. Observed UPC and Mem/Uop behavior at six different frequencies for different IPCxMEM grid configurations.

Figure 7 shows the strong dependence of UPC to DVFS settings. UPC mostly has an increasing trend with decreasing frequency. This is because memory latencies are not scaled with DVFS and therefore memory accesses complete in fewer CPU cycles at lower frequencies. The frequency dependence of UPC also varies with memory intensity. UPC values for completely CPU-bound configurations (legend entries with $Mem/Uop = 0$) show no dependence to frequency. On the other hand, for highly memory-bound configurations, UPC can change up to 80% across frequencies. These demonstrate the dangerous pitfall we avoid in our phase definitions. Directly using UPC in phase classification is not reliable for dynamic management, as the resulting phases vary with different power management settings.

Conversely, Figure 7 shows that the Mem/Uop parameter has virtually no dependence to DVFS settings. It is almost constant across all frequencies. Therefore, our phase classifications based on Mem/Uop are completely "DVFS invariant" and can be reliably used for runtime phase prediction under dynamic power management actions.

While not in the scope of this project, there are previous studies, which explore predicting performance behavior across different power management settings [16, 17]. These strategies can potentially be integrated to our runtime phase monitoring and prediction framework to employ more elaborate phase definitions in future dynamic management studies.

5 Real-System Implementation: Experimental Measurement and Evaluation Platform

We implemented our on-the-fly phase monitoring and prediction framework on a Pentium-M based, off-the-shelf laptop computer, running Linux kernel 2.6-11. Our prototype implementation monitors application execution via performance counters (PMCs) and performs phase predictions at fixed instruction intervals of execution in a performance monitoring interrupt (PMI) handler. We use our runtime phase predictions to guide dynamic voltage and frequency scaling (DVFS), readily available on our Pentium-M platform, as the example management application. At each interrupt invocation, after performing the next phase prediction with the GPHT predictor, the interrupt routine translates the predicted phase into a predefined DVFS setting. This setting is then applied to the processor for the next execution interval. Figure 8 shows a simplified overview of how this overall implementation operates on our system. After the initial configuration (performed once at system startup) all phase prediction and dynamic management actions operate autonomously, with no observable overheads to user applications. All applications can run natively, without any modifications or additional system or dynamic compiler support.

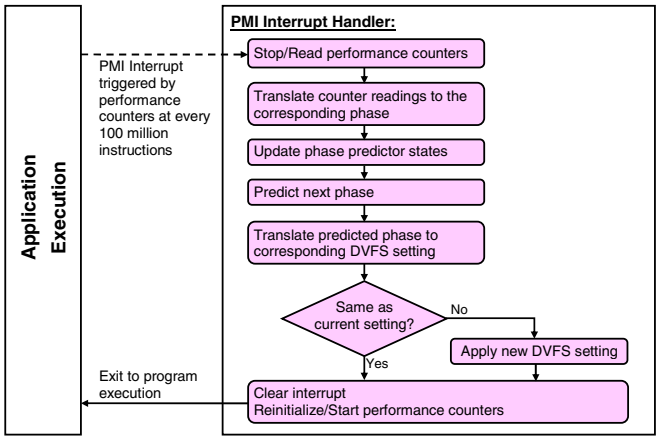

Figure 8. The flow of operation for our runtime phase prediction and dynamic power management framework.

In Figure 9, we show the overall prototype implementation and measurement setup for our experiments. This diagram depicts different aspects of our implementation that correspond to on-the-fly phase monitoring and prediction, dynamic power management via DVFS and additional mechanisms for evaluating runtime phase prediction and performing real power measurements that can match each phase. In the following subsections, we discuss the details of each of these aspects for our prototype platform.

5.1 Runtime Phase Monitoring and Prediction

One of the fundamental challenges of phase detection and prediction on a real system is the impact of system induced variability. Previous research has shown that application phases are prone to several variations at runtime [13], which can alter the timing and values of observed metrics. To eliminate the effect of timing variations, we monitor phases at fixed instruction granularities with the PMI. We have implemented our PMI handler and supporting system calls as a loadable kernel module (LKM), which can be loaded and unloaded during system operation. The implemented LKM also holds the state for our predictors and a log of PMC values, predicted and actual observed phases for our evaluations.

For our experiments, we configured the two available PMCs in the Pentium-M processor to monitor the retired micro-ops and memory bus accesses, with the

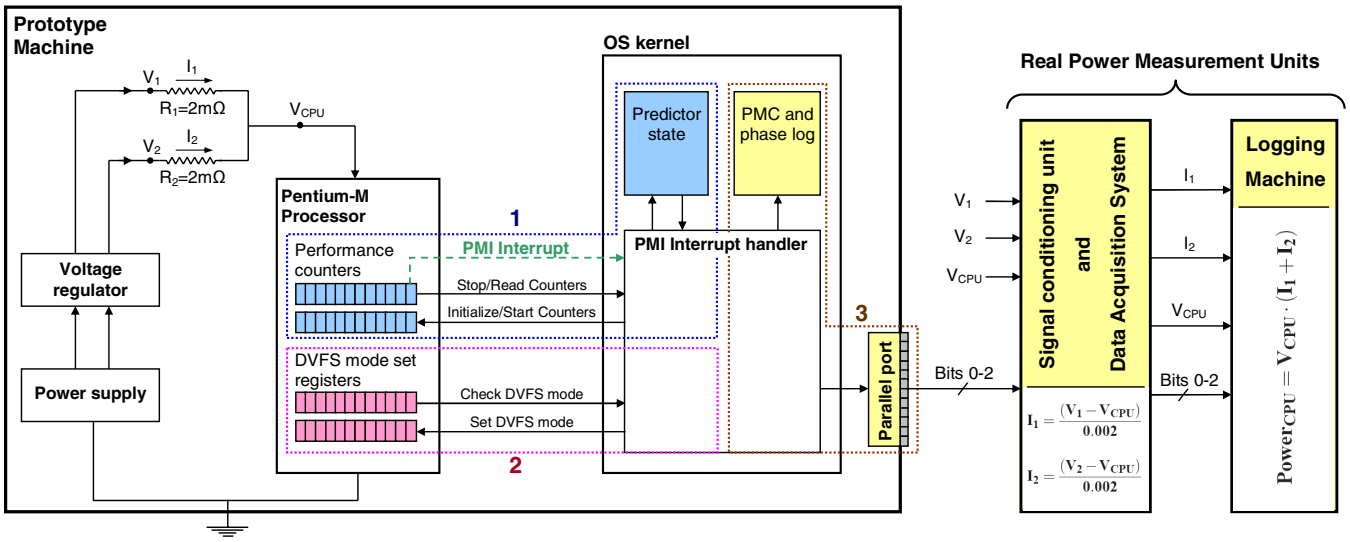

Figure 9. Developed measurement and evaluation platform. Regions identified as *1*, *2* and *3* via dashed lines correspond to different parts of implementation relevant to on-the-fly phase monitoring and prediction(*1*), dynamic management with DVFS(*2*) and measurement and evaluation support(*3*).

UOPS_RETIRED and BUS_TRAN_MEM event configurations. We have experimented with various instruction granularities and used 100 million instructions as a safe granularity to invoke our interrupt handler. Therefore, after each invocation, the first PMC is reinitialized to overflow after 100 million retired Uops.

After each 100 million instructions, the interrupt handler stops and reads the PMCs, updates the GPHT predictor states and performs the next phase prediction. It also logs the observed PMC values, actual observed phase for the past period and the predicted phase for the next period for our evaluations. At its exit, the handler clears the PMC overflow bit, reinitializes the PMCs and time stamp counter (TSC) and restarts the counters.

5.2 Dynamic Power Management via DVFS

Our on-the-fly phase prediction methodology can be utilized to guide a range of dynamic management techniques. In this work we consider DVFS as an example implementation, which is supported on our platform via Intel SpeedStep technology [9]. In our prototype implementation, we use a look-up table, defined at LKM initialization, to quickly translate the predicted phase to one of the 6 DVFS settings within the handler. For our prototype machine and our original phase definitions, these are shown in Table 2. For alternative phase definitions or management schemes, we can simply reconfigure this table. At each sampling interval, the handler translates the predicted phase to the corresponding DVFS setting. Then it compares this to the current setting and updates the DVFS mode set registers if necessary. Our 100 million instruction granularity (on the order of 100 ms) guarantees that the overheads induced by interrupt handling and DVFS application (on the order of 10-100 μs) are essentially invisible to native application execution.

5.3 Power Measurement

To track the power consumed by the Pentium-M processor, we measure the input voltage and current flow to the processor. For this purpose, we use an external data acquisition

Mem/Uop	Phase #	DVFS Setting
< 0.005	1	(1500 MHz, 1484 mV)
[0.005,0.010)	2	(1400 MHz, 1452 mV)
[0.010,0.015)	3	(1200 MHz, 1356 mV)
[0.015,0.020)	4	(1000 MHz, 1228 mV)
[0.020,0.030)	5	(800 MHz, 1116 mV)
> 0.030	6	(600 MHz, 956 mV)

Table 2. Translation of phases to DVFS settings.

system (DAQ) that is connected to the processor board. Our laptop board includes two 2 $m\Omega$ *precision sense resistors* that reside between the voltage regulator module and the Pentium-M CPU, shown as $R1$ and $R2$ in Figure 9. The total current that flows through these resistors represents the current flow into the CPU. The voltage after the resistors, denoted as V_{CPU}, represents the input voltage of the CPU.

In the actual measurement setup, we measure the three voltages V_1, V_2 and V_{CPU}, to track processor current and voltage. These voltages—and additional parallel port bits for evaluation support—are first fed into a National Instruments AI05 *Signal Conditioning Unit*. This unit filters the noise on the measured voltage signals and calculates the voltage drop across the two resistors. These voltage drops, $(V_1 - V_{CPU})$ and $(V_2 - V_{CPU})$ and the CPU voltage V_{CPU} are then fed into a National Instruments DAQPad 6070E *Data Acquisition System*. This unit then scales the voltage drops with the resistor values to compute the current flows as $I_1 = (V_1 - V_{CPU})/0.002$ and $I_2 = (V_2 - V_{CPU})/0.002$. The DAQ system monitors a total of eight signals, and has a sampling period of 40 μs. The two measured currents and CPU voltage, together with additional parallel port signals are sent to a separate *logging machine*, which logs the observed currents and voltages. The CPU power consumption for each sample is computed on this logging machine as $Power_{CPU} = V_{CPU} \cdot (I_1 + I_2)$. With this complete measurement setup, we can accurately track CPU power consumption. By also utilizing parallel port signaling, described below, our measurement setup can individually

compute the power consumption and performance statistics for each 100M-instruction phase sample, as well as for the whole execution of applications.

5.4 Evaluation Support

The overall correct operation of our developed system requires only on-the-fly phase monitoring and prediction, and dynamic power management with DVFS as highlighted in regions 1 and 2 in Figure 9. However, for experimental evaluation of phase prediction and dynamic management methods, we develop additional components on our prototype system. First, we use the above described real power measurement setup to measure processor power consumption. In addition, for detailed power/performance and phase prediction evaluations, we employ additional mechanisms in our implementation; these fall into region 3 in Figure 9.

To evaluate runtime phase prediction accuracy and to analyze application behavior, we use a separate kernel log in our LKM. This log keeps track of the actual observed and predicted phases for each sample as well as memory accesses per Uop and Uops per cycle for each phase. At each invocation, the handler logs relevant information in this log. Afterwards, a user-level tool can access this information via separate system calls.

The execution of the processor—under our runtime phase prediction and dynamic management—and the real power measurements are inherently two completely independent processes. To provide a synchronizing link between the two sides of our framework, we use parallel port bits that signal specific processor execution information to the DAQ system. We use three parallel port bits. *Bit 2* is set from the user level via system calls at the start of an application execution and is cleared when an application ends. This helps DAQ to measure power specifically during an application execution. *Bit 1* is used to distinguish between application and interrupt execution. This bit is set by the handler at the entrance to the handler routine and is cleared at exit. Finally, *bit 0* is used to help the DAQ track each phase. The handler flips this bit at each sampling interval so that the DAQ and the logging machine can distinguish each phase and compute power and performance statistics individually for each phase.

In Figure 10, we show a detailed view of the overall operation of our deployed system with the `applu` benchmark, performing on-the-fly phase predictions with the GPHT predictor and dynamic power management with DVFS. The figure shows the measured prediction, power and performance results with respect to a baseline, unmanaged system. In the top chart, we first show the observed Mem/Uop behavior for the two runs of `applu`, with and without our described techniques. The two curves are almost identical between the two real-system runs, which shows *(i)* our phases, defined by Mem/Uop, are DVFS invariant and can be safely used for phase prediction under dynamic management responses; and *(ii)* our fixed instruction granularity phase definitions are resilient to real-system variations. In the lower part of the top chart, we show the actual and predicted phases with the GPHT. Once again, the GPHT exhibits very good prediction accuracies for such a highly varying application. The middle chart shows the power measured for each phase during `applu` execution for both systems, where the shaded area between the two curves demonstrates power savings achieved with our approach. In the lower chart, we show the observed performance as billions of instructions per second (BIPS) for the two systems, where the shaded area demonstrates the relatively small performance degradation induced by our framework. These latter two charts clearly present the advantages brought by our framework for improving power/performance efficiency. By efficiently adapting processor execution to varying application behavior, we achieve significant power savings with small degradations in performance.

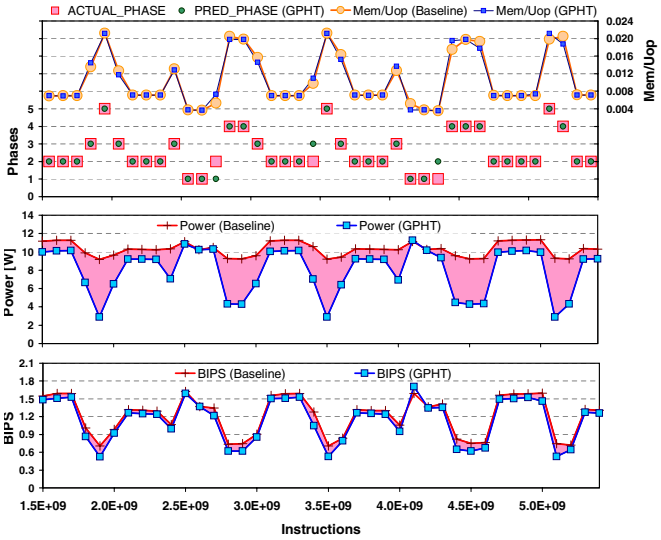

Figure 10. Overall operation of our framework, shown with applu benchmark, in comparison to the baseline system. Top chart shows the observed Mem/Uop and predicted phases. Middle and lower charts show achieved power savings and induced performance degradation in the shaded regions.

6 Experimental Results

In the previous sections, we described our phase definitions and on-the-fly phase prediction methodology, and presented a full-fledged deployed system. In this section, we evaluate the final target of our complete framework, dynamic power management with DVFS, guided by on-the-fly, GPHT-based phase predictions.

6.1 Dynamic Power Management Guided by Runtime Phase Prediction

Here, we present overall dynamic power management results for all experimented benchmarks with three sets of information. Figure 11 depicts obtained power and performance results with our experimental system, using the GPHT predictor, as normalized to baseline execution. From top to bottom, the figure shows achieved BIPS, power and energy-delay product (EDP) for the baseline unmanaged system and our dynamic management framework. The benchmarks are shown in decreasing EDP order with GPHT-based management.

The application categories we have discussed in Section 3 also guide our observations with the dynamic power management results. Many of the $Q1$ benchmarks experience little power savings and performance degradations. They exhibit highly stable, non-varying, execution behavior with little power saving potentials and close to baseline performance un-

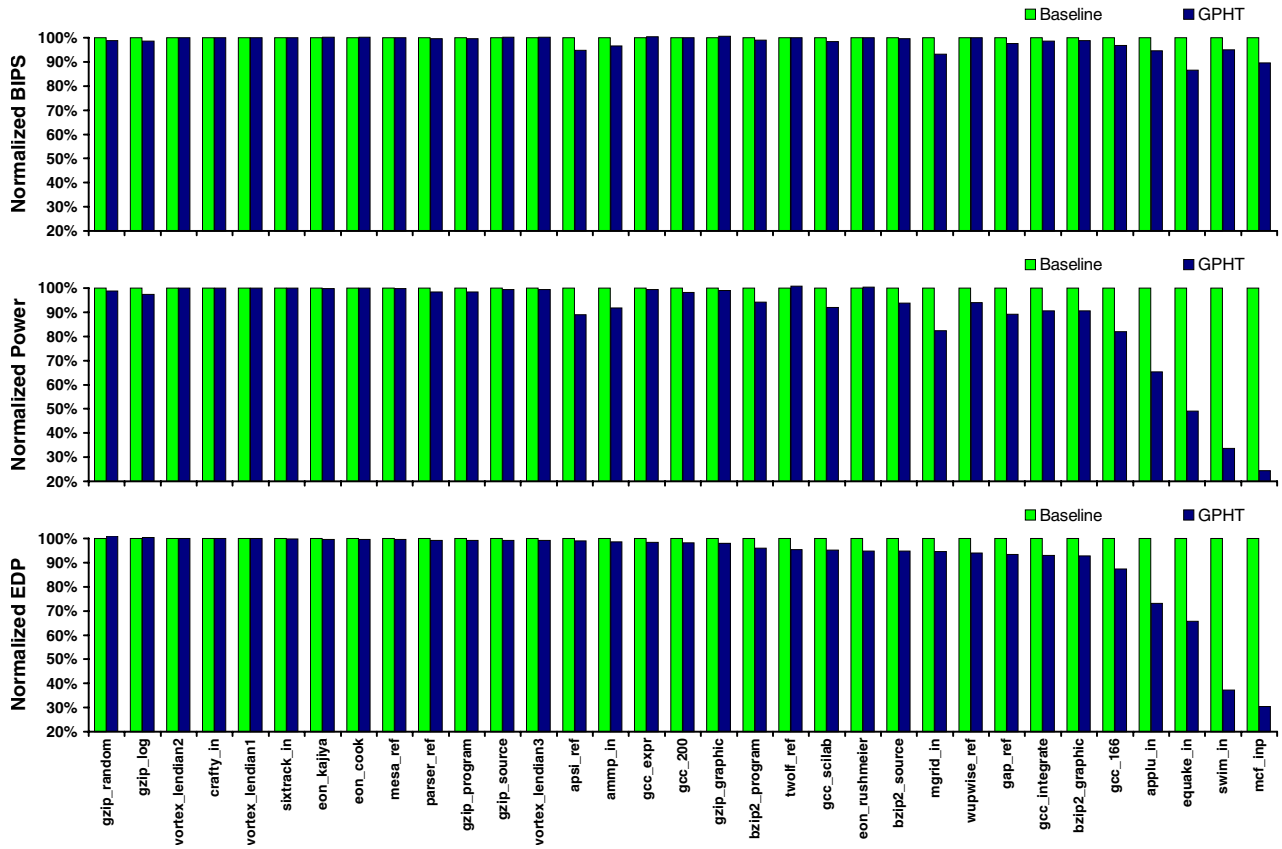

Figure 11. Runtime phase prediction guided dynamic power management results. From top to bottom, the charts show performance, power and energy delay product achieved by our framework with respect to baseline execution.

der dynamic management. Few of the $Q1$ applications, such as `apsi` and `ammp`, actually achieve significant power savings due to their relatively higher variability. However, due to their lower power saving potentials, these are also accompanied by observable performance degradations. Thus, overall EDP improvement remains less significant. On the other hand, $Q2$ and $Q3$ applications generally demonstrate substantial power savings as well as EDP improvements. One exception to this is `mgrid`. Although it shows high power savings, `mgrid` also experiences comparable performance degradation. Therefore, its EDP improvement remains less emphasized compared to the other $Q3$ applications. The trivial $Q2$ applications `swim` and `mcf` exhibit above 60% EDP improvements. Our experimental system also achieves EDP improvements as high as 34%—in the case of `equake`—for the highly variable $Q3$ benchmarks. For all $Q2$, $Q3$ and $Q4$ applications, the average EDP improvement is 27%, with an average 5% performance degradation. Considering all experimented applications, except for the ones with no variability and power savings potentials, the average EDP improvement is 18%, with an average 4% performance degradation.

6.2 Improvements with GPHT over Reactive Dynamic Management

Many of the previous dynamic management techniques simply respond to previously observed application behavior. We refer to these as "reactive" approaches. Although these approaches perform well for many applications, they are prone to significant misconfigurations for workloads with variable behavior. On the other hand, our on-the-fly, GPHT-based dynamic management framework can respond to these variations proactively, providing better system adaptation. Here we compare the achieved power/performance trade-offs of our GPHT-based dynamic management framework to those of a reactive system. For the reactive method, we choose a simple, commonly-used approach, where the configuration of the processor for an execution interval is chosen based on the last observed behavior. This method is identical to the *last value* prediction we had discussed in Section 3.

In Figure 12 we show the achieved EDP improvement and performance degradation with both dynamic management methods. We show the results for the highly variable $Q3$ and $Q4$ benchmarks, as well as the high-power-savings and low-variation $Q2$ benchmarks. For many of the $Q1$ applications, the trade-offs with the reactive approach is comparable to our GPHT-based approach. For these stable applications, responding to previously seen behavior is already the near-optimal approach.

Figure 12 depicts the advantage of employing our dynamic management techniques guided by on-the-fly phase predictions. For the variable $Q3$ and $Q4$ benchmarks, GPHT-based, proactive management achieves superior EDP improvements, compared to last-value-based, reactive dynamic power management. In general, the performance degradations experi-

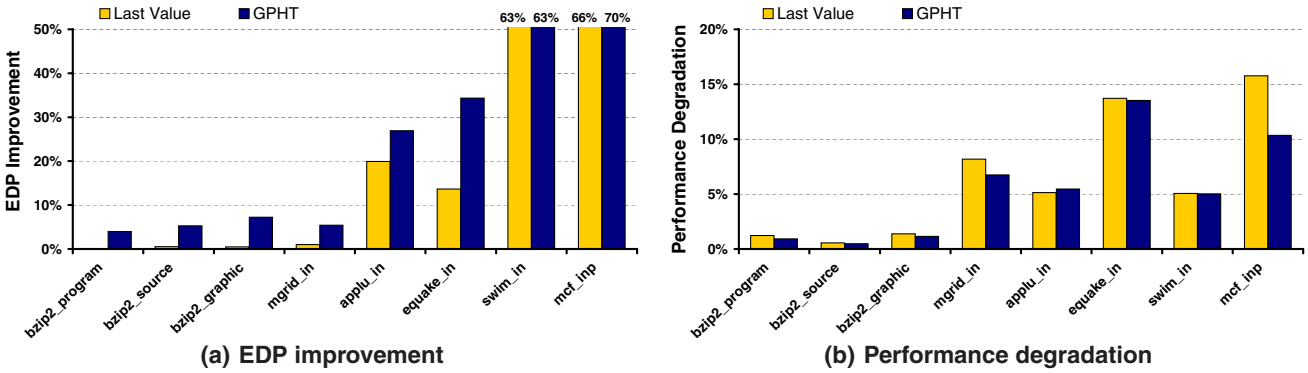

Figure 12. EDP improvement and performance degradation with GPHT and last value prediction for $Q2$, $Q3$ and $Q4$ benchmarks.

enced by our GPHT framework are less than or comparable to those of last value based management. The two $Q2$ benchmarks behave somewhat differently. For `swim`, which has virtually no variability (lying on the x axis in Figure 3), both approaches achieve almost identical results. For `mcf`, which shows a small amount of variability, GPHT-based management achieves a slightly better EDP and observably less performance degradation. As expected, the improvements with the less memory-bound $Q4$ applications are usually less significant than the other benchmarks. Nonetheless, while the reactive approach provides almost no benefits for these applications, GPHT-based dynamic management improves their EDP by approximately 5%. On average, GPHT-based dynamic management achieves an EDP improvement of 27%, with a performance degradation of 5%. Last value based reactive approach achieves 20% EDP improvement and 6% performance degradation for the same set of applications. Thus, applying dynamic management under the supervision of our on-the-fly phase predictions provides a 7% EDP improvement over reactive methods, while inducing comparable or less performance degradations than the reactive approaches. These results clearly show the significant benefits of runtime phase prediction and its application to dynamic power management.

6.3 Bounding Performance Degradation with More Conservative Phase Definitions

The EDP improvement results presented in the above subsections show that our GPHT-based dynamic power management framework offers substantial benefits to improving power/performance efficiency of applications. However, in some cases, the observed performance degradations for some of the applications may not be acceptable for a deployed system. In such a scenario, it might be preferable to relax the power savings to achieve bounded worst-case performance degradations.

In Section 5 we have explained that, in our real-system implementation we can simply reconfigure our phase definitions and the corresponding DVFS look-up table for alternative implementations. Here, we implement such an alternative dynamic management system that targets at bounding performance degradation by 5%. For this implementation, we redefine our phases to meet our performance goal with the help of previous *IPCxMEM* experiments described in Section 4. We look at the achieved BIPS at each DVFS setting for each of the IPCxMEM grid points, and draw the DVFS domains on our grid that satisfy our performance target. After this step, we redefine our phases to match these DVFS settings. Based on these phase definitions, our new deployed system meets the target performance with less aggressive power savings.

In Figure 13, we show the resulting performance degradations, power and energy savings, and EDP improvements for five of our benchmarks that originally had more than 5% performance degradations. With our new conservative phase definitions, all of these applications experience performance degradations significantly lower than 5%. Thus, our new system can successfully sustain application performances within our specified degradation limit. On the other hand, due to smaller power savings, the EDP improvements are reduced by more than 2X from previous results to conservatively meet the desired performance targets.

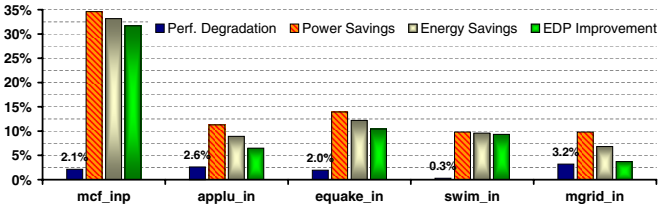

Figure 13. Power/Performance results for our conservative phase definitions that aim to bound performance degradation by 5%.

These presented results show the versatility of our phase-based dynamic management framework, which can be simply configured for different targets under different scenarios. These reconfigurations can be done even after system deployment, with minimal intrusion to overall system operation. Thus, our complete real-system implementation, presented in this paper, serves as an effective, generic power management framework, which can be employed on a running system to support a range of dynamic management goals.

7 Related Work

Several previous studies investigate methods to monitor and utilize application phases for architectural and system adaptations. Dhodapkar and Smith [7] use application working set information to guide dynamic hardware reconfigurations. Isci and Martonosi [13] track variations in application power behavior to detect repetitive execution. Zhou et al. monitor

memory access patterns for energy-efficient memory allocation [30]. Annavaram et al. identify sequential and parallel phases of parallel applications to distribute threads efficiently on an asymmetric multiprocessor [2]. Weissel and Bellosa also monitor memory boundedness of applications to adapt processor execution to different phases on the fly [27]. These works show interesting applications for different aspects of application phase behavior. However, these approaches do not consider predicting future phase behavior of applications and perform adaptive responses reactively, based on most recent behavior.

Some earlier work also considers prediction of future application behavior. Duesterwald et al. utilize performance counters to predict certain metric behavior such as IPC and cache misses based on previous history [8]. They also show that table based predictors perform significantly better than statistical approaches to predict variable application behavior. Lau et al. consider prediction of phase transitions as well as sample phase durations using different predictors [18]. While these works provide significant insights to predictability of application behavior, they do not evaluate the runtime applicability of these predictions to dynamic management.

Sherwood et al. describe a microarchitectural phase predictor based on the traversed basic blocks [24]. They apply this prediction methodology to dynamic cache reconfigurations and scaling of pipeline resources. This work describes fine-grained, microarchitecture-level phase monitoring and dynamic management, based on architectural simulations, while our work describes a deployed real-system framework for on-the-fly phase prediction of running applications and system-level management. Shen et al. detect repetitive phases at runtime by monitoring reuse distance patterns with application to cache configurations and memory remapping [21]. This work employs detailed program profiling and instrumentation to detect repetitive phases. In contrast, our work identifies recurrent execution and predicts phases seamlessly during native application execution without prior instrumentations or profiling. Wu et al. also describe a real-system implementation of a runtime DVFS optimizer that monitors application memory accesses [28]. This work requires the applications to execute from within a dynamic instrumentation framework and relies on periodic dynamic profiling of code regions, inducing additional operation overheads. In comparison, our deployed system operates autonomously on any running application, without necessitating any dynamic instrumentation support or prior profiling, and with no observable overheads to application execution.

8 Conclusion

This work presents a fully-automated, real-system framework for on-the-fly phase prediction of running applications. These runtime phase predictions are used to guide dynamic voltage and frequency scaling (DVFS) as the underlying dynamic management technique on a deployed system.

We experiment with different prediction methods and propose a *Global Phase History Table* (GPHT) predictor, leveraged from a common branch predictor architecture. Our GPHT predictor performs accurate on-the-fly phase predictions for running applications with no visible overheads. For highly variable applications, our GPHT predictor can reduce mispredictions by 6X, compared to statistical prediction approaches. This phase prediction framework efficiently cooperates with DVFS to dynamically adapt processor execution to varying workload behavior. DVFS, guided by these phase predictions, improves the energy-delay product of variable workloads by as much as 34%, and on average by 18%. Compared to previous reactive approaches, our method improves the energy-delay product of applications by as much as 20% and on average by 7%.

Our presented results show the promising benefits of runtime phase prediction and its application to dynamic management. As power management is an increasingly pressing concern, the necessity of such workload-adaptive techniques also increases. Our real-system solution, with its energy-saving potential and negligible-overhead operation, can serve as a foundation for many dynamic management applications in current and emerging systems.

Acknowledgments

We would like to thank Qiang Wu for his help during the development of this work and for several useful discussions. We also thank the anonymous reviewers for their useful suggestions. This research was supported by NSF grant CNS-0410937. Martonosi's work is also supported in part by Intel, IBM, SRC and the GSRC/C2S2 joint microarchitecture thrust.

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] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law Through EPI Throttling. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32)*, 2005.

[3] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. Hankins, and B. Davies. The Fuzzy Correlation between Code and Performance Predictability. In *Proceedings of the 37th International Symp. on Microarchitecture*, 2004.

[4] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W.Hwu. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *Proceedings of the 35th International Symp. on Microarchitecture*, Nov. 2002.

[5] 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.

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

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

[8] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *IEEE PACT*, pages 220–231, 2003.

[9] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal, Q2, 2003*, 7(02), 2003.

[10] C. Hu, D. Jimenez, and U. Kremer. Toward an Evaluation Infrastructure for Power and Energy Optimizations. In *Workshop on High-Performance, Power-Aware Computing*, 2005.

[11] 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.

[12] C. Isci and M. Martonosi. Identifying Program Power Phase Behavior using Power Vectors. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-6)*, 2003.

[13] C. Isci and M. Martonosi. Detecting Recurrent Phase Behavior under Real-System Variability. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct. 2005.

[14] C. Isci, M. Martonosi, and A. Buyuktosunoglu. Long-term Workload Phases: Duration Predictions and Applications to DVFS. *IEEE Micro: Special Issue on Energy Efficient Design*, 25(5):39–51, Sep/Oct 2005.

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

[16] R. Kotla, A. Devgan, S. Ghiasi, T. Keller, and F. Rawson. Characterizing the Impact of Different Memory-Intensity Levels. In *IEEE 7th Annual Workshop on Workload Characterization (WWC-7)*, Oct. 2004.

[17] R. Kotla, S. Ghiasi, T. Keller, and F. Rawson. Scheduling Processor Voltage and Frequency in Server and Cluster Systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[18] J. Lau, S. Schoenmackers, and B. Calder. Transition Phase Classification and Prediction. In *11th International Symposium on High Performance Computer Architecture*, 2005.

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation (PLDI)*, June 2005.

[20] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *Proceedings of the 37th International Symp. on Microarchitecture*, 2004.

[21] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Oct. 2004.

[22] 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.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, Oct 2002.

[24] 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.

[25] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

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

[27] 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.

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

[29] T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, May 1992.

[30] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004.