Cross-Component Energy Management: Joint Adaptation of Processor and Memory

XIAODONG LI, RITU GUPTA, SARITA V. ADVE, and YUANYUAN ZHOU
University of Illinois at Urbana-Champaign

Researchers have proposed the use of adaptation to reduce the energy consumption of different hardware components, such as the processor, memory, disk, and display for general-purpose applications. Previous algorithms to control these adaptations, however, have focused on a single component. This work takes the first step toward developing algorithms that can jointly control adaptations in multiple interacting components for general-purpose applications, with the goal of minimizing the total energy consumed within a specified performance loss. Specifically, we develop a joint-adaptation algorithm for processor and memory adaptations. We identify two properties that enable per-component algorithms to be easily used in a cross-component context—the algorithms' performance impact must be guaranteed and composable. We then modify a current processor and a memory algorithm to obey these properties. This allows the cross-component problem to be reduced to determine an appropriate (energy-optimal) allocation of the target performance loss (slack) between the two components. We develop such an optimal slack allocation algorithm that exploits the above properties. The result is an efficient cross-component adaptation framework that minimizes the total energy of the processor and memory without exceeding the target performance loss, while substantially leveraging current per-component algorithms. Our experiments show that joint processor and memory adaptation provides significantly more energy savings than adapting either component alone; intelligent slack distribution is specifically effective for highly compute- or memory-intensive applications; and the performance slowdown never exceeds the specification.

Categories and Subject Descriptors: C.5.5 [**Computer System Implementation**]: Servers; C.4 [**Performance of Systems**]

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Energy management, low-power design, processor, memory, performance guarantee, control algorithms, adaptive systems

ACM Reference Format:

Li, X., Gupta, R., Adve, S. V., and Zhou, Y. 2007. Cross-component energy management: Joint adaptation of processor and memory. ACM Trans. Archit. Code Optim. 4, 3, Article 14 (September 2007), 31 pages. DOI = 10.1145/ 1275937.1275938 http://doi.acm.org/10.1145/1275937.1275938

This work is supported in part by an equipment donation from AMD and the National Science Foundation under Grant No. CCR-0205638, CCR-0209198, CCF-0313286, CCF-0541188, EIA-0224453, and EIA-0313286.

Authors' address: Xiaodong Li, Ritu Gupta, Sarita V. Adve, and Yuanyuan Zhou, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801; email: xli3@uiuc.edu, rgupta5@uiuc.edu, sadve@cs.uiuc.edu, yyzhou@cs.uiuc.edu.

1. INTRODUCTION

Energy consumption is an important design-time concern across a large spectrum of computer systems. These systems include mobile systems, such as laptops, where battery life must be maximized, as well as high-end servers in data centers, where the energy and cooling bill must be minimized [Carrera et al. 2003; Gurumurthi et al. 2003; Lefurgy et al. 2003]. The last few years have seen a significant amount of research in the use of adaptation to save energy in various hardware components, such as the processor [Albonesi 1999; Brooks and Martonosi 1999], memory [Lebeck et al. 2000], disk [Gurumurthi et al. 2003], network card [Kravets and Krishnan 1998], and display [Flinn and Satyanarayanan 1999]. The general idea is to support multiple power modes for a given component and provide a control algorithm that sets the appropriate mode at a given time (e.g., appropriate size of the processor instruction queue [Folegnani and Gonzlez 2001], power down state of a memory chip [Lebeck et al. 2000; Li et al. 2004], and speed of a disk [Gurumurthi et al. 2003]). The control algorithm is critical to the effectiveness of the adaptive components and much recent research has focused on the design of such algorithms. Most such algorithms, however, focus on a single hardware component (e.g., processor, memory, disk), seeking to minimize the energy consumed by that component.

In a real system, multiple components can be significant contributors to the total system energy. For example, while much of the early focus on energy management was on processors, more recently, researchers have recognized that memory is also a key consumer of energy. A study shows that for fully configured IBM server systems, memory energy can be as high as 150% of processor energy [Lefurgy et al. 2003], making both processor and memory significant contributors to the overall system energy. Further, as energy-management techniques are applied to one component to reduce its energy, other components become dominant contributors. Thus, to reap the full benefits of energy management research for real systems will require the ability to use per-component control algorithms *in concert* with each other. There are at least two concerns that motivate rethinking these algorithms for use in such a *cross-component* context.

1. Minimizing per-component energy may not minimize total system energy. First, in a system with multiple adaptive components, minimizing the energy of a given component may not minimize the energy of the system. Fan et al. [2003] show data to illustrate this for processor and memory components, using an idealized dynamic voltage-scaling algorithm for the processor and a simple memory power-control algorithm. Table I illustrates this effect for the adaptations and algorithms studied in this paper. For the equake SPEC benchmark, Table I shows the total and component-wise energy consumption of systems that (i) minimize processor energy with processor-only adaptation (P), (ii) minimize memory energy with memory-only adaptation (M), and (iii) minimize total processor and memory energy with joint adaptation of the two components (PM). (The energy reported is normalized to the base, nonadaptive system.) Details for these results appear in later sections. Here we note that the PM system has higher processor energy than P and higher

Table I. Total and Per-Component Energy with Processor Only, Memory Only, and Joint-Processor–Memory Adaptation for Equake with 5% Slack (Defined as the Target Performance Slowdown)[a]

	Components Adapted (%)		
	Processor-Only	Memory-Only	Processor and Memory
Processor energy	36.2	72.2	41.7
Memory energy	27.9	7.6	10.7
Total energy	64.1	79.8	52.4

[a]The energy reported is normalized to the total energy of the base, non-adaptive system.

memory energy than M. Thus, the processor configuration that produces system-wide minimum energy uses more energy than the processor configuration that minimizes only the processor energy. An analogous observation holds for memory energy. This data clearly indicates that the naive combinations of the processor and memory management algorithms may not minimize total energy.

2. Total per-component performance degradation may exceed acceptable system limit. Second, energy-driven adaptations typically have an adverse impact on performance. Most (but not all) prior work for general-purpose applications aims to reduce energy without "much" loss in performance, but without bounding this loss. These algorithms are typically based on heuristics that are painstakingly hand-tuned to limit performance loss. Even so, these algorithms may incur unpredictable and unacceptable performance losses for situations outside the training set of the tuning. For example, Li et al. [2004] report that heuristics based memory algorithms tuned for one SPEC application with 10% slowdown give a slowdown of 60% for another SPEC application. Combining such heuristics-based algorithms in multiple components would make this situation even worse, potentially exploding the parameter space that needs to be hand-tuned. Recognizing the limitations of hand-tuned heuristics-based algorithms, recent work has considered algorithms that assume an acceptable slowdown, and specifically target that slowdown for reducing energy [Li et al. 2004; Huang et al. 2000, 2003b; Dropsho et al. 2002; Hughes and Adve 2004]. When using such algorithms for multiple components, it is unclear how to spread the target performance loss across the different components' energy-management algorithms.

The above two concerns motivate a rethinking of adaptation algorithms for cross-component adaptation. *This paper takes the first step in designing such cross-component control algorithms for saving energy for general-purpose applications.* In this work, we focus on the processor and memory components; since they interact closely, they are significant consumers of system energy in important system classes [Lefurgy et al. 2003] and there has been much work done in control algorithms for these components.

Given the limitations of the hand-tuned heuristics-based algorithms discussed above, our work follows the algorithms that assume a specified acceptable slowdown and guarantees not to exceed that slowdown. (Section 2 further justifies this decision.) Thus, our goal is to develop control algorithms for processor and memory adaptations that can minimize the *total* processor and memory energy while incurring no more than a target performance slowdown (referred to as the *slack*).

Rather than develop new control algorithms from scratch, our approach is to leverage the substantial work already done in the domains of processor and memory adaptation. We identify properties of component-wise control algorithms that enable their use in a cross-component adaptive system. Specifically, the performance impact of the algorithms must be guaranteed and composable (further elaborated in Section 2). We then show how a current algorithm for each of processor and memory can be modified to satisfy the identified properties. The composability property allows our cross-component framework to use the modified algorithms virtually independently for their respective components. To minimize total energy and provide a total performance guarantee, our framework uses a new, but simple, piece called the optimal slack-allocation algorithm. This algorithm uses the performance guarantee and composability properties of the per-component algorithms to distribute the target slack between them in a manner that minimizes total energy (and limits total performance degradation). Each component's algorithm is now able to take the allocated slack and independently perform its component's energy management.

Our cross-component framework has the following benefits: (1) The most difficult parts of energy management, i.e., the control algorithms for the individual components leverage currently existing work. The changes to the current algorithms are relatively small. (2) The adaptations of the two components are coupled through the slack-allocation algorithm, but this is a loose coupling and the actual control algorithms operate independently of each other. This is important since different vendors provide the processor and memory; a framework that requires a tight coupling among the two components could be difficult to deploy in practice and potentially incur excessive overheads.

We perform experiments with six SPEC benchmarks and three values of slack. We find that the modified processor and memory adaptation algorithms alone are both effective in saving energy, but the cross-component adaptation algorithm saves more energy than either technique alone. Our results show the importance of distributing slack among processor and memory in an intelligent- and application-dependent way. Specifically, for applications that are strongly computation-intensive or strongly memory-intensive, the optimal slack algorithm provides significant benefits over any naive distribution. For applications that are in-between, we find that there is a large area where the energy savings are not very sensitive to the distribution. Thus, dividing slack equally among processor and memory is almost as effective as the optimal distribution. Finally, our algorithm is able to guarantee the required performance, in all cases.

2. OVERVIEW OF OUR APPROACH

2.1 Targeting a Performance Slowdown

As discussed in Section 1, previous heuristics-based algorithms that attempt to save energy without "much" loss in performance require painstaking hand-tuning, and, even so, can result in unbounded and unpredictable performance loss for scenarios outside the tuning training set. This is not acceptable for systems, such as high-end data centers, which need to guarantee a performance level to honor service-level agreements with customers. Second, with the advent of the utility computing model, host data centers may find it profitable to follow a business model where users give up some performance to save energy (i.e., cost). We, therefore, follow the approach of the previous algorithms that target a specific performance slowdown and save energy within this slowdown (e.g., [Dropsho et al. 2002; Huang et al. 2000, 2003b; Hughes and Adve 2004; Li et al. 2004]). As discussed in more detail in Li et al. [2004], the right metric for measuring delivered performance is unclear. However, such a metric is independent of whether there is support for energy management and is the subject of other research, especially in the context of utility-based computing models. In this paper, following previous work [Li et al. 2004; Huang et al. 2000, 2003b], we assume that the user is provided the option to get a base "best" performance without energy management, or to further save cost (i.e., energy) by accepting a slowdown relative to this base performance [Li et al. 2004]. We refer to this acceptable slowdown as *slack* and assume it is an input to our system. Other measures and methods of specifying slack are possible, but are independent of and outside the scope of this work.

There are two practical challenges to meeting our goal of minimizing processor + memory energy within the target slack.

2.1.1 *Separating the Impact of Processor and Memory Adaptations.* As Table I illustrates, processor and memory adaptations have an impact on each other. Processor adaptation can affect the time between memory requests, leading to reactions by the underlying memory algorithm. For example, many memory algorithms transition to a low power state if the interrequest time exceeds a certain threshold [Lebeck et al. 2000; Li et al. 2004]. This results in slower response for a subsequent request. This increased response time could, in turn, affect the processor adaptation algorithm. For example, many processor algorithms monitor the utilization of a resource and shut it down if the utilization is below a threshold [Bahar and Manne 2001; Folegnani and Gonzlez 2001; Ponomarev et al. 2001; Sasanka et al. 2002]. The increased memory response time could result in crossing this threshold. This, in turn, could further increase the processor's interrequest time to memory.

The above potentially cyclic interaction implies that a joint processor and memory adaptation algorithm would need to tightly integrate the decisions for the two adaptations. A straightforward such algorithm is to search the cross-product of the space of all possible processor and memory configurations to determine the optimal processor + memory configuration. Unfortunately, current processor and memory adaptation algorithms are already complex enough when used in isolation; tightly coupling them will significantly exacerbate their complexity. Furthermore, since processors and memories are designed by different vendors, an algorithm that tightly integrates their adaptations would be difficult to deploy, in practice.

To address this problem, we require the use of processor and memory algorithms for which the performance impact is *guaranteed* and *composable*. By guaranteed performance, we mean that for any target performance slowdown, the algorithm should provide a guarantee to not exceed that slowdown. By composable performance, we mean that it should be possible to independently deploy both algorithms with a specified slowdown value for each and bound the total net slowdown.

Thus, the first piece of the framework is to design performance guaranteed and composable algorithms for processor and memory adaptation.

2.1.2 *Optimal Slack Allocation between Processor and Memory*. Once we create performance guaranteed and composable per-component algorithms, the next step is to determine the allocation of slack to each component such that the total slack does not exceed the target and the total energy is minimized. We expect that depending on the application, the optimal allocation of the slack to the processor or memory would differ. The final piece of the framework, therefore, is to determine a method for optimally distributing the total available slack among the processor and memory adaptation algorithms for a given application.

The following sections describe the individual processor and memory algorithms followed by the slack-allocation algorithm.

3. PROCESSOR ADAPTATION ALGORITHM

Recently researchers have proposed several processor adaptations to save energy, e.g., instruction window/issue queue resizing [Buyuktosunoglu et al. 2000; Folegnani and Gonzlez 2001; Ponomarev et al. 2001], changing the number of active functional units [Bahar and Manne 2001], pipeline gating on some branches [Manne et al. 1998], and dynamic voltage scaling [Govil et al. 1995; Lu et al. 2002]. As mentioned, the focus of our work is on algorithms to *control* such adaptations.

3.1 Time Scale of Adaptation

To design performance guaranteed and composable control algorithms, a key decision is the choice of time scale of adaptation. Previous processor algorithms for general-purpose applications have adapted on a time scale of several hundred to thousand instructions [Bahar and Manne 2001; Brooks and Martonosi 1999; Buyuktosunoglu et al. 2000; Dropsho et al. 2002; Folegnani and Gonzlez 2001; Manne et al. 1998; Maro et al. 2000; Ponomarev et al. 2001], several million instructions [Huang et al. 2000], subroutines [Huang et al. 2003b], or based on phase-based behavior of programs [Sherwood et al. 2003; Dhodhapkar and Smith 2002]. Each decision exploits a certain type of execution variability. Of these, Huang et al. [2000, 2003b] target a given slack, however, none of these algorithms provide a guarantee on performance loss.

We choose to use a phase-based granularity for processor (and memory) adaptation for the following reasons. Sherwood et al. [2003] have shown that at a large scale (millions of instructions), programs repeat their behavior, and it is possible to predict the occurrence and performance characteristics of these repeated *phases*. Thus, using this granularity is convenient, because it can allow for predictable performance loss (and energy usage). Later, we discuss how we also modify the memory algorithm to operate at phase granularity, enabling composability of the two algorithms.

To track and classify phases, we can use any technique from the literature. For our experiments in this work, we chose the widely used technique of Sherwood et al. [2003]. We discuss more recent alternatives (e.g., Isci and Martonosi [2006]) in the related work section. The technique by Sherwood et al. [2003] tracks and classifies phases every 10 million instructions. It is based upon code execution frequencies and is independent of the architecture configuration used to run the phase. At the end of each group of 10 M instructions (referred to as a *phase interval*), the phase-classification technique assigns a unique *PhaseID* corresponding to the tracked phase. We refer to a phase interval classified as a given phase as an *occurrence* of that phase.

The algorithm also requires a phase predictor to predict the phaseID of the next phase interval. We experiment with two phase predictors: (1) a perfect predictor to determine the limits of the energy benefits from our adaptations, and (2) a simple predictor, which predicts the phase of the next interval to be the same as the current interval, referred to as *Simple* predictor.

3.2 Phase-Based Processor Adaptation Algorithm

We derive a phase-based processor adaptation algorithm that chooses the lowest energy processor configuration for the allocated slack at the beginning of each phase interval. In principle, we can use any processor algorithm that can determine such a configuration. Our algorithm is derived from previous work for multimedia applications [Hughes et al. 2001]. That algorithm works at the granularity of a multimedia application frame, which is analogous (but not identical) to the notion of a phase (see Section 8 for more details). We combine that algorithm with a performance guarantee algorithm (derived from Li et al. [2004]) that ensures that the targeted slack will not be exceeded, *even if the phase was mispredicted* and the wrong configuration inadvertently used. The performance guarantee algorithm assumes that different occurrences of the same phase show stable behavior and the programs are long-running (further elaborated below).

3.2.1 *Algorithm Assuming Perfect Phase Predictor.*

For simplicity, we first describe our processor algorithm assuming a perfect phase predictor. The goal of the algorithm is to slow down each occurrence of a phase by the specified slack. The algorithm employs on-line profiling of the initial occurrences of each phase. These initial occurrences are run with different processor configurations (one configuration for an entire phase occurrence) to provide the execution time and energy for each combination of processor configuration and phaseID. This time and energy is predicted to be the same across all occurrences of the phase. After all the profile data is collected for a given phase, the algorithm can simply determine the processor configuration with the lowest energy such that its execution time is within the targeted slack (i.e., the slowdown relative to the base architecture is less than the user-specified constraint). Subsequent occurrences of the phase are now run at this chosen processor configuration.

The total number of phase intervals used for profiling with this algorithm equals *Number of distinct phases* $\times$ *Number of processor configurations*. Note that the profiling occurs on-line and possibly interspersed with adaptation of other phases. For example, if the first phase interval of phaseID i occurs halfway through the computation, then the first one-half of the computation may have already been adapting the other phases before profiling for phaseID i begins.

3.2.2 *Algorithm Assuming Imperfect Phase Predictor.* The above algorithm is simple, but assumes that the phaseID for the next phase interval can be predicted perfectly. However, this is not always the case. A phase misprediction could result in choosing a configuration that violates the performance constraint by using up too much slack. To accommodate this case, we modify the algorithm (using memory-driven techniques derived from Li et al. [2004]) to track the slack used in each phase interval. If too much slack is used, then different configurations are chosen in subsequent intervals. These configurations use up less than the user-specified slack and are used until the previous overuse is compensated for. To achieve this, at the end of the profiling intervals for a given phaseID, the algorithm builds a table for that phaseID, with an entry corresponding to each architecture configuration and sorted in increasing order of energy. The entry stores the execution time for the architecture for that phaseID.

Now consider the execution of a phase interval i with a certain phaseID P, which may have been predicted incorrectly and run with an inappropriate architecture *Arch*. We use the following terms:

- T^{P}_{Arch} : Execution time for P with the architecture *Arch*.
- T^{P}_{base} : Execution time for P with the base architecture (as recorded in the above table).
- *Slack* : Target user slack (specified as a fraction).
- $UsedSlack_i$: Absolute slack (in terms of time) used in interval i
  $UsedSlack_i = T^{P}_{Arch} - T^{P}_{base}$
- $RemainingSlack_i$: Unused slack (in terms of time) at the end of interval i
  $RemainingSlack_i$ = (Absolute slack available at start of interval i)—$UsedSlack_i = T^{P}_{base} * (Slack) + RemainingSlack_{i-1} - (T^{P}_{Arch} - T^{P}_{base})$

If the remaining slack at the end of interval i is negative, then the algorithm calculates a new desired execution time for the next interval $i+1$ with phaseID P' as $T^{P'}_{base} * (1 + Slack) + RemainingSlack_i$ where $T^{P'}_{base}$ is the execution time with the base architecture for the predicted phaseID for interval $i+1$.

The algorithm looks up the table for the predicted phase for interval $i+1$. It chooses the first architecture configuration in the table that has an execution time value less than the above and uses it to run interval $i+1$. Assuming enough phase intervals remain to execute, any overuse of slack is compensated for, and a performance guarantee is maintained.[1]

We note that the above algorithm makes the following two assumptions for a performance guarantee. First, it assumes a phase classification algorithm such that different occurrences of the same phase exhibit stable behavior (very small variations in execution time and energy). Previous work has shown such classification algorithms for various applications and our experiments show similar behavior for the algorithm and applications we use. There is also much ongoing research in phase-classification algorithms that strives to achieve this goal for increasingly large classes of applications (e.g., Isci and Martonosi [2006]) Note that the performance guarantee algorithm does *not* assume perfect phase detection. The second assumption is that the program runs long enough that excessive use of slack because of a phase misprediction can be compensated for in successive phase intervals. Again, this is a reasonable assumption since energy management is likely to be important primarily for long-running programs. A related assumption is that the program exhibits phase behavior (i.e., each phase occurs many times and phases that occur infrequently comprise a negligible portion of the program). Recently there have been many studies that exploit the same assumption [Sherwood et al. 2003; Huang et al. 2003b; Dhodhapkar and Smith 2002; Balasubramonian et al. 2000]. The above assumptions allow our algorithm to provide a performance guarantee, in practice, as confirmed by our experimental results (not a single case of performance violation).

The above discussion focused on phase misprediction during adaptation. A misprediction during profiling is easily handled—the profiling information can be discarded or used (and a future profile avoided) depending on whether that combination of phaseID and architecture has previously been profiled.

It is also acceptable to have a context switch during a phase—the adaptation configuration tables and other information collected by the adaptation or profiling algorithms during the phase interval must be saved and restored across the context switch. Thus, the operating system is a natural place to implement our algorithms.

Finally, although the experiments reported here adapt resources in the processor core, cache adaptations (at phase granularity) can also be handled as part of our processor adaptation algorithm and the subsequent joint-adaptation algorithm. The adaptive caches are simply treated as yet another adaptive resource in the processor. Thus, when considering processor configurations for profiling and adaptation, we also include the different cache configurations in the configuration space.

3.3 Overheads

The processor algorithm incurs overheads for phase detection/classification, profiling, and adaptation.

[1]Note that if there are not enough phase intervals remaining to execute, then assuming a reasonably long-running execution, the overuse of slack in one phase interval will be a negligible fraction of the execution time.

3.3.1 *Phase Detection and Classification.* For phase detection and classification, we use an architecture similar to the one by Sherwood et al. [2003], with similar space and time overheads. Specifically, on a branch, its PC is used to hash into a small (32-entry) accumulator table. The corresponding counter entry is incremented by the total number of instructions executed since the last branch. We expect the energy overhead for this to be relatively small compared to the energy saving from the adaptation algorithm (30–50% of the entire processor energy). In addition, a history table is required to store the phase footprints (maximum 150 entries for our benchmarks) [Sherwood et al. 2003]. This table is accessed only once for each phase interval; therefore, its energy overhead is amortized over 10 million instructions and is negligible. Further, we note that many optimizations have been proposed using phase detection [Sherwood et al. 2003; Dhodhapkar and Smith 2002] and the overheads can be further amortized across those optimizations as well. Finally, as noted above, our algorithm works with any phase-detection mechanism and future improvements in such mechanisms can be incorporated in our work as well.

3.3.2 *Profiling.* Since profiling is on-line, the overhead stems from increased execution time and/or energy because of the possible use of suboptimal processor configurations during profiling. However, this overhead is also expected to be negligible for long-running applications since the number of phase intervals profiled is expected to be a small fraction of the total intervals executed. This overhead can be even further bounded by monitoring the execution time of the profiled frame. If it exceeds the maximum allowed time for the frame, the profiled configuration will never be used, profiling can be aborted, and the processor switched to the base configuration for this phase occurrence.

3.3.3 *Adaptation.* For adaptation, at the beginning of each phase interval, we need to predict the phaseID of the next phase interval and perform a table lookup to find the lowest energy configuration that provides the performance guarantee. Our Simple predictor simply assumes the next phase interval has the same phaseID as the previous one; therefore, this overhead is trivial. The table lookup requires N_{config} number of comparisons in the worst case, where N_{config} is the total number of processor configurations. Both of these overheads are incurred only once per phase interval. They are, therefore, amortized over 10 million instructions and are negligible.

4. MEMORY ADAPTATION ALGORITHMS

4.1 Previous Memory Adaptation Algorithms

To reduce memory energy consumption, modern memory systems such as RDRAM [Rambus 1999] allow each memory chip to transition from normal *active* operating mode into several low-power operating modes—*standby*, *nap*, and *powerdown*. To service a request, a chip in a low-power mode needs to transition to *active*, which incurs extra delay and energy.

Researchers have recently proposed several memory adaptation control algorithms [Lebeck et al. 2000; Li et al. 2004] and have shown that a dynamic

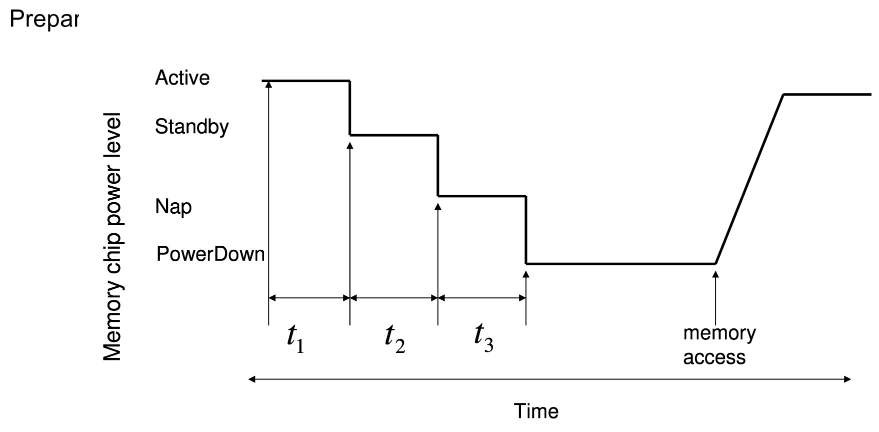

Fig. 1. Conceptual view of the dynamic memory adaptation algorithm. t_1, t_2, and t_3 are the idle time thresholds that memory chips use to transition into a lower power mode. A chip needs to transition into the active power mode to serve a memory access request. PD calculates the thresholds every epoch as a function of predicted memory traffic and acceptable slowdown for the next epoch.

scheme that transitions a chip into low-power modes after a threshold of idle time performs better than a static scheme that places all chips in a fixed power mode, except when necessary to service a request. Figure 1 gives a conceptual view of the dynamic memory adaptation algorithm. In this work, we study the best available state-of-the-art dynamic algorithm, PD, proposed by Li et al. [2004]. PD eliminates the painstaking manual threshold parameter tuning of previous work [Lebeck et al. 2000] by periodically tuning its thresholds automatically using a heuristic function. It takes a specified slack as input to both tune its thresholds and to provide a performance guarantee, based on that slack.

PD uses the insight that the optimal thresholds are a function of the memory traffic and the acceptable slowdown (slack) that can be incurred. PD recalculates its thresholds (or adapts) every few million instructions, called an epoch. At the start of an epoch, it predicts the memory traffic and the available slack for that epoch. Using these predictions and a predetermined heuristic function, it calculates a set of thresholds for that epoch. These thresholds specify the idle times after which PD will transition memory from one power mode to another during that epoch.

To overcome any errors in its predictions and heuristics, PD provides an additional heuristic parameter to self-adjust the threshold computation functions for the next epoch, based on its performance in the last epoch.

To provide a performance guarantee, PD tracks the slowdown introduced by its memory adaptation and forces all chips to active when the observed slowdown is greater than the maximum slowdown allowed by the user.

4.2 Modified PD Algorithm

We modify PD for cross-component adaptation as follows. First, we modify PD to adapt at the granularity of a phase (versus an epoch). Having both the processor and memory algorithms adapt at the same granularity allows for composability.

Using the phase granularity for memory also increases the accuracy of some predictions required by the memory algorithm. Specifically, to set its thresholds for the next epoch, the original PD needs to predict the memory access behavior of the next epoch. The modified PD sets the thresholds at the start of a new phase interval and needs to predict the memory access behavior in that phase interval. The algorithm can make this prediction quite accurately, based on profiled information for the phase (analogous to the processor algorithm).

Second, the original PD sets its thresholds (and resulting energy savings) based on a function of the available slack for the next epoch and a dynamically adjustable heuristic parameter called *Self adjust factor* (term C in [Li et al. 2004]). As discussed in Section 5, the joint processor-memory algorithm relies on profiling information to determine the energy savings with different memory adaptations (thresholds). To reduce the number of profiles needed, we fix the *Self adjust factor* to an empirically determined value (10). The modified algorithm is still able to adapt and save energy effectively, responding to the available slack. As we will see later from Figures 4 and 5, the modified PD algorithm reduces memory energy to a small fraction of the base memory energy, in most cases. Further, the system energy is dominated by the processor energy. Therefore, we expect the total system energy savings from our modifications to be close to those achievable from the original algorithm.

The overhead of the modified PD algorithm is almost the same as that of the original PD algorithm (discussed in detail in Li et al. [2004]). The algorithm uses the same phase detector (and amortizes its overhead) as the processor-adaptation algorithm.

5. JOINT PROCESSOR AND MEMORY ADAPTATION

As mentioned in Section 2, our framework for joint processor and memory adaptation requires (1) performance guaranteed and composable processor- and memory-adaptation algorithms and (2) an optimal slack-allocation algorithm. Section 5.1 discusses how the modified algorithms described in Sections 3 and 4.2 satisfy the performance guarantee and composability properties. Section 5.2 then presents the slack-distribution algorithm. Section 5.3 summarizes the full framework.

5.1 Performance Guarantee and Composability

Sections 3 and 4 already describe how our processor and memory algorithms provide a performance guarantee. We next show that they are also performance composable at the phase granularity. That is, we need to show that it should be possible to deploy both algorithms with a specified slack value for each and bound the total net slowdown at the phase granularity.

Let the total execution time without any processor and memory adaptation be T_{base}. First, consider the case where only the processor adapts with allocated slack S_{cpu}. The processor algorithm (Section 3) chooses a minimum energy configuration, C, while satisfying the performance bound corresponding to S_{cpu}. Let the execution time for this processor configuration be T_C (for the given phase). Then $T_C \leq T_{base} * (1 + S_{cpu})$.

Next, consider the case where memory adapts in addition to the processor. Since the processor configuration C stays fixed for the entire phase interval, the impact of our memory adaptation algorithm is to incur a slack, say S_{mem}, *relative to the execution time with the processor configuration C and no memory adaptation*. That is, if C is different from the base processor, then from the viewpoint of the memory algorithm, the processor appears simply as a lower configuration new processor on which memory exerts an independent slack of S_{mem}. Thus, if T is the execution time with the processor and memory adaptations, we have $T \leq T_C * (1 + S_{mem})$. Substituting for T_C, we have:

$$T \leq T_{base} * (1 + S_{cpu}) * (1 + S_{mem}) \tag{1}$$

The above equation allows us to express the total slack as a function of the slack given to the processor and memory and, therefore, the algorithms are composable at the phase granularity.

Note that this composability property occurs as a result of the design of our algorithms, because (1) both processor and memory adapt at the same time scale of a phase interval, (2) the processor's configuration stays fixed throughout a phase interval and is not affected by the memory algorithm, and (3) the memory algorithm measures its slack with respect to this fixed processor configuration, but is not affected by the processor's adaptation in any other way.

In general, processor and memory adaptation algorithms may not be composable. As described in Section 2, the adaptation of one component can potentially affect the other. For example, processor adaptation can potentially affect the time between memory requests. This can lead to reactions by the underlying memory algorithm, e.g., the interrequest time may exceed the threshold for the PD algorithm, resulting in the memory going to a lower power state and increasing the service time for the subsequent accesses. This increased service time could, in turn, affect the processor adaptation algorithm (e.g., if the processor adapts its resources based on the utilization of the resources), causing a cyclic effect.

Our choice of the processor and memory algorithms breaks the above cycle and allows the two to operate independently.

5.2 Optimal Slack-Distribution Algorithm

The goal of the *slack-distribution* algorithm is to divide the total target slack between processor and memory to minimize the total energy consumed. The optimal distribution is not straightforward to determine because the energy savings per unit slack are different for processor and memory and are application-dependent. To achieve an optimal algorithm, we first formalize an optimization problem for each phase interval as follows.

5.2.1 *Problem Formalization.* Let T_{base}, S_{cpu}, and S_{mem} be as defined in Section 5.1. Suppose the user specified slack is *AvailableSlack* (e.g., 10%). Let T be the execution time, E_{cpu} be the processor energy, and E_{mem} be the memory energy with processor and memory adaptations for the given phase interval. T, E_{cpu}, and E_{mem} are functions of S_{cpu} and S_{mem}, and T should satisfy the performance constraint. The slack distribution problem can now be stated as the following optimization problem:

minimize $E_{cpu}(S_{cpu}, S_{mem}) + E_{mem}(S_{cpu}, S_{mem})$
subject to $T(S_{cpu}, S_{mem}) \leq T_{base} * (1 + AvailableSlack) \ldots$ performance constraint

5.2.2 *Determining Functions $E_{cpu}(S_{cpu}, S_{mem})$ and $E_{mem}(S_{cpu}, S_{mem})$.* The first challenge to solve the above optimization problem is to find the relationship between a slack distribution, S_{cpu} and S_{mem}, and the energy, E_{cpu} and E_{mem}. One method to address this challenge is to analytically estimate E_{cpu} and E_{mem} based on S_{cpu} and S_{mem}; however, such an estimate is difficult enough even with adaptation in a single component. Our solution is to use a profile-based method analogous to the processor algorithm. The slack distribution, S_{cpu} and S_{mem}, determines a corresponding processor and memory (threshold) configuration, say $P(S_{cpu})$ and $M(S_{mem})$, respectively. At run time, for each phaseID and a given slack distribution, we use one phase occurrence to set the processor and memory configuration to $P(S_{cpu})$ and $M(S_{cpu})$, respectively, and measure the resulting energy consumption, E_{cpu} and E_{mem}.

5.2.3 *Satisfying the Performance Constraint.* The next challenge is to determine which slack distributions will satisfy the performance constraint. Using the composability property (Eq. 1 from Section 5.1), it follows that the performance constraint in the above formulation is satisfied if

$$(1 + S_{cpu}) * (1 + S_{mem}) \leq (1 + AvailableSlack) \qquad (2)$$

5.2.4 *Solving the Optimization Problem.* Equation (2) enables an efficient search through the space of possible slack distributions to solve the optimization problem. First, we discretize the interval [0, *AvailableSlack*] into several steps. For each step, we assign the value of the step to S_{cpu} and calculate S_{mem} based on Eq. (2). Equation (2) allows several values for S_{mem}; we choose the maximum value, because the energy E_{mem} is usually smaller with a larger slack. This yields a small number of slack distributions that satisfy the performance constraint and that need to be explored. We can simply perform a linear search over these distributions. For each distribution, we determine the total energy ($E_{cpu} + E_{mem}$) for that phase (using the profile method described above). We solve the problem by choosing the distribution (S_{cpu}, S_{mem}) with the minimum total energy.

In our experiments, we report results with *AvailableSlack* values of 5, 10, and 20%. We used a total of 6 slack distributions (i.e., profile 6 occurrences of each phaseID) to obtain the best slack distribution for the 5% *AvailableSlack* case and 11 slack distributions for the 10 and 20% cases. (The number of slack distributions explored is adjustable. The user may explore more distributions to improve the precision of solution at the expense of more profiling. We chose the above numbers to achieve a good trade-off between profiling overhead and precision.)

5.3 Putting it Together

Figure 2 provides a high-level conceptual overview of the joint algorithm. In our implementation, once the optimal slack distribution for a phaseID is chosen,

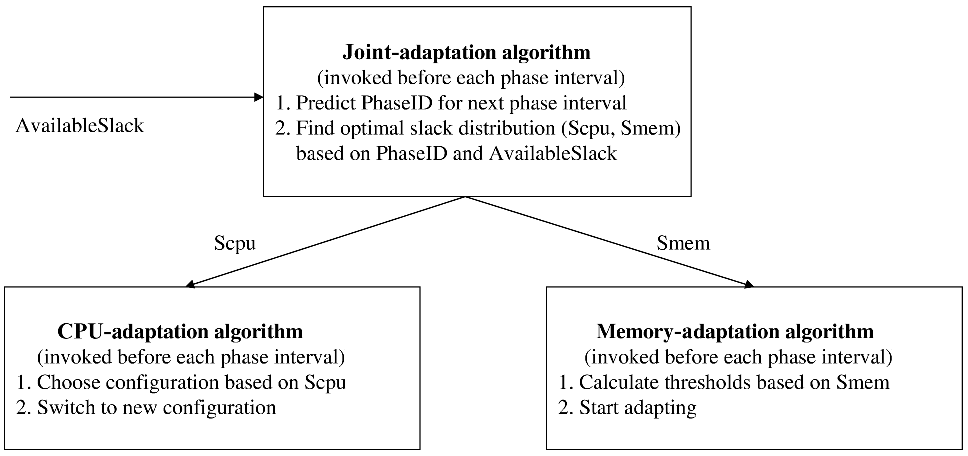
Joint-adaptation algorithm
(invoked before each phase interval)
1. Predict PhaseID for next phase interval
2. Find optimal slack distribution (Scpu, Smem) based on PhaseID and AvailableSlack
AvailableSlack
Scpu
Smem
CPU-adaptation algorithm
(invoked before each phase interval)
1. Choose configuration based on Scpu
2. Switch to new configuration
Memory-adaptation algorithm
(invoked before each phase interval)
1. Calculate thresholds based on Smem
2. Start adapting

Fig. 2. Conceptual view of the joint adaptation algorithm.

```
**Algorithm 1** Joint adaptation algorithm for each phaseID
1: Run initial occurrences of phaseID with different processor configurations. Measure the energy and slack used and the number of memory references.
2: Let $STEP$ be the number of slack distributions we want to profile.
3: for each $i$ in $0,\ 1, ...,\ STEP$ do
4:     $S_{cpu} = \frac{i \cdot AvailableSlack}{STEP}$. Let $P(S_{cpu})$ be the minimum energy processor configuration that uses at most slack $S_{cpu}$.
5:     $S_{mem} = \frac{(1+AvailableSlack)}{(1+S_{cpu})} - 1$. Let $M(S_{mem})$ be the memory thresholds based on $S_{mem}$.
6:     Run the next occurrence of phaseID with processor configuration $P(S_{cpu})$ and memory thresholds $M(S_{mem})$. Record the sum of processor and memory energy as $E_i$.
7: end for
8: Compare $E_i$, for $i \in 0,\ 1, ..,\ STEP$. Store in a table indexed by phaseID the processor configuration and memory thresholds that give the minimum $E_i$.
9: Run subsequent occurrences of phaseID with the above processor and memory configurations.
```

Fig. 3. Joint algorithm.

the processor configuration and memory thresholds corresponding to that distribution are stored in a table indexed by the phaseID. At the beginning of a phase interval, the implementation looks up this table and sets the processor and memory configurations accordingly for the phase. The full algorithm is summarized as Algorithm 1 in Figure 3.

In case of phase mispredictions, we could use the individual processor and memory performance guarantee algorithms separately to ensure that the overall performance constraint is satisfied. In our implementation, we use a unified performance guarantee algorithm similar to that of the processor algorithm. This algorithm tracks the slack used in each phase interval (as in the processor algorithm). If too much slack is used, subsequent phase intervals use the base processor configuration and *active* power mode for memory until the initial target slack is reaccumulated.

5.3.1 *Overhead Analysis.* The overheads of the joint algorithm because of the constituent processor and memory adaptation algorithms have already been discussed in Sections 3.3 and 4.2. The slack distribution aspect incurs the following additional overhead.

For each phaseID, the number of intervals profiled now increases (from the number of processor configurations) by the number of slack distributions (steps 3–7 in algorithm 1). This increase is small, relative to the total number of phase intervals. It is also small when contrasted with a naive joint algorithm that performs a search of the cross-product of the configuration space for processor and memory. In general, if M is the number of processor configurations and N is the number of slack distributions, then for each phaseID, our algorithm needs N additional profiles over the M already required for processor adaptation. For our experiments M is 54 and N is 6 or 11. In contrast, a naive joint algorithm would need $(M \times N)$ additional profiles per phase over processor adaptation. For our experiments, this is a substantial 324–594 additional profiles over the 54 already needed.

The algorithm also stores the optimal processor configuration and memory thresholds for each phaseID in a table. The space overhead for this is small since the number of distinct phases in the application is typically small (< 35) and each table entry is only a few bytes. For adaptation, at the beginning of each interval, the algorithm determines the appropriate configuration using a single lookup in the above (small) table. This is negligible, considering that it is incurred once every 10 M instructions.

6. EXPERIMENTAL METHODOLOGY

We use the execution-driven RSIM simulator [Hughes et al. 2002] for performance evaluation and the Wattch tool [Brooks et al. 2000] integrated with RSIM for energy. We enhanced RSIM with the RDRAM memory model [Rambus 1999] to simulate memory energy and delay.

The base processor studied is similar to the MIPS R10000 and is summarized in Table II. We assume a centralized instruction window with a unified reorder buffer and issue queue, but a separate physical register file. We allow adaptation of *issue width*, *instruction window size*, and the number of *functional units*, similar to previous work [Hughes et al. 2001; Sasanka et al. 2002]. Our algorithm can incorporate adaptations in other processor resources as well, including adaptive caches, as mentioned in Section 3. It is applicable to other architectures as well, including those with a separate issue queue and reorder buffer.

As in previous work [Hughes et al. 2001; Sasanka et al. 2002], experiments with instruction window adaptation assume eight entry instruction window segments and that at least two segments are always active. A smaller instruction window requires fewer physical registers. We deactivate one integer and one floating-point physical register with each deactivated instruction window entry. Experiments with functional unit adaptations assume that issue width is equal to the sum of the functional units and, hence, changes with the number of functional units. Consequently, when a functional unit is deactivated, the corresponding instruction selection logic is also deactivated. Similarly, the corresponding parts of the result bus, the wake up ports of the instruction window, and ports of the register file are also deactivated.

Table II. Base System Parameters

Base Processor Parameters	
Processor speed	1 GHz
Fetch/retire rate	6 per cycle
Functional units	6 int, 4 FP, 2 address generation
Integer FU latencies	1/7/12 add/multiply/divide(pipelined)
FPU latencies	4 default, 12 divide (all but divide pipelined)
Instruction window	128 entries
Register file size	192 integer and 192 FP
Memory queue size	32 entries
Branch prediction	2 KB bimodal agree, 32-entry RAS
Base Memory Hierarchy Parameters	
L1 (Data)	64 KB, 2-way associative, 64B line
	2 ports, 12 MSHRs
L1 (Instr)	32 KB, 2-way associative
L2 (Unified)	1 MB, 4-way associative, 64B line
	1 port, 12 MSHRs
Base Contentionless Memory Latencies	
L1(Data) hit time (on-chip)	2 cycles
L2 hit time (off-chip)	20 cycles
Main memory (off-chip)	100 cycles

We assume clock gating for all components of all the processor configurations (adaptive and nonadaptive). If a component is clock-gated (i.e., not accessed) in a given cycle, we charge 10% of its maximum power. To fairly represent the state-of-the-art, we also gate the wake-up logic for empty and ready entries in the instruction window as proposed in Folegnani and Gonzlez [2001]. We assume that the resources deactivated by our adaptive algorithms do not consume any power (i.e., they are power gated). Thus, deactivating an unused component saves 10% of the maximum power of the component (i.e., the remaining power after gating).

We profile all possible combinations of the following configurations (54 total): instruction window $\in \{128, 96, 64, 48, 32, 16\}$, number of ALUs $\in \{6, 4, 2\}$ and number of FPUs $\in \{4, 2, 1\}$. We also evaluated our algorithms with a reduced set of configurations to reduce profiling overhead (10 for integer, 13 for FP applications). The results were qualitatively similar; we present results with full profiling to give the processor-only algorithm the best showing.

Table III shows the RDRAM model we use, including the energy consumption for each 512 MB chip and the transition time between different states. We simulate the memory system with eight such memory chips.

We evaluate our algorithms using six SPEC CPU2000 benchmarks, *gzip*, *mcf*, and *twolf* from SPECint and *ammp*, *equake*, and *mesa* from SPECfp. We do not simulate all the SPEC benchmarks because the simulations take a very long time (see below). We chose the above six SPEC benchmarks to cover the space of compute-bound versus memory-bound behavior. Our detailed analysis (Section 7.1.2) shows that this is the primary aspect that affects the impact of our algorithm. Since we already cover applications that demonstrate a broad spectrum of compute-bound versus memory-bound behavior, we do not expect that numbers for more benchmarks would provide more insight on the efficacy

Table III. RDRAM Power Consumption and Transition Time/Power for Different Power Modes

Power State	Power
Active/standby/nap/powerdown	300/180/30/3 mW
Transition	Power and Time
Active → standby	240 mW/1 memory cycle
Active → nap	160 mW/8 memory cycles
Active → powerdown	15 mW/8 memory cycles
Standby → active	240 mW/6 ns
Nap → active	160 m0W/60 ns
Powerdown → active	15 mW/6000 ns

of our algorithms than already provided by our analysis. All applications were compiled with the SPARC SC4.2 compiler. In our simulations, we fast forward each benchmark to skip the initialization part of the application, based on the data by Sherwood et al. [2002].

Since the applications take an extremely long time to simulate, we perform the following two approximations for our simulations. In a real system, these are not required. Both approximations assume a phase classification algorithm that shows little variability in performance and energy among occurrences of the same phaseID. The approximations do *not* assume that the phase detector makes perfect predictions of phaseIDs.

For the first approximation, we collect a trace of the phase behavior of all the applications for their *entire length*. This does not require detailed timing simulation and can be quickly collected. Then for each experiment, we performed a more detailed, but slower, timing/energy simulation. For this slower timing/energy simulation, we simulate the application long enough to ensure that we collect the necessary profiling data for each phaseID and, subsequently, each phaseID occurs at least ten times for adaptation.[2] (This part alone takes 1 month to simulate for *ammp*.) The execution times and energy across these last ten occurrences of a phaseID are averaged to give the execution time and energy for that phaseID. This average value is then fed into the phase trace initially collected to determine the energy and execution time of the entire application for that experiment. In the results reported, energy is reported for all phase intervals—for the profiled phase intervals, we conservatively report the energy with the base architecture (which has the highest energy).

Second, for the joint adaptation runs with the simple predictor, we make the following approximation to determine the data for mispredicted phases (again, only for our simulations). As required for the algorithm, we find the best processor and memory configuration for each phaseID using profiling. In addition, we profile an occurrence of each phaseID with each of the processor/memory configurations determined to be the best for the other phaseIDs. We then use the trace of the phase behavior along with the simple predictor to determine the time and energy for the application.

[2]If the same phase occurs too often through this run, it is fast forwarded, while ensuring that the simulation is adequately warmed up for the next phase that needs to be measured.

7. RESULTS

Sections 7.1 and 7.2 discuss the energy savings of our algorithms with the perfect and simple phase predictors respectively. Section 7.3 discusses the impact on performance. To assess the sensitivity to the optimal slack distribution between processor and memory in the joint adaptation case, we also report results with joint adaptation assuming approximately equal slack distribution among processor and memory. That is, of the discrete slack distributions profiled, we choose the one that is closest to an equal distribution. For brevity, we refer to this as the equal slack distribution algorithm.

7.1 Energy Savings with the Perfect Phase Predictor

7.1.1 *Overall Results.* With the perfect predictor, for each application and slack value of 5, 10, and 20%, Figure 4a shows the energy consumption for the system with only processor adaptation (P), only memory adaptation (M), and joint processor and memory adaptation using the optimal slack-distribution algorithm (PM) and using equal slack distribution (PMe), all normalized to the base system with no adaptation. The first bar in each set in the figure is for the base system. All bars show the distribution of energy between memory and processor. Table IV summarizes the above data by presenting the average and the range of the relative energy savings between key pairs of adaptation algorithms. Our high-level results are as follows. Section 7.1.2 analyzes these results in more detail.

7.1.1.1 *Processor versus Memory Adaptation.* Processor and memory adaptation both provide significant energy savings in all cases, with processor adaptation generally being more effective than memory adaptation for our system. Across all applications and slack values, separate processor and memory adaptation, respectively, provide an average savings of 31 (range 14–44%) and 16% (range 13–20%).

7.1.1.2 *Joint Adaptation.* Joint processor and memory adaptation provides significantly more energy savings than either one alone. Across all applications and slack values, compared to the base system, joint processor, and memory adaptation with optimal slack distribution (PM) provides average energy savings of 46% (25–64%). When compared to a system that already includes processor adaptation (P), PM provides energy savings of an average of 22% (10–35%) (relative to the energy consumed by P). Compared to a system that already includes memory adaptation (M), PM saves an additional 36% on average (13–55%).

7.1.1.3 *Impact of Slack.* Increasing slack from 5 to 20% improves energy savings for many cases. The improvement, however, is modest, at best, for both processor and memory adaptation alone. With joint adaptation, however, the modest benefits in each component add up to significant benefits for some applications. For example, for *mcf*, the energy saving with joint adaptation increases from 49 to 64%, when slack increases from 5 to 20%, but for the same slack increase, processor adaptation alone only improves the energy saving from 38 to

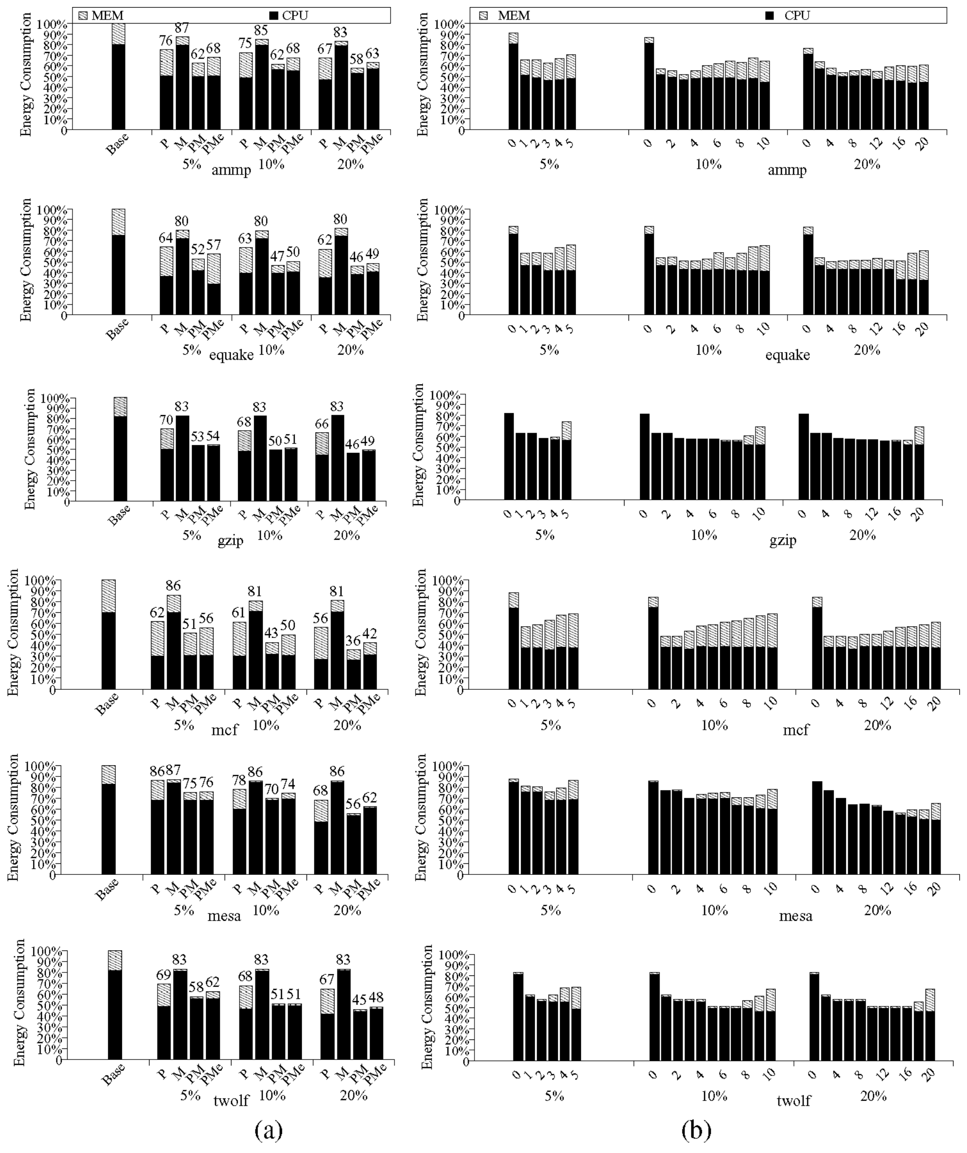

Fig. 4. (a) Energy (normalized to the energy of the base nonadaptive processor) for processor only (P), memory only (M), and joint adaptation with optimal slack (PM) and equal slack (PMe) for the full application and three total slack values with perfect phase prediction. (b) Energy (normalized to the base processor) for joint adaptation for one application phase for three total slack values, with different slack distributions (the number on the x axis is the slack allocated to the processor).

44% and memory adaptation alone improves the energy saving from 14 to 19%. The reason for this is explained in detail in Section 7.1.2.

7.1.1.4 *Slack Distribution.* For joint adaptation, using an appropriate slack distribution is important—clearly, a distribution of 0% slack to processor/

Table IV. Relative Energy Savings (in %) for Different Pairs of Algorithms with the Perfect Phase Predictor[a]

Specified Slack	5%	10%	20%
P vs. base	28 [14, 38]	31 [22, 39]	35 [32, 44]
M vs. base	15 [13, 20]	17 [14, 20]	17 [14, 20]
PM vs. base	41 [25, 49]	46 [30, 57]	52 [42, 64]
PM vs. P	17 [12, 44]	22 [10, 29]	25 [13, 35]
PM vs. M	30 [13, 40]	35 [18, 46]	42 [30, 55]
PM vs. PMe	6 [1, 8]	6 [0, 14]	8 [6, 14]

[a]The numbers are *average [min, max]* reduction in energy consumption of the first algorithm over the second.

all slack to memory (i.e., the M case) or 0% slack to memory/all slack to processor (i.e., the P case) is much worse than the optimal slack distribution (i.e., the PM case) for all applications and total slack values. However, our results show that the energy savings from the optimal slack distribution are close to those of equal distribution for many applications—average savings of PM over PMe are 6, 6, and 8 for 5, 10, and 20% total slack values, respectively.[3] Nevertheless, there are some applications for which the optimal distribution is significantly better. For example, for *mcf* at 10 and 20% slack, optimal distribution saves 14% more energy over equal distribution; for *mesa* at 20% slack, optimal distribution saves 10% energy over equal distribution. These benefits seem worth exploiting, given that the optimal distribution incurs a small overhead (profiling of a handful of extra phase intervals).

7.1.2 *Analysis.* This section uses supplemental data to provide further insights into the high level trends identified. For key phases of each application and each total slack, Table V shows the processor configuration chosen by P and PM (shown as P/PM) and the amount of slack allocated to memory in P/PM. For the longest phase (in terms of number of instructions) of each application and for each value of total slack, Figure 4b presents energy consumption (normalized to base) for different distributions of the total slack between processor and memory. The number on the *x* axis is the amount of slack allocated to the processor.

7.1.2.1 *Processor Adaptation.* From Table V, we see that across all applications, P chooses from a fair number of configurations and often uses different configurations for different phases of the same application. This results in significant energy savings. However, in most cases, the configuration does not change much with increasing slack, indicating that most (but not all) of the energy benefits are obtained with a small slack value. Note that for integer ALUs and instruction window, the base resource sizes are used in several cases for P and/or PM; therefore, our choice of these sizes for the base was not overly aggressive. The choice of four base floating point units was overly aggressive for these applications, since all four units are never used.

[3]The energy saving of system A over system B is defined as $\frac{(energy_B - energy_A)}{energy_B}$, where $energy_A$ and $energy_B$ are the energy consumptions of system A and B, respectively.

Table V. Processor Configuration Chosen (# of Active Instruction Window Entries (IW), ALUs (A), and FPUs (F)) and Slack Allocated to Memory (Mem(%)) for Dominant Phases of Each Application for Different Total Slack for P and PM Systems (shown as **P/PM**)[a]

			5%				10%				20%			
PhaseID	%	IPC	A	F	IW	Mem(%)	A	F	IW	Mem(%)	A	F	IW	Mem(%)
ammp														
9	25	1.35	2/2	1/1	96/128	0/1.9	2/2	2/1	64/128	0/6.7	2/2	1/1	64/96	0/13.2
14	10	1.42	2/2	2/2	48/48	0/1.9	2/2	2/2	48/48	0/6.7	2/2	2/2	48/48	0/5.3
equake														
0	36	0.49	2/2	1/1	96/128	0/3.9	2/2	1/1	96/96	0/6.7	2/2	1/1	48/96	0/15.4
138	5	0.58	2/2	1/1	96/128	0/3.9	2/2	1/1	96/96	0/6.7	2/2	1/1	48/96	0/13.2
gzip														
54	20	1.57	4/4	1/1	48/64	0/1.9	4/4	1/1	48/48	0/4.7	2/2	1/1	32/48	0/5.3
13	17	2.17	6/6	1/1	64/64	0/1.9	4/4	1/2	64/64	0/2.8	4/4	1/1	32/48	0/5.3
mcf														
11	18	0.16	2/2	1/1	16/32	0/3.9	2/2	1/1	16/32	0/7.8	2/2	1/1	16/16	0/15.4
33	9	0.05	2/2	1/1	48/48	0/3.9	2/2	1/1	48/48	0/8.9	2/2	1/1	48/48	0/17.6
mesa														
2	79	1.81	6/6	2/2	96/96	0/1.9	6/6	2/2	64/96	0/6.7	2/4	1/2	48/48	0/5.3
21	4	1.75	4/4	2/2	64/64	0/3.9	2/4	1/2	64/64	0/7.8	2/2	1/1	32/64	0/7.14
twolf														
0	57	1.22	4/4	1/1	64/96	0/2.9	2/4	1/1	64/64	0/4.7	2/2	1/1	32/64	0/7.14
8	43	1.42	2/4	1/1	96/96	0/3.4	2/2	1/1	96/96	0/5.3	2/2	1/1	64/64	0/7.14

[a]The % column shows the percentage of time that the program spends in the given phase.

7.1.2.2 *Memory Adaptation.* From Figure 4a, we see that the main reason that memory adaptation shows lower benefit than processor adaptation is that in our system, processor energy dominates in the base case. The memory adaptation algorithm itself is successful in removing a large fraction of the memory energy. Again, most (but not all) of the benefit comes at 5% slack. This situation could change for different systems and applications (e.g., with a server or data center system and workloads [Lefurgy et al. 2003]).

7.1.2.3 *Joint Adaptation—PM versus P and M.* Table V shows that for a given slack, across all applications and across different phases of an application, the joint-adaptation algorithm often chooses different processor configurations and different allocation of slack to memory, seeking to maximize energy savings. Comparing PM to P, we see several cases where the processor configuration changes to increase the processor energy in PM, as the memory adaptation tries to use up some of the slack available to minimize total system energy.

In addition, as observed earlier, PM is able to better exploit increasing slack than either P or M alone. This is explained as follows. For large slack values (e.g., 20% in Figure 4), the energy saving from PM is close to the sum of the saving from P and M. For smaller slack values (e.g., 5% in Figure 4), the energy saving from PM is often less than the sum of the savings from P and M. The reason is that for the 20% case, P and M are able to put processor and memory, respectively, into low-power modes and yet not use up all available slack. There is enough slack that PM is also able to put *both* processor and memory into low-power modes. Thus, PM obtains the individual processor and memory savings from P and M, respectively, to show additive savings. This is not the case with

5% slack, since 5% is too little for *both* processor and memory to go into the same low-power modes, as in the processor only and memory only cases. Thus, the PM energy savings are less than the sum from the individual components given 5% each.

7.1.2.4 *Impact of Slack Distribution—PM versus PMe.* Figure 4b shows that no single slack distribution is optimal for all applications. For example, memory intensive *mcf* uses a slack division so that memory gets a greater portion of the slack. The processor is given about 1, 2, and 6% slack for total slack of 5, 10, and 20%, respectively. Increasing slack beyond a small value for processor-only adaptations is not beneficial because of the memory intensive nature of this benchmark. The extra slack is beneficial for energy savings only if it is used for memory adaptations. On the other hand, the compute intensive *mesa* chooses a slack distribution where the processor gets a greater chunk—about 3, 8, and 16% for total slack of 5, 10, and 20%, respectively.

However, in many (but not all) cases, an equal slack distribution provides energy savings similar to that of the optimal slack distribution algorithm. This can be explained by the shape of the energy versus slack distribution curves in Figure 4b. These curves are often flat in the center, most notably for total slack of 10 and 20%. Since the optimal and equal slack often fall in this flat part, they both give similar energy savings. The reason for the flatness of the curve is as follows. Moving from left to right on the *x* axis, slack is taken away from memory and given to the processor. In general, this causes memory energy to rise and processor energy to drop. In many cases, this increase and decrease tends to negate each other, creating a virtually flat energy curve. Exceptions, however, arise in the following cases, which result in greater savings with optimal slack distribution.

The energy curves in Figure 4b are not flat for highly memory- or compute-intensive applications. In the memory-intensive case (e.g., mcf), the processor energy part of the curve tends to be flat. This is because even with a small slack, these applications already choose the minimum (or close) resources for their architectural configuration (reducing processor resources does not reduce performance if the application is memory-bound). Increasing slack, therefore, does not have any further effect on processor adaptation. The memory energy curve, on the other hand, shows a sharp rise in this case. With more slack, memory adaptation can reduce memory energy significantly, in this case. The net result is a rising total energy curve where the optimal slack distribution is more to the left of the equal distribution, providing significant benefits for PM over PMe.

The compute-intensive applications behave in the opposite way. For example, consider *mesa* with 20% slack. Here the memory energy tends to be flat, since a small amount of slack to memory is sufficient to bring the memory down to low-power mode most of the time—giving memory more slack does not help. On the other hand, processor energy continues to go down with increasing slack to the processor, resulting in a decreasing total energy curve and better savings from PM over PMe.

A final aspect that affects the flatness of the curve is the discrete nature of the energy savings from different configurations. For example, starting from a slack

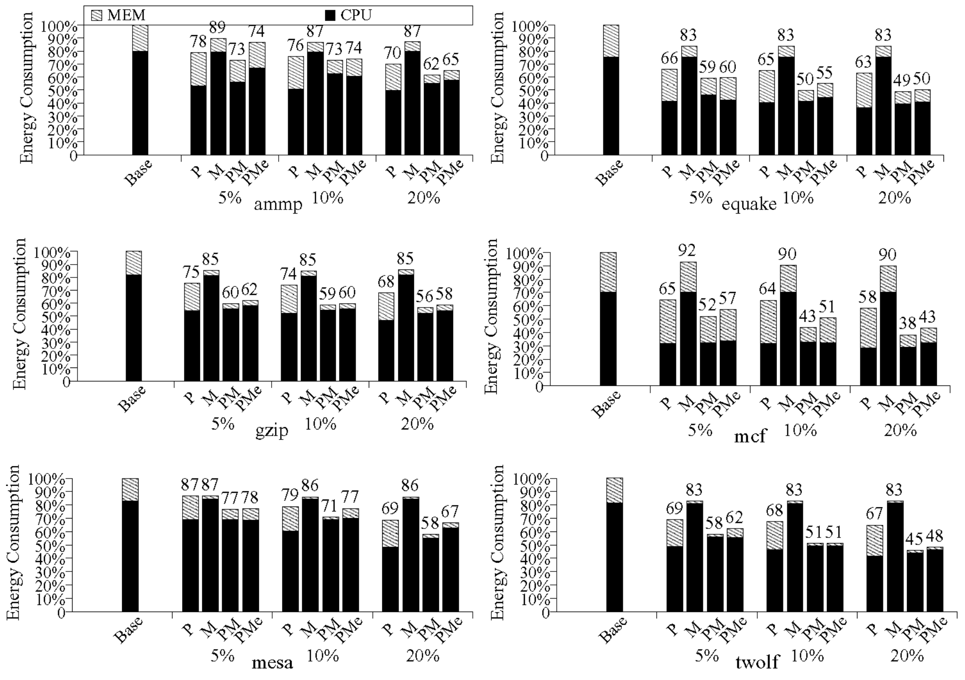

Fig. 5. Energy (normalized to the energy of the base non-adaptive processor) for processor-only (P), memory-only (M), and joint adaptations (PM, PMe) for the full application and three total slack values with simple phase prediction.

Table VI. Relative Energy Savings (in %) for Different Pairs of Algorithms with the Simple Phase Predictor[a]

Specified Slack	5%	10%	20%
P vs. Base	29 [13, 42]	29 [21, 36]	34 [30, 42]
M vs. Base	13 [8, 17]	13 [10, 17]	13 [10, 17]
PM vs. Base	35 [19, 48]	42 [27, 57]	48 [38, 62]
PM vs. P	12 [−3, 20]	19 [3, 32]	22 [11, 34]
PM vs. M	25 [8, 43]	32 [16, 52]	40 [28, 57]
PM vs. PMe	4 [1, 8]	5 [0, 15]	6 [2, 13]

[a]The numbers are *average [min, max]* reduction in energy consumption of the first algorithm over the second.

of 5%, the processor may not be able to exploit an additional 5% slack, but may be able to exploit an additional 10% slack, because of the discrete configuration space available to it. In this case, the processor curve will be flat from 5 to 15%, but could rise sharply at 15%. This results in some jumps between flat parts in the curve and also affects the relative savings from PM to PMe.

7.2 Simple versus Perfect Phase Predictor

Figure 5 shows normalized energy with the simple phase predictor, analogous to Figure 4a. Table VI shows the summary statistics analogous to Table IV. This data shows the same high level trends as with the perfect predictor:

Table VII. Percentage Performance Degradation, Relative to the Base Non Adaptive Architecture

	5%				10%				20%				5%				10%				20%			
Slack	P	M	PM	PMe	P	M	PM	PMe	P	M	PM	PMe	P	M	PM	PMe	P	M	PM	PMe	P	M	PM	PMe
App	Perfect Predictor												Simple Predictor											
ammp	3	3	5	5	7	4	9	9	13	10	15	19	5	4	5	5	10	5	9	10	15	5	7	18
equake	3	1	4	3	4	1	5	4	17	1	18	6	4	1	3	4	5	1	7	6	17	1	17	7
gzip	3	1	5	5	6	1	6	7	16	3	10	8	4	2	4	4	6	2	6	7	13	3	7	6
mcf	3	2	4	4	5	4	6	8	6	5	16	9	5	2	4	5	7	4	6	7	8	6	12	7
mesa	4	3	4	4	6	4	8	8	14	5	17	16	3	5	5	5	9	5	8	9	19	5	19	16
twolf	4	1	4	3	9	2	6	6	18	3	15	11	4	1	4	3	9	2	6	6	18	3	15	11

(1) processor and memory adaptation are effective in all cases, with processor adaptation being more effective than memory adaptation; (2) joint processor and memory adaptation gives significant benefits over either adaptation alone; and (3) appropriate slack distribution for the joint algorithm is important, equal distribution gives almost the same benefits as optimal distribution in many cases, but optimal distribution gives significant benefits over equal distribution, for some cases, at little additional cost.

Comparing the graphs for perfect and simple phase predictors (Figures 4 and 5), we can see that except for *ammp* and *gzip*, even the absolute (normalized) energy consumption of the individual systems is comparable for the two predictors (within a few percentage). This is because the simple predictor gives a high prediction rate on most applications. Further, often even on a misprediction, the configurations chosen are very close to those with the perfect predictor (the case for *equake*).

7.3 Performance Impact

Table VII gives the performance degradation for each case for both the perfect and simple predictors. It shows that the performance guarantee algorithm is effective in all cases. Since the processor configurations are discrete, increasing the amount of available slack does not necessarily result in a linear increase in the actual performance degradation. Furthermore, the algorithm is conservative in estimating the performance degradation to provide the performance guarantee; therefore, in some cases, only a small portion of the allowed slack is used.

8. RELATED WORK

8.1 Cross-Component Joint Adaptation

To our knowledge, there has been little previous work on control algorithms for cross-component joint adaptation for general-purpose applications.

Recent work by Felter et al. [2005] is the most related to our work. That work considers the problem of maintaining a peak power constraint while maximizing performance. The work is based on the insight that different components use their peak power at different times; therefore, judicious allocation of peak power among the components can reduce the overall peak power requirement without much impact on performance. Their technique, called power shifting, dynamically changes the allocation of peak power to processor and memory every interval (e.g., every few 1000 cycles) based on the workload. They achieve this by calculating a maximum number of processor instructions and memory accesses that can be performed each interval based on certain heuristic functions. In contrast, our work seeks to minimize energy for a given performance degradation by distributing performance slack among different components.

Related to cross-component adaptation is the work on cross-layer adaptation for multimedia applications (e.g., Yuan et al. [2003]). The focus of that work is on cooperative adaptation of different system layers such as hardware, applications, network, and the operating system. For example, the GRACE project has proposed an adaptive video encoder application that allows a tradeoff between CPU and network energy and works with an adaptive processor and an adaptive operating system scheduling algorithm. The GRACE control algorithm determines the appropriate configuration for each application, hardware, and a schedule to minimize the total processor and network energy [Sachs et al. 2003; Vardhan et al. 2005]. The network itself is not adaptive and any interaction between the processor and network energy arises from the adaptations in the application. This line of work exploits special properties of multimedia applications. Further, we are not aware of any work that optimizes the total energy of closely interacting components, such as processor and memory.

Fan et al. [2003] studied a system with processor dynamic voltage/scaling (DVS) and memory adaptation for multimedia workloads and showed that there is a positive synergistic effect between DVS and memory adaptation. Their work, however, does not provide any control algorithm to decide the optimal configuration for the processor and memory. Recently, Cho et al. [Cho and Chang 2004] studied a system with processor DVS and nonadaptive memory. They analytically derive an energy-optimal frequency assignment to optimize the total (processor memory) energy. We also optimize for the total energy, but consider a more complex system where both processor and memory are adaptive and where the impact of the processor adaptations is difficult to model analytically.

8.2 Separate Memory and Processor Adaptation

Substantial work has been done on control algorithms for separately adapting memory and processor for saving energy for general-purpose applications.

Section 4.1 already discussed some memory-adaptation algorithms. In addition, Delaluz et al. [2000, 2001] studied compiler-directed techniques, as well as operating-system-based approaches [Delaluz et al. 2002a, 2002b] to reduce the energy consumed by the memory subsystem. Recently, Padnamabhan et al. [Huang et al. 2003b] proposed power-aware virtual memory implementation in operating systems to reduce memory energy consumption.

Most of the processor algorithms exploit fine-grain temporal variability for which it is difficult to predict slack used [Albonesi 1999; Bahar and Manne 2001; Balasubramonian et al. 2000; Buyuktosunoglu et al. 2000; Dhodhapkar and Smith 2002; Dropsho et al. 2002; Folegnani and Gonzlez 2001; Manne et al. 1998; Ponomarev et al. 2001]. Sherwood et al. [2003] performed a brief evaluation of processor energy adaptations at the phase granularity. They focus on providing energy savings without significant performance loss. Our processor algorithm is based on a similar approach, but also allows explicit performance (slack) tradeoffs and we show how to integrate it with a memory-adaptation algorithm. Huang et al. [2000] propose DEETM, which adapts at the granularity of several milliseconds. More recently, they developed an algorithm that adapts at the temporal granularity of subroutines [Huang et al. 2003b]. It uses either offline or online profile information to select the best adaptations for the subroutines for given target slack. Dhodapkar et al. [Dhodhapkar and Smith 2002] and Balasubramonian et al. [2000] propose processor adaptation methods based on variable-sized time intervals. It would be interesting to evaluate subroutine- versus phase-based adaptations, as well as the impact of variable-sized intervals for joint adaptation; however, this is outside the scope of our work.

Dropsho et al. [2002] propose an algorithm to control multiple adaptive resources in the processor and cache. The algorithm controls each structure independently and makes greedy control decisions based solely on the structure's local information. For individual adaptations (specifically caches), the algorithm avoids pathological performance degradations and shares similarities with our performance guarantee algorithm. However, that algorithm does not factor in the interaction and coupled effects between multiple resource adaptations when making control decisions. That work concludes that an important piece of future work is to develop techniques that can capture such global effects and bound the total performance loss. Our joint algorithm considers the interaction between different components and makes decisions based on such global information.

Our processor algorithm is derived from the algorithm developed by Hughes et al. [2001] for real-time multimedia applications. That algorithm adapts at the granularity of each application frame—it picks the lowest energy configuration that will meet the real-time deadline of the frame. Hughes et al. show how to use specific properties of multimedia applications to predict the energy and execution time of the next frame for each architecture configuration. To make the prediction, they (1) initially profile a frame for each architecture configuration, determining the frame's average IPC and energy per instruction, and (2) before each subsequent frame's execution, predict the instruction count for that frame using simple heuristics. With the above information, they can pick the architecture configuration that uses minimal energy and meets the deadline. Our algorithm is also profile based, but we use phases as the granularity of adaptation (instead of frames) and do not need instruction count predictors (because we can directly predict execution time and energy from a profile of a phase). In addition, our algorithm aims to provide a performance guarantee.

8.3 Phase Characterizations

Underlying our work is the requirement for adequate phase-characterization algorithms. We have used the basic block vector (BBV)-based algorithm by Sherwood et al. [2003]. Recently, Isci et al. compared the BBV-based classification method with a performance-monitoring counter (PMC)-based classification method for characterizing power for both SPEC and non-SPEC benchmarks [Isci and Martonosi 2006]. They conclude that the BBV-based method can lead to phase-characterizations where there is some variability in the power usage of the intervals of some phases, particularly for non-SPEC benchmarks. The PMC based method is more accurate for power. Our algorithm can make use of any phase-characterization algorithm, including future improvements to such algorithms. For the applications and systems used in our experiments, we found the BBV-based method to be adequate. We also note that for our purpose, accuracy of performance behavior is arguably more important than accuracy of power, because the former affects the performance guarantee whereas the latter affects the extent of energy savings.

9. CONCLUSIONS AND FUTURE WORK

There has been an explosion of research on energy-driven adaptations for different hardware components for general-purpose applications, but little work on controlling the adaptations of multiple interacting components. To reap the energy benefits of the various proposed adaptations on real systems, it is necessary to determine how to use per-component adaptation algorithms in concert with each other, with bounded performance loss. This paper has taken the first step toward that goal.

We develop a cross-component adaptation control algorithm for processor and memory adaptations. The algorithm assumes a specified target slowdown and seeks to minimize the total processor + memory energy without exceeding this slowdown.

Rather than develop a new algorithm from scratch, we leverage the substantial work in per-component algorithms. We modify a current processor and a memory algorithm so they satisfy the properties of performance guarantee and composability. These properties enable determining an optimal slack allocation for the per-component algorithms, such that these algorithms are able to work independently and yet are able to minimize the total energy of the two components while meeting the target performance constraint. The ability to use two per-component algorithms independently is an important asset since processors and memories are provided by different vendors; a tight coupling between the two would be difficult, in practice.

Our results show that joint adaptation of both components provides significantly more energy savings than adapting either component alone. Further, our results illustrate the importance of distributing the targeted slack among the processor and memory in an intelligent and application-specific way. Our results also clearly demonstrate that our framework is able to maintain the required performance constraint.

Although our focus is not on the per-component algorithms per se, it is worth noting that this work also advances the state-of-the-art in processor adaptation for energy. Most previous processor algorithms for general-purpose applications focused on saving energy without “much” performance loss, but they normally do not try to bound the performance loss. We show how to trade off a targeted amount of performance to save energy. To do so, we assume an application consists of phases that repeat often and an algorithm that can classify phase intervals with stable execution time and energy. There is much recent and concurrent work that exploits similar assumptions and also work developing increasingly accurate phase classification and detection algorithms. We can easily incorporate future improvements in such algorithms within our framework.

There are several avenues for future work. First, currently, we assume a user-defined slack and seek to minimize energy while staying within this slack. We would like to explore explicit performance–energy tradeoffs, and the involvement of the operating system in determining application-specific tradeoffs that can maximize the utility of the entire system. Second, as mentioned earlier, it would be interesting to study alternative time granularities for adaptation, e.g., subroutines as in [Huang et al. 2003b]. Third, we would like to extend this work to other component adaptations (e.g., disk) and to other adaptations in the processor (e.g., dynamic voltage scaling or DVS and other architecture adaptations).[4] Fourth, we would like to use a full system simulator modeling a chip multiprocessor (CMP) to evaluate multithreaded commercial workloads such as databases, to explore if a phase-based approach can be extended to such applications and systems. We would also like to explore alternate phase-characterization strategies that can classify phases more accurately for a broader class of single-threaded applications (e.g., Isci and Martonosi [2006]). Finally, we would like to confirm that operating system effects, such as context switches and interrupts, are properly handled with our algorithms.

Although there are many open questions, most of these must also be resolved for per-component adaptation (e.g., we are not aware of performance-guaranteed processor or memory adaptation on CMPs). Overcoming all the limitations of per-component adaptations is outside the scope of this paper. Rather, our goal has been to provide a framework that will enable per-component algorithms to be used in a cross-component context. We hope that this work will guide the developers of future per-component algorithms in a direction that will make those algorithms amenable to cross-component adaptation.

REFERENCES

Albonesi, D. H. 1999. Selective cache ways: On-demand cache resource allocation. In *Proc. of the 32nd Annual Intl. Symp. on Microarchitecture*.

Bahar, R. I. and Manne, S. 2001. Power and energy reduction via pipeline balancing. In *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*.

Balasubramonian, R. et al. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proc. of the 33rd Annual Intl. Symp. on Microarchitecture*.

Brooks, D. and Martonosi, M. 1999. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proc. of the 5th Intl. Symp. on High Performance Comp. Architecture*.

Brooks, D. et al. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Annual Intl. Symp. on Comp. Architecture*.

Buyuktosunoglu, A. et al. 2000. An adaptive issue queue for reduced power at high performance. In *Proc. of the Workshop on Power-Aware Computer Systems*.

[4]We chose to study architecture adaptations over DVS in this work because it is becoming difficult to scale voltage down further. Nevertheless, it is possible to integrate DVS into our framework.

Carrera, E. V. et al. 2003. Conserving disk energy in network servers. In *Proceedings of the 17th International Conference on Supercomputing*.

Cho, Y. and Chang, N. 2004. Memory-aware energy-optimal frequency assignment for dynamic supply voltage scaling. In *Proc. of the Intl. Symposium on Low Power Electronics and Design*.

Delaluz, V. et al. 2000. Energy-oriented compiler optimizations for partitioned memory architectures. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*.

Delaluz, V. et al. 2001. Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers*.

Delaluz, V. et al. 2002a. Automatic data migration for reducing energy consumption in multi-bank memory systems. In *Design Automation Conference*.

Delaluz, V. et al. 2002b. Scheduler-based dram energy management. In *Design Automation Conference*.

Dhodhapkar, A. and Smith, J. E. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Proc. of the 29th Annual Intl. Symp. on Comp. Architecture*.

Dropsho, S. et al. 2002. Integrating adaptive on-chip storage structures for reduced dynamic power. In *International Conference on Parallel Architectures and Compilation Techniques*.

Fan, X. et al. 2003. Synergy between power-aware memory systems and processor voltage scaling. In *Proc. of the Workshop on Power-Aware Computer Systems*.

Felter, W., Rajamani, K., Keller, T., and Rusu, C. 2005. A performance-conserving approach for reducing peak power consumption in server systems. In *Proc. of the 2005 Intl Conf. on Supercomputing*.

Flinn, J. and Satyanarayanan, M. 1999. Energy-aware adaptation for mobile applications. In *Proceedings of the Symposium on Operating Systems Principles*.

Folegnani, D. and Gonzlez, A. 2001. Energy-effective issue logic. In *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*.

Govil, K., Chan, E., and Wasserman, H. 1995. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proc. of the 1st Intl. Conf. on Mobile Computing and Networking*.

Gurumurthi, S. et al. 2003. Drpm: Dynamic speed control for power management in server class disks. In *Proc. of the 30th Annual Intl. Symp. on Comp. Architecture*.

Huang, M. et al. 2000. A framework for dynamic energy efficiency and temperature management. In *Proc. of the 33rd Annual Intl. Symp. on Microarchitecture*.

Huang, H., Pillai, P., and Shin, K. 2003a. Design and implementation of power-aware virtual memory. In *USENIX Conference*.

Huang, M. C. et al. 2003b. Positional processor adaptation: Application to energy reduction. In *Proc. of the 30th Annual Intl. Symp. on Comp. Architecture*.

Hughes, C. J. and Adve, S. V. 2004. A formal approach to frequent energy adaptations for multimedia applications. In *Proc. of the 31th Annual Intl. Symp. on Comp. Architecture*.

Hughes, C. J. et al. 2001. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proc. of the 34th Annual Intl. Symp. on Microarchitecture*.

Hughes, C. J. et al. 2002. RSIM: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*.

Isci, C. and Martonosi, M. 2006. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques . In *Proc. of the 12th Intl. Symp. on High Performance Comp. Architecture*.

Kravets, R. and Krishnan, P. 1998. Power management techniques for mobile communication. In *MobiCom*. 157–168.

Lebeck, A. R. et al. 2000. Power aware page allocation. In *Proc. of the 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

Lefurgy, C. et al. 2003. Energy management for commercial servers. *IEEE Computer 36*, 12 (Dec.), 39–48.

Li, X. et al. 2004. Performance directed energy management for main memory and disk. In *Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

Lu, Z. et al. 2002. Control-theoretic dynamic voltage and frequency scaling for multimedia workloads. In *Proc. of the 2002 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*.

Manne, S. et al. 1998. Pipeline gating: Speculation control for energy reduction. In *Proc. of the 25th Annual Intl. Symp. on Comp. Architecture*.

Maro, R. et al. 2000. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Proc. of the Workshop on Power-Aware Computer Systems*.

Ponomarev, D. et al. 2001. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proc. of the 34th Annual Intl. Symp. on Microarchitecture*.

Rambus. 1999. Rdram. http://www.rambus.com.

Sachs, D., Adve, S., and Jones, D. 2003. Cross-layer adaptive video coding to reduce energy on general-purpose processors. In *Proc. Intl. Conf. Image Processing (ICIP)*.

Sasanka, R. et al. 2002. Joint local and global hardware adaptations for energy. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

Sherwood, T. et al. 2002. Automatically characterizing large scale program behavior. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

Sherwood, T. et al. 2003. Phase tracking and prediction. In *Proc. of the 30th Annual Intl. Symp. on Comp. Architecture*.

Vardhan, V. et al. 2005. Integrating fine-grained application adaptation with global adaptation for saving energy. In *Proceedings of the 2nd International Workshop on Power-Aware Real-Time Computing (PARC)*. Extended version submitted to the International Journal of Embedded Systems, special issue on “Low Power Real-Time Embedded Computing”.

Yuan, W. et al. 2003. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. In *Proc. SPIE Conf. on Multimedia Computing and Networking (MMCN)*.

Received March 2006; revised November 2006; accepted November 2006