Performance Directed Energy Management for Main Memory and Disks *

Xiaodong Li, Zhenmin Li, Francis David, Pin Zhou, Yuanyuan Zhou, Sarita Adve and Sanjeev Kumar‡

Department of Computer Science
University of Illinois at Urbana-Champaign
{xli3, zli4, fdavid, pinzhou, yyzhou, sadve}@uiuc.edu

‡Intel Labs
Santa Clara, CA
Sanjeev.Kumar@intel.com

ABSTRACT

Much research has been conducted on energy management for memory and disks. Most studies use control algorithms that dynamically transition devices to low power modes after they are idle for a certain threshold period of time. The control algorithms used in the past have two major limitations. First, they require painstaking, application-dependent manual tuning of their thresholds to achieve energy savings without significantly degrading performance. Second, they do not provide performance guarantees. In one case, they slowed down an application by 835%!

This paper addresses these two limitations for both memory and disks, making memory/disk energy-saving schemes practical enough to use in real systems. Specifically, we make three contributions: (1) We propose a technique that provides a performance guarantee for control algorithms. We show that our method works well for all tested cases, even with previously proposed algorithms that are not performance-aware. (2) We propose a new control algorithm, *Performance-directed Dynamic* (PD), that dynamically adjusts its thresholds periodically, based on available slack and recent workload characteristics. For memory, PD consumes the least energy, when compared to previous hand-tuned algorithms combined with a performance guarantee. However, for disks, PD is too complex and its self-tuning is unable to beat previous hand-tuned algorithms. (3) To improve on PD, we propose a simple, optimization-based, threshold-free control algorithm, *Performance-directed Static* (PS). PS periodically assigns a static configuration by solving an optimization problem that incorporates information about the available slack and recent traffic variability to different chips/disks. We find that PS is the best or close to the best across all performance-guaranteed disk algorithms, including hand-tuned versions.

*This work is supported in part by an equipment donation from AMD, an IBM SUR grant, a gift from Intel Corp., and the National Science Foundation under Grant No. CCR-0096126, EIA-0103645, EIA-0224453, CCR-0209198, CCR-0205638, EIA-0224453, CCR-0305854, and CCR-0313286.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'04, October 9–13, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-804-0/04/0010 ...$5.00.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Miscellaneous; C.4 [**Performance of System**]

General Terms

Algorithms

Keywords

memory and disk energy management, low power design, adaptation algorithms, control algorithms, multiple power mode device

1. INTRODUCTION

Energy consumption has emerged as an important issue in the design of computing systems. For battery-operated mobile devices, energy consumption directly affects the battery life, and for high-end data centers, the annual cost of energy is increasing dramatically [28].

The storage hierarchy, which includes memory and disks, is a major energy consumer in computer systems. This is especially true for high-end servers at data centers [5, 14, 23]. Recent measurements from real server systems show that memory could consume 50% more power than processors [24]. A recent industry report shows that storage devices at a data center account for almost 27% of the total energy consumed [1].

To reduce energy consumption, modern memory such as RDRAM allows each individual memory device to transition into different low-power operating modes [32]. Similarly, many disks also support several low-power operating modes [19, 30]. Gurumurthi et al. have recently proposed a multi-speed disk model to further reduce disk energy consumption [14]. Transitioning a device (a memory chip or a disk) to a low power mode can save energy, but can degrade performance. The key to the effective use of these low-power modes, therefore, is an effective control algorithm that decides which power mode each device (memory chip or disk) should be in at any time. This paper concerns effective control algorithms for memory and disk energy management.

Memory and disks share many similarities in their low-power operating modes and their cost/benefit analysis; therefore, we take a common view of the energy management problem in both of these subsystems. All the solutions we propose here are effective for both subsystems; however, the best solution is different for each. For simplicity, we use the term ***storage*** to refer to both memory and disk and the term ***device*** to refer to a single memory chip or disk.

The best previously proposed storage energy control algorithms monitor usage (e.g., through idle time or through degradation of response time), and move to a different power mode if this usage function exceeds (or is less than) a specified threshold [5, 14, 23]. In general, the number of thresholds depends on the number of power modes as well as the number of usage functions monitored. Thus, the thresholds are a key feature of these algorithms. Although these algorithms are effective in reducing energy, two problems make them difficult (if not impossible) to use in practice, as discussed below.

1.1 Limitations of the State-of-the-Art

There are two key problems with current threshold-based energy management algorithms.

(1) Painstaking, application-dependent manual tuning of thresholds: Threshold-based algorithms require manual tuning of the threshold values. Section 5.2 shows that reasonable threshold values are highly application-dependent. For example, for the memory subsystem, a set of thresholds derived from competitive analysis proposed in [23] showed a performance degradation of 8% to 40% when applied to six SPEC benchmarks. Similarly, a set of thresholds tuned in [23] for a system similar to ours but for a different set of applications showed a degradation of 4% to 835% for three SPEC benchmarks! During our hand tuning efforts as well, we repeatedly found that the best threshold values for a given application caused high performance degradation in others; for example, the best set for *gzip* gave 63% degradation for *parser* (Section 5.2).

(2) No performance guarantee: Even if a system were designed with thresholds tuned for an expected set of applications, there is no mechanism to bound the performance degradation for applications that may deviate from the behavior used for tuning. As discussed above, the potential performance degradation with the "wrong" set of thresholds can be very high (up to 835% in our experiments). This type of unpredictable behavior or lack of a safety net is clearly a problem for all users. However, it can be particularly catastrophic for high-end servers in host data centers that have to honor service level contracts with customers. Such data centers are becoming increasingly important consumers of high-end servers, and the ability to provide some form of performance guarantee is crucial for their business models to be viable. Furthermore, as motivated earlier, it is exactly in such high-end server scenarios that reducing memory and disk energy can lead to significant cost savings.

1.2 Contributions of this Work

In this paper, we decouple the above two problems and provide solutions to both in an orthogonal way:

(1) Technique to guarantee performance: First, we propose a new technique that guarantees that performance will not be degraded by the underlying control algorithm beyond a specified limit. It dynamically monitors the performance degradation at run time and forces all devices to full power mode when the degradation exceeds the specified limit. This performance guarantee algorithm can potentially be combined with any underlying control algorithm for managing the system power modes. Furthermore, the presence of this algorithm enables making conscious, user-specific tradeoffs in performance for increased energy savings. In this paper, we allow the user to specify an acceptable slowdown, and the system seeks to minimize energy within this constraint.

We evaluated our algorithm for memory and disk using simulation and report results for more than 200 scenarios. In each case, the algorithm successfully limits the performance degradation to the specified limit.

(2) A self-tuning thresholds based control algorithm (called PD): Second, we develop an algorithm that automatically tunes its thresholds periodically, eliminating the need for hand tuning. The period for tuning is a large number of instructions, referred to as an *epoch*. At every epoch, the algorithm changes its thresholds based on the insight that the optimal thresholds are a function of the (predicted) access traffic and acceptable slowdown that can be incurred for that epoch. We refer to this algorithm combined with the performance-guarantee algorithm as PD (for Performance-directed Dynamic).

We compare PD with the original threshold-based algorithm [23] without and with performance guarantee (referred to as OD and OD+ respectively, where OD stands for Original Dynamic). We find that for memory, PD consumes the least energy of all performance-guaranteed algorithms (up to 68% less than the best OD+). Even compared to the best hand tuned OD (no performance guarantee), PD performs well in most cases without any manual tuning and providing a performance guarantee. For disks, however, PD does not perform as well because the number of parameters involved is much larger than for memory, making it too complex to self-tune all parameters dynamically. Thus, the self-tuned algorithm is unable to compete with the hand-tuned one in a few cases.

(3) A simpler, optimization based, thresholds-free control algorithm (called PS): Since PD is relatively complex (although not much more complex than the original algorithms) and is still primarily based on heuristics to determine the best thresholds, we also explore a relatively simpler algorithm based on formal optimization. Like PD, this algorithm also works on an epoch granularity. However, it eliminates the thresholds-based nature of the dynamic algorithm by choosing a single configuration for each device for the entire epoch. We refer to this algorithm as PS because it is inspired by the static algorithm (referred to as OS) proposed in [23]. OS uses a fixed configuration for all devices throughout the entire execution. In contrast, PS exploits variability in space by assigning different modes (configurations) to different devices, and also exploits variability in time by reassigning configurations at the start of a new epoch. At each epoch, the configuration is chosen by mapping this problem to a constrained optimization problem. Applying standard optimization techniques, we can achieve a close to optimal solution (for fixed configurations through an epoch) without resorting to complex heuristics.

For memory, as mentioned earlier, PD performs very well and PS is not competitive. For disks, PS is the best or close to the best in all but one case, when compared to all performance-guaranteed algorithms studied here.

Summary: Overall, this paper makes a significant step towards making control algorithms for memory/disk energy conservation usable in real systems, especially systems such as data centers that require service guarantees. We do this by eliminating the need for painstaking, application-dependent parameter tuning, and by minimizing energy while providing a performance guarantee. With our schemes, users never need to worry about whether the underlying energy conservation scheme may degrade the performance by some unpredictable values such as 835% (as seen by some previous threshold-based schemes)!

Due to space limitations, Sections 2 – 5 focus entirely on memory energy management. Section 6 then relates this work to disks and presents results for disk energy management.

2. BACKGROUND

Memory Power Model: We base the power model for the memory subsystem on recent advances that have yielded memory chips capable of operating in multiple power modes. In particular, our

model follows the specifications for Rambus DRAM (RDRAM) [32]. Each RDRAM chip can be activated independently. When not in active use, it can be placed into a low-power operating mode to save energy. RDRAM supports three such modes: *standby*, *nap*, and *powerdown*. Each mode works by activating only specific parts of the memory circuitry such as column decoders, row decoders, clock synchronization circuitry, and refresh circuitry (instead of all parts of the chip) [32]. Data is preserved in all power modes. More details of the workings of these modes can be found elsewhere [32, 35] and are not the focus of this paper.

An RDRAM chip must be in active mode to perform a read or write operation. Accesses to chips in low-power operating modes incur additional delay and energy for bringing the chip back to active state. The delay time varies from several cycles to several thousand cycles depending on which low power state the chip is in. In general, the lower the power mode, the more time and the more energy it takes to activate it for access.

Previous Control Algorithms: Previous control algorithms for storage energy management can be classified into two groups: static and dynamic. Static algorithms always put a device in a fixed low-power mode. A device only transitions into full power mode if it needs to service a request as in the memory case. After a request is serviced, it immediately transitions back to the original mode unless there is another request waiting. Lebeck et al. have studied several static algorithms that, respectively, put all memory chips in a standby, nap and powerdown mode [23]. Their results show that the static nap algorithm has the best $Energy \times Delay$ values. We refer to their static algorithms as OS, and specifically to the versions that invoke the static standby, nap, and powerdown configurations as OSs, OSn, and OSp respectively.

Dynamic algorithms transition a device from the current power mode to the next lower power mode after being idle for a specified threshold amount of time (different thresholds are used for different power modes). When a request arrives, the memory chip transitions into active mode to service the request (and then waits for the next threshold period of idle time to transition to the next lower power mode). Lebeck et al. have shown that dynamic algorithms have better energy savings than all static algorithms [23].

The dynamic algorithms for the modeled RDRAM style power modes require three thresholds for three different transitions: active to standby, standby to nap, and nap to powerDown. As described in Section 1, the energy consumption and performance degradation are very sensitive to these thresholds, and manually tuning these parameters for each application is not easy.

3. PROVIDING PERFORMANCE GUARANTEES

Although there is clear motivation for providing performance guarantees (Section 1), the appropriate metric and methodology for measuring delivered performance is unclear. For example, absolute guarantees on delivered MIPS, MFLOPS, IPC, transactions per second, etc. all depend on a priori knowledge of the workload, which may be hard to ascertain. This issue, however, is independent of whether the system employs energy management techniques and outside the scope of this paper (although adding energy management may add further complexity). In our work, we assume that the user has been guaranteed some base "best" performance assuming no energy management, and has an option to further save cost (i.e., energy) by accepting a slowdown relative to this best offered base performance. We assume such an "acceptable slowdown" as an input to our system and refer to it as $Slowdown_{limit}$ (expressed as a percentage increase in execution time, relative to the base). We can envisage future systems where the operating system automatically assigns appropriate slowdown to different workloads based on utility functions that incorporate appropriate notions of benefits and costs, but again, such work is outside the scope of this paper. We can also extend this work by letting the user specify an acceptable tradeoff between performance and energy (e.g., slowdown X% for Y% energy savings). In this paper, we choose to minimize energy within the acceptable slowdown.

Given $Slowdown_{limit}$, the goal of the performance-guarantee algorithm is to ensure that the underlying energy management algorithm does not slow down the execution beyond this acceptable limit. Thus, there are two key components to the performance-guarantee algorithm – (1) estimating the actual slowdown due to energy management, and (2) enforcing that the actual slowdown does not exceed the specified limit.

3.1 Estimating Actual Slowdown – Key Idea

At each memory access, the performance-guarantee algorithm estimates the absolute delay in execution time due to energy management, and then determines if the resulting percentage slowdown so far is within the absolute limit. We use the following terms:

- t = Execution time using the underlying energy management algorithm until some point P in the program.
- $T_{base}(t)$ = Execution time without any energy management until the same point in the program.
- $Delay(t)$ = Absolute increase in execution time due to energy management = $t - T_{base}(t)$
- Actual percentage slowdown = $\frac{Delay(t)}{T_{base}} * 100 = \frac{Delay(t)}{t - Delay(t)} * 100$

The constraint that must be ensured by the performance-guarantee algorithm is Actual slowdown $\leq Slowdown_{limit}$. That is,

$$\frac{Delay(t)}{t - Delay(t)} * 100 \leq Slowdown_{limit}$$

So conceptually, the only unknown to be determined is the delay. To guarantee the performance requirement, the delay estimation should be as accurate as possible, but *conservative* (estimated delay $\geq$ actual delay).

Naive method: A simple way to estimate delay is to sum up the delay for each access. Here, the *delay* for an access that arrives at a device in low power mode is the transition time from the current low power mode to active mode (required for it to be serviced).

Refinements: Although the above method can estimate delay, it is too conservative because it does not consider the effect of overlapped accesses and other latency hiding techniques in modern processors (e.g., out-of-order/decoupled execution, prefetching, non-blocking caches, and write-buffers). Such techniques can hide a portion of the memory access latency, resulting in an actual program slowdown that is much smaller than the sum of the slowdowns for each access.

In general, it is difficult to account for all the latency hiding techniques because there are too many uncertain factors. We refine our program slowdown estimation method to consider some of the major sources of inaccuracy.

The first refinement is to exploit information about overlapped or concurrent requests. For example, in Figure 1, access A is first issued to device 0. Before A finishes, another request, B, can be issued to device 1. Suppose both devices are in low-power mode and therefore access A is delayed by D_1 time units and B is delayed by D_2 time units. Obviously, the total delay in execution time will be smaller than $D_1 + D_2$. A tighter bound is D, the value obtained by subtracting the overlapped time from $D_1 + D_2$. This idea can

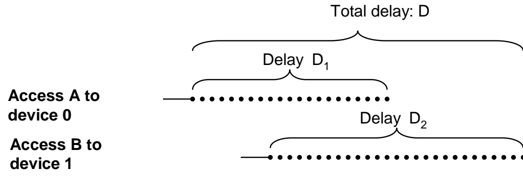

Figure 1: An example of overlapped requests and refinement of the delay estimate.

be extended to multiple overlapped requests, and is incorporated in our delay estimation.

Second, writes to memory are mostly write-backs from cache line displacement, and most cache architectures can perform these in the background. It is therefore unnecessary to consider the extra delay for memory writes unless there is another request waiting to access the same chip. Third, modern processors often do not block on store instructions. Delays of memory reads caused by `store` instructions therefore need not also be considered unless there is memory contention.

3.2 Enforcing Performance Guarantee

It is useful to define another term:

$$\begin{aligned} Slack(t) = &\text{ The amount of allowed execution delay that would not violate the slowdown constraint} \\ = &\ T_{base}(t) * Slowdown_{limit}/100 - Delay(t) \\ = &\ (t - Delay(t)) * Slowdown_{limit}/100 - Delay(t) \end{aligned}$$

Simple Method: A simple way to enforce the performance guarantee is to ensure that slack is never negative. If slack goes negative, the performance-guarantee algorithm disables the underlying energy management algorithm, pulling all devices to full power mode. The system continues like this until enough slack is accumulated to activate the underlying control algorithm again.

The performance-guarantee algorithm described so far can be coupled with any energy management algorithm, in principle. In the general case, (e.g., with OS and OD), the delay and slack updates are performed at each access. If the slack is negative, the underlying control mechanism is temporarily disabled until enough slack is generated. This value of "enough slack" is a new parameter for the algorithm that may need to be tuned, for the general case.

Refinement: The above method has two limitations. First, it relies on a new tunable parameter called "enough slack". Second, it has to check the actual *percentage* slowdown against the slack limit (at least one division and one comparison) after every access, incurring too much overhead.

To overcome the above two limitations, a refinement is to break the execution time into epochs. An epoch is a relatively large time interval over which the application execution is assumed to be predictable. In our experiments, we set the epoch length to be 1 million instructions (as reported in Section 5.1, we found that our results are not very sensitive to the epoch length). At the start of an epoch, the algorithm estimates the *absolute* available slack for the entire epoch (as shown below). Now, after each access, the algorithm only needs to check the actual absolute delay so far in this epoch against the estimated available slack for the entire epoch. If the actual delay is more than the available slack, all devices are forced to active power mode until the end of the epoch. This method does not need the "enough slack" parameter, and avoids the division computation at each access. Since our two new control algorithms (Section 4) are already epoch-based, it is fairly easy to use this refinement.

The available slack for the next epoch can be estimated based on the $Slowdown_{limit}$ specified by the application and the predicted execution time of the next epoch without power management (denoted t_{epoch} and predicted to be the same as for the last epoch). The available slack for the next epoch needs to satisfy the following constraint:

$$\frac{AvailableSlack + Delay(t)}{t - Delay(t) + t_{epoch}} * 100 \leq Slowdown_{limit}$$

.

Solving this for $AvailableSlack$, we have

$$\begin{aligned} AvailableSlack \leq &\ \frac{Slowdown_{limit}}{100} \times t_{epoch} \\ &+ \frac{Slowdown_{limit}}{100} \times (t - Delay(t)) - Delay(t) \end{aligned}$$

The first part above is the next epoch's fair share of the allowed slowdown and the second part is the leftover slack carried forward from the previous epochs. So if the previous epochs have not used up their share of slack, this epoch can afford to use more than its fair share. Conversely, if the previous epoch used up too much slack (e.g., because of incorrect prediction of the epoch length), the next epoch will attempt to make up that slack. Overshooting of the slack by the last few epochs of the program may be difficult to compensate. However, if the program runs for a reasonably long time (as in data center workloads), any error introduced by this is relatively small and in fact negligible. In our experiments, we found our method of reclaiming slack from previous epochs to be very effective in conserving energy while providing a performance guarantee.

3.3 Implementation and Overhead

The above performance guarantee method can be implemented in the memory controller. The memory controller keeps track of the actual delay for each epoch. After each access, based on the delay estimation described in Section 3.1, it updates the actual total delay. This is then compared against the available slack for this epoch; if the former is larger, all chips are forced to active mode. At the end of an epoch, it calculates the available slack for the next epoch using the delay estimate and the equation above.

The overhead for this method is quite small, consisting of only 1-2 integer comparisons and fewer than 4 arithmetic additions or subtractions per access. The available slack calculation requires some multiplications, but occurs once each epoch; this overhead is therefore amortized over a large interval and is negligible.

Although our method assumes a single memory controller that manages all memory chips, we can extend it to systems with multiple memory controllers. This requires optimally spreading the total available slack across all controllers to minimize total energy consumption, using a method similar to our new PS control algorithm presented in Section 4.1.

4. CONTROL ALGORITHMS

This section presents two new control algorithms, Performance-directed Static algorithm (PS) and Performance-directed Dynamic algorithm (PD). Based on awareness of slack available during program execution, these algorithms tune themselves to be more or less aggressive, resulting in higher energy savings. Consequently, they do not require extensive manual tuning. Both algorithms provide a performance guarantee using the method described in Section 3, and use the slack information generated by the performance-guarantee algorithm to guide energy management.

As described in Section 3, we divide the execution into epochs. At the end of each epoch, the performance-guarantee algorithm estimates the available slack for the next epoch. Both energy control algorithms use this slack as a guide.

4.1 The PS Algorithm

PS is inspired by previous static algorithms in [23], but improves them using two insights. First, like OS, PS assigns a fixed configuration (power mode) to a memory chip for the entire duration of an epoch. The chip transitions into active mode only to service a request. However, unlike OS, PS allows this configuration to be changed at epoch boundaries, based on available slack. Thus, PS can adapt to large epoch-scale changes in application behavior. Second, unlike OS, PS allows different configurations for different devices. This allows PS to exploit variability in the amount of traffic to different storage devices – PS effectively apportions total slack differently to different chips.

4.1.1 Problem Formulation and PS Algorithm

The goal of the PS algorithm is to choose, for each chip, i, a configuration (power mode), C_i, that maximizes the total energy savings subject to the constraint of the total available slack for the epoch. That is:

$$\textbf{maximize} \quad \sum_{i=0}^{N-1} E(C_i)$$
$$\textbf{subject to} \quad \sum_{i=0}^{N-1} D(C_i) \leq AvailableSlack$$

where $E(C_i)$ is a prediction of the energy that will be saved by keeping device i in configuration C_i in the next epoch, $D(C_i)$ is a prediction of the increase in execution time due to keeping device i in configuration C_i in the next epoch, and $AvailableSlack$ is a prediction of the slack available for the next epoch. N is the total number of devices.

The prediction of $AvailableSlack$ is obtained from the performance guarantee algorithm as discussed in Section 3.2. The predictions for $E(C_i)$ and $D(C_i)$ are described in the next sections. With the knowledge of $E(C_i)$ and $D(C_i)$, the above equations represent the well-known multiple choice knapsack problem (MCKP). Although finding the optimal solution is NP complete, several close-to-optimal solutions have been proposed [27]. In this work, we use a linear greedy algorithm (we omit details of the solution due to space limitations). The overall PS algorithm is summarized as Algorithm 1.

Algorithm 1 PS Algorithm (called at the beginning of each epoch)

1: Predict $AvailableSlack$ for the next epoch (obtained from the performance-guarantee algorithm).
2: Predict $E(C_i)$ and $D(C_i)$ for each device i and for each available power mode assigned to C_i.
3: Solve the knapsack problem.
4: Set the power mode for each device for the next epoch, as determined by the knapsack solution.

4.1.2 Estimation of $D(C_i)$ and $E(C_i)$

For each device i, for each possible power mode, we need to estimate the energy that would be saved and the delay that would be incurred for the next epoch. For an accurate estimation of these terms, we would need to predict the number and distribution (both across chips and in time) of the accesses in the next epoch.

For the number of accesses in the next epoch and the distribution of these accesses across the different chips, we make the simple assumption that these are the same as in the last epoch. We found this assumption to work well in practice since the epoch lengths are relatively large and the number of accesses to each device changes slowly over epochs. For example, figure 2 shows the access count for 1500 epochs (1 million instructions each) for the application *vortex* (see Section 5.1 for experimental methodology). The figure shows that for the most part, the access rate remains relatively stable for each device; the figure also clearly shows the importance of distinguishing between different devices. There are, however, some bursty periods where the access count changes abruptly for a short time. These will result in sub-optimal configurations. The performance-guarantee algorithm, however, compensates for this. If the access count is under-predicted and the power mode is too low, the performance-guarantee algorithm will force the device to go active. Conversely, if the access count is over-predicted and too little slack is used up, the performance-guarantee algorithm will reclaim the leftover slack for the next epoch.

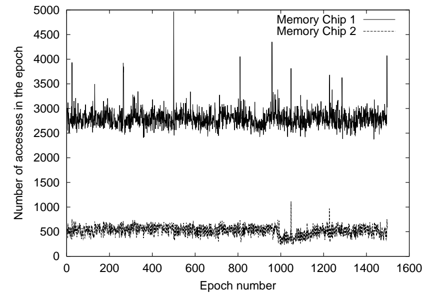

Figure 2: Access counts per epoch to 2 different memory chips for *vortex* with 1 million instruction epochs.

For estimating the temporal distribution of accesses, we make a simplifying assumption that accesses to a given chip are uniformly distributed in time and there is no overlap among accesses to the same or different chips. This is clearly a simplistic assumption; however, more accurate information would require prohibitively expensive tracking of access time distribution in the previous epochs or some as yet unavailable analytic models. The assumption, although simplistic, is strictly conservative. That is, a non-uniform distribution for a given device provides more opportunities for energy saving and reduces delay (since a single activation of the chip can potentially handle multiple buffered requests). Similarly, ignoring overlap among accesses to different chips also over-estimates delay (as explained in Section 3.1). Nevertheless, note again that the performance-guarantee algorithm can again partly compensate for some of the resulting sub-optimality by reclaiming any unused slack for the subsequent epoch. The performance-guarantee algorithm is able to account for overlap in its slack calculation as described in Section 3.2 (the difference is that this overlap is determined as it occurs in the actual execution, while the above discussion is concerned with predicting overlap for the future, which is more difficult).

With the above assumptions, we can now calculate the contribution of device i in power mode P_k (i.e., $C_i = P_k$) to the overall execution time delay as:

$$D(P_k) = A_i * (t_{access}(P_k) - t_{access}(P_{active})) \quad (1)$$

where A_i is the predicted number of accesses to device i in the next epoch, $t_{access}(P_k)$ is the average device access time for power mode k, and $t_{access}(P_{active})$ is the average active mode device access time.

The energy saved by placing device i in power mode P_k can also be calculated in a similar way.

4.1.3 Enforcing Performance Guarantee

In addition to the performance guarantee algorithm described in Section 3, PS also provides the ability to have a chip-level "performance watchdog." The PS optimization described above essen-

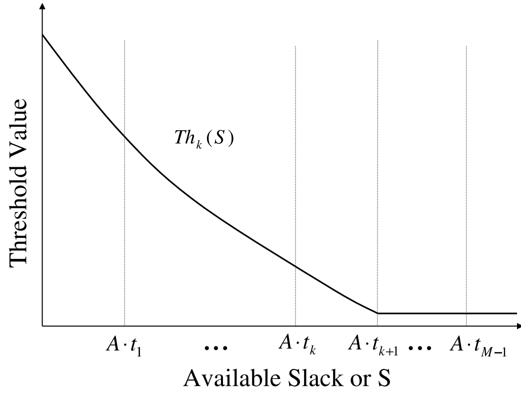

Figure 3: Threshold function $Th_k(S)$. A is the access count for the next epoch. t_i is the transition time from power mode i to active.

tially apportions a part of the available slack to each chip (D(i,P_i)). To provide finer grained control, we also keep track of the actual delay that each chip incurs and compare with this predicted (apportioned) delay. If the former ever exceeds the latter, that chip is forced to the active or full power mode until the end of the epoch. This ensures that one "bad" device (a device that uses up its slack too fast) does not penalize all other devices. This turns out to subsume the guarantee provided by the global (cross-chip) algorithm of Section 3. However, we still use that algorithm to determine $AvailableSlack$ to apportion over the next epoch, since that algorithm accounts for overlap and other refinements discussed in Section 3.

4.1.4 Overhead of the PS algorithm

Similar to the performance guarantee method, PS can also be implemented in a memory controller with some processing power. Many memory controllers (e.g., the Impulse memory controller [38]) already contain low-power processors.

At the beginning of each epoch, PS has to first evaluate $D(C_i)$ and $E(C_i)$ for all devices. This requires $3MN$ multiplications and a similar number of additions, where M is the number of power modes (usually less than 5) and N is the number of chips (less than 16 usually). Since we use a linear greedy algorithm to solve the knapsack problem and $D(C_i)$ and $E(C_i)$ are monotonically non-decreasing, the overhead of the algorithm is not significant. On average, it requires $2MN$ computation steps, with each step consisting of 1-2 integer comparisons and 1-2 subtractions.

Since PS is invoked only at the beginning of each epoch (1 million instructions in our experiments), PS's overhead is amortized over the entire epoch.

4.2 The PD Algorithm

The PS algorithm maintains a single configuration for a device throughout an epoch; hence, it does not exploit temporal variability in the access stream within an epoch. The PD algorithm seeks to exploit such variability, inspired by previous dynamic algorithms which transition to lower power modes after being idle for a certain threshold period of time [23]. However, unlike previous dynamic algorithms, PD automatically re-tunes its thresholds at the end of each epoch, based on available slack and workload characteristics. Further, PD also provides a performance guarantee using the method described in Section 3.

4.2.1 Problem Formulation and PD Algorithm

For the PD algorithm also, we can map the problem to a constrained optimization problem using the same equations as for PS in Section 4.1.1. The difference, however, is that now the configuration, C_i, for device i is described by thresholds Th_1, Th_2, ..., Th_{M-1}, where M is the number of power modes and Th_i is the amount of time the device will stay in power mode $i-1$ before going down to power mode i. A key difference between PS and PD is that the search space for this problem is prohibitively large (a total of $M-1$ threshold variables and each variable could be any integer between $[0, \infty]$). In the absence of an efficient solution for this large space, we consider a heuristics based technique.

First, we curtail the space of solutions by using the same set of thresholds for all devices in a given epoch. Second, we observe that the thresholds must have a first-order dependence on the available slack for the epoch as well as the number of accesses. Specifically, for larger slack, devices can go to lower power modes more aggressively (i.e., thresholds can be smaller) since a larger delay can be tolerated. Similarly, for a given slack S, lower access counts allow for lower thresholds since they cause a lower total delay. There is also a strong dependence on the distribution of accesses; however, as explained in Section 4.1.2, it incurs too much overhead to predict this distribution and so we do not exploit it.

Thus, we seek to determine Th_k as a function of available slack and access count (for each k, $1 \le k \le M-1$). Given that both available slack and access count can be predicted from techniques in previous sections, we reduce our problem to determining Th_k as a function of slack S for a given access count. The next section shows how to determine this function. The overall PD algorithm is summarized as Algorithm 2.

Algorithm 2 PD Algorithm (called at the beginning of each epoch)

1: Predict $AvailableSlack$ for the next epoch (see Section 3.2)
2: Predict number of accesses for the next epoch (same as measured access count for the last epoch)
3: Adjust the functions for $Th_k(S)$ ($1 \le k \le M-1$) for the access count measured from the last epoch
4: **for** $k = 1, ..., M-1$ **do**
5: Use the $Th_k(S)$ functions to determine the values for Th_k, given the value for $AvailableSlack$
6: **end for**
7: Set thresholds $Th_1, ..., Th_{M-1}$ for all chips

4.2.2 Selection of function $Th_k(S)$

In general, if the number of accesses A is fixed, $Th_k(S)$ is monotonically non-increasing with respect to available slack S, as shown in Figure 3. To reduce the computation complexity, we approximate $Th_k(S)$ using multiple linear segments. We first consider M key values of slack S and approximate Th_k for each of these values. These values are $S = A \cdot t_i$ where $0 \le i \le M-1$ (we assume $t_0 = 0$). This divides the available slack or S axis into M distinct intervals – $[0, A \cdot t_1]$, $\cdots$, $[A \cdot t_{M-2}, A \cdot t_{M-1}]$, $[A \cdot t_{M-1}, \infty]$. We use the approximated values of the function at the various $A \cdot t_i$'s to interpolate the values of the rest of the points in the corresponding intervals. For function $Th_k(S)$, these approximate values and interpolations are determined as below.

Consider the key identified slack values $A{\cdot}t_i$, where $i > k$. These values imply available slack that is large enough to allow every access to wait for the transition time t_k. Therefore, ideally, Th_k should be 0 in this case; however, in practice, we found this does not work very well for two reasons. First, the inter-arrival times

of accesses are not uniformly distributed. Using a zero threshold wastes energy when the inter-arrival time is too short to justify the power-up/down cost. In this case, it can be more effective to keep the chip active during the short idle time. Second, the prediction for the number of future accesses may not be accurate. Therefore, we set the minimal threshold for Th_k to be the energy break-even point described in [20]. This provides the 2-competitive property in the worst case energy consumption.

Now consider the remaining identified slack values; i.e., $A \cdot t_i$, where $0 \leq i \leq k$. For these cases, the available slack is either not enough or just barely enough for each access to wait for the transition time t_k. Therefore, we need to be conservative about putting a device in mode k – unless the device is already idle for a long time, we should not put it in mode k. To achieve this, we should set the threshold $Th_k(A{\cdot}t_i)$ to be much larger than for $Th_k(A{\cdot}t_{k+1})$. Further, the lower the value of i, the higher we should set the threshold. We propose setting the threshold to $C^{k-i} \cdot t_k$ because it satisfies all the qualitative properties of Th_K discussed above. Here C is a dynamically adjusted factor discussed later.

Now we have the approximate values for the values of available slack $S = A \cdot t_i$, $0 \leq i \leq M-1$. For an available slack value S in an interval $(A \cdot t_{i-1}, A \cdot t_i)$ where $0 < i < M-1$, we determine the value of $Th_k(S)$ by a linear iterpolation between the end-points of the interval. For available slack values S in the interval $(A \cdot t_{M-1}, \infty]$, we determine the value of $Th_k(S)$ to be the same as that at $S = A \cdot t_{M-1}$.

The remaining problem is of choosing the C factor. PD uses feedback-based control to dynamically adjust the constant value C at run time. If during the last epoch, the system does not use up all its slack, it indicates that the thresholds are too conservative. So PD reduces the constant value C to 95% of the current value to reduce the threshold values. Next epoch, the chip will go to lower power mode more aggressively. On the other hand, if during the last epoch, the system used up all its slack and forces all devices to become active in order to provide a performance guarantee, it indicates that the thresholds are aggressive. So PD doubles the constant to increase the threshold values. Our experimental results show that our dynamic threshold adjustment scheme works very well and we never need to tune the adjustment speeds for decreasing and increasing constant C. The selection of 95% and a factor of 2 are based on insights from the TCP/IP congestion control method.

4.2.3 Overhead of the PD algorithm

At the beginning of each epoch, for each threshold Th_k, PD needs to first compare the current slack with the k key points to see which segment of $Th_k(S)$ we should use. This involves less than $M-1$ comparisons for each $Th_k(S)$ function. So the total number of comparisons is less than M^2. Then we evaluate the linear functions, which takes 4-5 multiplications, divisions and additions. So the total computational complexity is smaller than $M^2 + 5M$. (M is number of power modes, smaller than 5).

Similar to PS, the threshold adjustment in PD is only performed at the beginning of each epoch. Therefore, its overhead is amortized over 1 million instructions.

5. RESULTS FOR MEMORY ENERGY MANAGEMENT

5.1 Experimental Setup

We enhanced the SimpleScalar simulator with the RDRAM memory model [3]. Table 2 gives the processor and cache configuration used in our experiments. There are a total of 4 256Mb RDRAM chips in our system. Table 3 shows the energy consumption and resynchronization time for the RDRAM chips we simulate. The numbers are from the latest RDRAM specifications [32]. We use the power-aware page allocation suggested in [23].

We evaluate our algorithms using execution-driven simulations with SPEC 2000 benchmarks. There are two main reasons for choosing the SPEC benchmarks. First, our infrastructure does not support an operating system, so we cannot run more advanced server-based applications. We are in the process of building a full system simulator based on SIMICS [26] to study the effects of data-center workloads. Second, the use of SPEC benchmarks makes it easier to compare our results with previous work on memory energy management which also used the same benchmarks [7, 23]. In particular, for the dynamic algorithms, we evaluate the threshold settings found to perform well in previous studies, in addition to our own set of tuned parameters. We randomly selected 6 SPEC benchmarks for our evaluation – *bzip, gcc, gzip, parser, vortex* and *vpr*. We expect the results with other SPEC benchmarks to be similar and our algorithms to apply to more advanced applications.

We report energy consumption and performance degradation results for the new PS and PD algorithms. For comparison, we also implement the original static and dynamic algorithms studied in [23]. We call the original static algorithms OSs (Static Standby), OSn (Static Nap) and OSp (Static Powerdown). For the original dynamic algorithms, we use four different settings for the required set of thresholds. The first set (ODs) was suggested by [23] to give the best $E \cdot D$ results for their simulation experiments with SPEC benchmarks. The second set of threshold values (ODn), also from [23], is the best setting for their Windows NT benchmarks. The third set (ODc) is calculated based on $E \cdot D$ competitive analysis shown in [23]. The fourth set (ODt) is obtained by extensive hand tuning, to account for the differences in the applications and system studied here and in [23]. For tuning, we started with the above thresholds and explored the space around them to find a set of "best" thresholds for each application that minimized energy within 10% performance degradation. We run OD with each of these thresholds and refer to these algorithms as ODs, ODn, ODc, and ODt respectively (where the subscripts stand for SPEC, NT, competitive-analysis, and tuned respectively). Table 1 gives the values of the various thresholds used.

Scheme		Thresholds (ns)
ODs		(0, 2000, 50000)
ODn		(0, 100, 5000)
ODc		(27, 103, 9131)
ODt	bzip	(0, 1000, 50000)
	gcc	(25, 500, 50000)
	gzip	(25, 150, 17830)
	parser	(13, 2000, 75000)
	vortex	(0, 1000, 50000)
	vpr	(13, 250, 50000)

Table 1: Thresholds used for different configurations of OD (The first number in a tuple (Th_0, Th_1, Th_2) is the threshold from active to standby, the second number is from standby to nap, and the third number is from nap to power down).

In addition, we enhance the original dynamic algorithms and the original static algorithms to provide performance guarantees using the method described in Section 3. We call the performance guaranteed static algorithms OSs+, OSn+, and OSp+ and the dynamic ones ODs+, ODn+, ODc+, and ODt+.

For all the performance-guaranteed algorithms, we vary the $Slowdown_{limit}$ parameter from 5% to 30%.

In our experiments for PS and PD, we set the epoch length to

Processor	
Clock frequency	2 GHz
Issue queue	128 entries
Fetch, issue, commit width	8 instructions
Branch prediction	2 level
Branch misprediction penalty	6 cycles
Int ALU & mult/div	8 & 2
FP ALU & mult/div	4 & 2
Cache memory	
L1 D-cache, I-cache	32KB 2-way 32-byte lines, 2 cycles
L2 unified cache	512KB 4-way 64-byte lines, 8 cycles

Table 2: Processor and cache configuration.

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

Table 3: Power consumption and transition time for different power modes.

1 million instructions for all applications. Epoch length may have some effect on the final energy saved – too short an epoch length could cause access counts to vary a lot from epoch to epoch, making the predictions in PS and PD less accurate. To test the sensitivity of our energy results to epoch length, we varied the epoch length from 0.2 million to 10 million instructions for PS and PD. We found a difference from our reported results of less than 6% for almost all applications and all slack values. In addition, when the slack is larger than 20%, this variation is even smaller.

5.2 Results for Performance Guarantee

5.2.1 Original Algorithms

Table 4 shows the performance degradation for the original static (OSs, OSn, OSp) and dynamic algorithms with the three different settings for thresholds (ODs, ODn and ODc). We do not show the results for ODt because this is tuned with 10% slowdown as the limit.

Scheme	OSs	OSn	OSp	ODs	ODn	ODc
bzip	1	9	832	6	219	21
gcc	1	14	603	6	140	29
gzip	1	6	470	4	25	8
parser	4	33	2013	9	835	40
vortex	2	22	1633	5	466	22
vpr	2	18	1635	3	505	12

Table 4: % execution time degradation for original memory algorithms.

As expected, the performance degradation for the static algorithms increases dramatically from OSs to OSp. This is because the lower the power mode that a chip stays in, the longer it takes to transition into active to service a request, making OSp virtually unusable. For the dynamic algorithms, the performance degradation is different for different threshold settings. In general, ODs has reasonable performance degradation (3–9%). This is not surprising since ODs was hand-tuned for various SPEC benchmarks. However, as shown later, ODs saves less energy than PD. ODc has medium to high performance degradation, around 8–40%. ODn is the worst, with most applications' performance degraded over 100% and one up to 835%. The reason is that this threshold setting is tuned for Windows NT benchmarks, not SPEC2000. These results unequivocally show the strong sensitivity of the performance degradation to the thresholds. Thresholds tuned for one set of applications give unacceptable performance for another set, providing evidence for the need for painstaking, application-dependent manual tuning in the original dynamic algorithms.

$Slowdown_{limit}$	5%	10%	20%	30%
PS vs. best OS+	36 [19, 42]	18 [-45, 55]	19 [-37, 56]	-2 [-35, 27]
PD vs. best OD+	49 [6, 68]	29 [14, 40]	12 [3, 29]	15 [5, 30]
PD vs. PS	27 [10, 37]	28 [5, 40]	23 [10, 36]	22 [4, 37]
PD vs. best OD	N/A	-2.2 [-15, 13]	10 [10, 29]	8 [-9,26]
PS vs. best OS	42 [21,61]	22 [-54, 60]	11 [-37, 62]	3 [-35, 52]

Table 5: Relative comparison of energy consumption of different algorithms. The numbers are *average* [*min*, *max*]% improvement in energy consumption of the first algorithm over the second. Best OS+, OD+, OS, and OD imply cases with the lowest energy. For OS and OD, only the cases that are within the specified slowdown are considered.

5.2.2 Performance Guaranteed Algorithms

Table 6 shows the performance degradation for the 8 algorithms that use the performance guarantee method described in Section 3. $Slowdown_{limit}$ ranges from 5% to 30%. Across all the 192 cases (covering all the algorithms, applications, slowdown limits, and threshold settings), the performance degradation stays within the specified limit. This indicates that our method for guaranteeing performance is indeed effective, even when combined with algorithms such as OD+ and OS+ that are not designed to be performance-aware.

5.3 Results for Energy Savings

Figure 4 shows the energy consumption for the various control algorithms. For OS+ and OD+, we show the results of the setting with the minimum energy consumption (for each application). On the right side of each figure, we also show the results for OS and OD for reference only – as discussed, their tuning requirements likely make them impractical to implement in a real system. Since these algorithms do not provide a performance guarantee, their performance degradations are shown on top of the energy bars. Each bar is also split into energy consumed in different power modes. Table 5 provides a summary of the above data by showing the average, minimum, and maximum relative improvements of energy savings for key pairs of algorithms for each $Slowdown_{limit}$ value.

Overall results: Comparing all algorithms that provide a performance guarantee, we find that PD consumes the least energy in all cases. PS does better than OS+ in most (but not all) cases, but is never able to beat PD. PD and PS also compare favorably to the original algorithms without performance guarantee in many (but not all) cases. We next discuss the results in some more detail, comparing key pairs of algorithms.

PS vs. best OS+: PS consumes less or similar energy as the best OS+ in most cases, particularly with smaller slack. This is because PS allows different configurations for different chips and changes

$Slowdown_{limit}$	5%								10%							
Scheme	ODs+	ODn+	ODc+	OSs+	OSn+	OSp+	PS	PD	ODs+	ODn+	ODc+	OSs+	OSn+	OSp+	PS	PD
bzip	3	4	4	1	5	5	4	4	6	8	7	1	9	10	8	8
gcc	3	4	3	1	4	4	3	3	6	7	6	2	8	9	7	6
gzip	3	3	3	1	4	5	3	3	4	7	6	1	6	9	6	6
parser	4	4	3	4	5	5	4	4	8	8	7	4	9	10	8	8
vortex	3	4	3	2	5	5	2	3	5	8	6	2	9	10	6	7
vpr	3	4	4	2	4	5	3	3	3	8	7	2	9	9	6	7
$Slowdown_{limit}$	20%								30%							
Scheme	ODs+	ODn+	ODc+	OSs+	OSn+	OSp+	PS	PD	ODs+	ODn+	ODc+	OSs+	OSn+	OSp+	PS	PD
bzip	6	17	14	1	9	19	16	16	6	25	20	1	9	29	24	24
gcc	6	14	11	1	14	17	15	12	6	21	17	1	14	26	23	19
gzip	4	13	8	1	6	18	11	12	4	19	8	1	6	26	18	19
parser	9	16	13	4	19	19	17	17	9	24	20	4	28	29	27	24
vortex	5	16	12	2	18	19	17	15	5	23	18	2	22	28	26	21
vpr	3	16	12	2	17	18	16	15	3	24	12	2	18	27	26	24

Table 6: Percentage execution time degradation for performance-guaranteed memory algorithms.

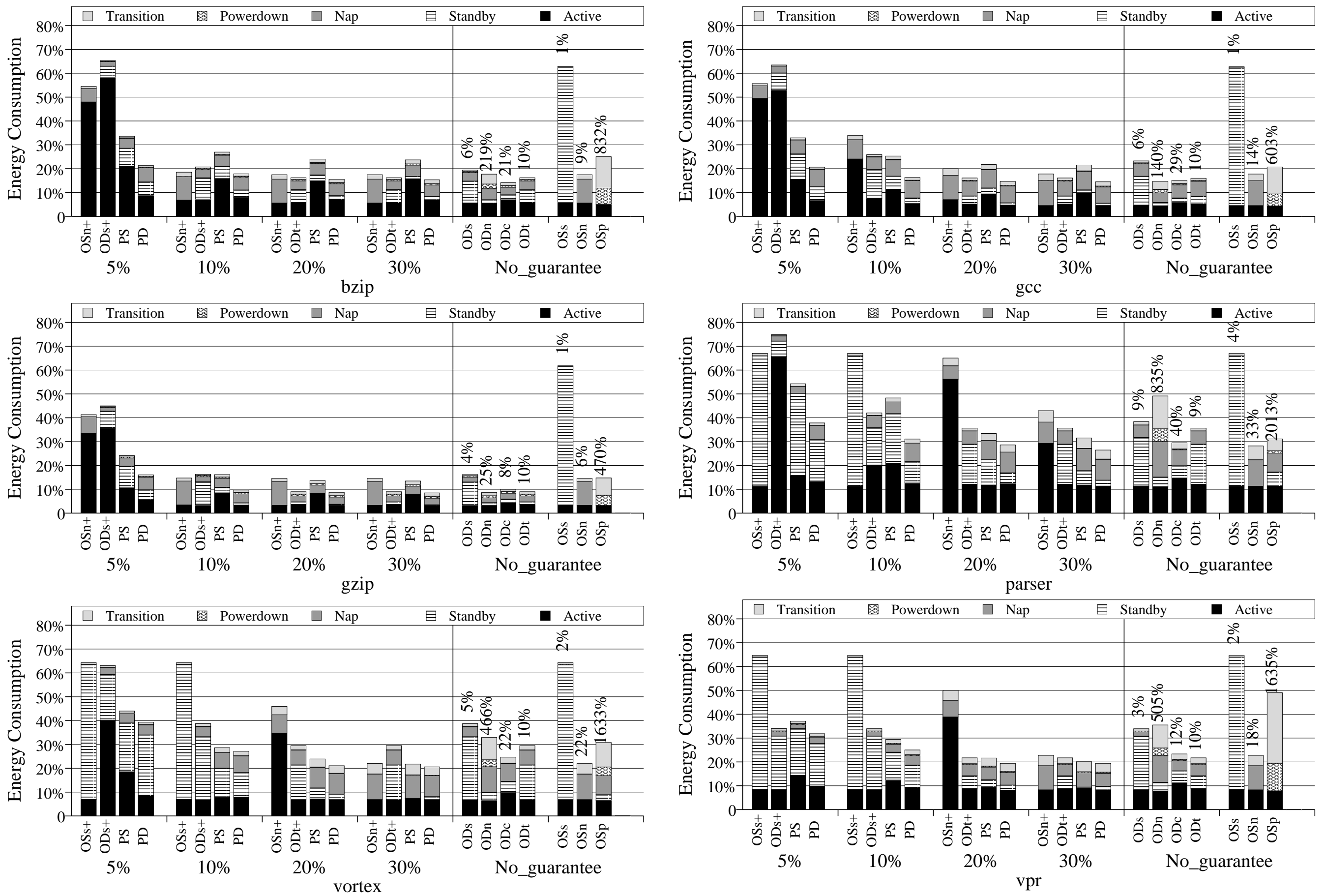

Figure 4: Memory energy consumption for different $Slowdown_{limit}$, normalized to the case without energy management. For OD and OS, the numbers above the bars represent the % performance degradation.

these at epoch granularity, taking into account changes in the spatial and temporal access pattern and available slack. In contrast, OS+ simply uses the same configuration for all chips throughout the entire execution and never considers the available slack. Especially when the available slack is small, OS+ uses up the slack quickly and has to force all chips to active in order to provide the performance guarantee.

There are a few cases where PS consumes 36-46% more energy than the best OS+ (e.g., *bzip* with 10-30% $Slowdown_{limit}$). Note, however, that this comparison is with the best OS+, and determining the best OS+ also requires tuning. These cases occur when PS's prediction is inaccurate, potentially resulting its use of the wrong configuration for the next epoch. The applications in these cases (e.g., bzip) have irregular access traffic that varies substan-

tially from epoch to epoch. We expect future improvement on PS by dynamically changing the epoch length; e.g., using the recent phase-tracking work [9, 34].

PD vs. best OD+: PD always consumes less energy than the best OD+, even though PD does not require any manual threshold tuning and the best OD+ includes a threshold setting manually tuned for that application. In some cases, the energy reduction is quite significant (up to 68%). The reason is that PD is able to change its threshold settings each epoch to respond to varying access count and available slack. OD+, however, uses a fixed setting throughout the run. The results clearly indicate the limitation of using a single threshold setting even for a single application, especially at low allowed slowdown.

PS vs. PD: PD always consumes less energy than PS, saving up to 40% in one case. The reason is that within an epoch, PD can also exploit temporal variability while PS only exploits spatial variability. Once PS sets a chip into a certain power mode, even if the chip is idle for a long time, PS does not put the chip into lower power modes. PD, however, will take advantage of this gap and move to a lower power mode. The difference between PD and PS is more pronounced for applications with unbalanced traffic in time but relatively uniform traffic across chips (e.g., *bzip* where PS consumes 33-37% more energy than PD).

PS vs. best OS and PD vs. best OD: Just for reference, for given $Slowdown_{limit}$, we compare PS (PD) with the lowest energy OS (OD) that incurs slowdown within the specified limit. This is an unfair comparison since OS/OD require extensive tuning, including per-application tuning, and do not provide a performance guarantee. Even so, in most cases, PD compares favorably to OD and in many cases, PS compares favorably to OS (Table 5).

The reason that PS does not do better in some cases is that the performance guarantee is too conservative: whenever the slack is used up, all chips are forced active until the next epoch. It does not allow "borrowing" slack from future epochs that may not need this much slack. Thus, examining the actual slowdown by PS, it is significantly lower than the given slack. For example, with 10% slack, PS slows down vpr only by 6%.

6. DISK ENERGY MANAGEMENT

6.1 Disk Power Model

To reduce energy consumption, modern disks use multiple power modes including active, standby, powerdown, and other intermediate modes [35]. In the active mode, a disk is spinning at its full speed even when there is no disk request, and therefore it provides the best-possible access time but consumes the most energy. In the active mode, serving a request requires extra energy to move the disk head in addition to the energy consumed by disk-spinning. In the standby mode, the disk consumes much less energy, but servicing a request incurs significant energy and time overhead to spin up to active mode.

Recently, Gurumurthi et al. [14] have proposed using multi-speed disks, called DRPM, to reduce energy for data center workloads. Lower rotational speed modes consume less energy than higher ones, and the energy and time costs to shift between rotational speed modes are relatively small compared to the costs for shifting from standby to active in the traditional disk power models. Furthermore, a DRPM disk can service requests at a low rotational speed without the need to transition to full speed. Although the service time is longer for all accesses at slower speeds, it can avoid the transition overhead. We use the DRPM disk model in our study since, for data center workloads, it saves much more energy than the traditional model.

6.2 Algorithms

The performance-guarantee method for disks is similar to the memory case described in Section 3. The main difference is in estimating delay due to the energy management algorithm. There are two sources for performance delay that we incorporate: (1) time spent transitioning between two speeds, since no request can be serviced during this time, and (2) higher rotational delay when a request is serviced at a low speed.

For energy control, we study algorithms analogous to the memory case. Although static algorithms for disks have not been previously evaluated, we can define OS and OS+ algorithms analogous to the memory case: all disks stay at a fixed speed level to service requests. For performance guarantee in OS+, all disks are forced to full speed when the actual percentage slowdown exceeds the specified limit.

The dynamic algorithms, OD and OD+, are based on the heuristic DRPM algorithm proposed in [14]. This algorithm dynamically transitions a disk from one speed to another based on changes in the average response time and the request queue length. It requires tuning of five parameters: (1) checking period p to examine the disk queue length, (2) the upper tolerance UT in percentage response time changes to spin up a disk to a higher RPM, (3) the lower tolerance LT in percentage response time changes to spin down a disk, (4) window size W, and (5) the disk queue length threshold N_{min}. Our experiments indicate that the last four parameters have significant impact on performance and energy.

Compared to memory, OD and OD+ for disks has many more parameters, each with a different meaning. Dynamically tuning all of them for PD is difficult; specifically, it is difficult to model or analyze the impact of the period length, the window length, and the disk queue length parameters on performance slowdown and energy. We therefore chose to restrict dynamic tuning to the two threshold parameters of LT and UT, using a method similar to that for memory (Section 4.2). A larger LT or UT saves more energy but incurs a higher slowdown (larger LT implies more aggressive transitions to lower speeds while a larger UT implies lazier transitions to higher speeds). Therefore, similar to memory, PD dynamically adjusts LT and UT based on the predicted access count and available slack in the next epoch, to reduce energy consumption while only using the available slack. For the rest of the parameters, we started from the values in [14] and explored the space around these values to find a best set, which is then used throughout all experiments (discussed further in Section 6.3).

6.3 Experimental Setup

We evaluated our disk energy control algorithms using three traces, with the widely used DiskSim trace-driven simulator [12] modified to support the DRPM disk model. The disk modeled is similar to the IBM Ultrastar 36Z15, but enhanced with the linear multiple power model [14]. The parameters are taken from the disk's data sheet [19] and [5, 14]. Table 7 shows some key parameters.

We use both synthetic traces and real system traces in our simulations. Similar to [14], we consider two types of distributions for inter-arrival times for our synthetic traces, exponential and Pareto. The real system trace is the Cello trace collected from HP Cello File Servers in 1996. In our experiments, the trace includes 10 hours of execution during busy daytime (1996.11.1.8am-6pm). The original trace contains accesses to 20 disks. But many disks are idle most of the time. To prevent these disks from polluting the results, we filter the trace to only contain accesses to the 5 busiest disks. In addition, since this trace is quite old, we replay the trace 10 times faster based on the fact that current processors are about 10 times

IBM Ultrastar 36Z15 with DRPM	
Standard Interface	SCSI
Individual Disk Capacity	18.4 GB
Maximum Disk Rotation Speed	15000 RPM
Minimum Disk Rotation Speed	3000 RPM
RPM Step-Size	3000 RPM
Active Power(R/W)	13.5 W
Seek Power	13.5 W
Idle Power@15000RPM	10.2 W
Standby Power	2.5 W
Spinup Time (Standby → Active)	10.9 secs
Spinup Energy (Standby → Active)	135 J
Spindown Time (Active → Standby)	1.5 secs
Spindown Energy (Active → Standby)	13 J

Table 7: Disk model parameters. Standby and active power are used to feed the linear power model to derive power at different speeds.

Trace	#requests	#disks	average inter-arrival time (ms)
Exponential	1000000	12	10.0
Pareto	1000000	12	10.0
Cello'96	536937	5	8.9

Table 8: Trace parameters

faster than processors in 1996. The parameters of these traces are shown in table 8.

To accurately estimate the performance delay due to energy management, we need application-level inter-request dependence information. For instance, an application may need to wait for results from previous reads for subsequent computation. After certain computation time, it may send the next few independent requests. Unfortunately, the traces do not provide us with such information. To simulate dependence effects, we randomly assign dependency for every request in the two synthetic traces: each request is dependent on one of the most recent n requests with probability $prob$. We set $n = 10$ and $prob = 0.95$ for the synthetic traces. The Cello trace contains some information about the process ID. We assume that each process uses synchronous mode to read data (which is true in most file system workloads [33]), so a request depends on the previous read request issued by the same process.

In the results reported here, the epoch length is 100 seconds. Extensive experiments indicated that our algorithms are not very sensitive to the epoch length. We used the following procedure for the remaining parameters for the OD, OD+, and PD algorithms. We started with the parameters used in [14] and explored the space near those parameters to minimize the overall energy-delay product for OD. Specifically, we varied the period length from 2 seconds (which was used in [14]) to 12 seconds and explored window sizes of 250, 500 and 1000 (also explored in [14]). The best setting we found was: ($p = 6, LT = 5\%, UT = 50\%, W = 250, N_{min} = 0$). We refer to OD and OD+ with these settings as OD1 and OD1+ respectively. We also ran OD and OD+ at the parameter settings chosen by [14]. We refer to this setting as OD2 and OD2+ respectively, and the parameters are ($p = 2, LT = 5\%, UT = 15\%, W = 250, N_{min} = 0$).

For PD, we used the parameter settings of OD1, except that as mentioned before, LT and UT are dynamically adjusted.

For space reasons, we do not report results for the OS algorithms (these have not been evaluated in previous work either). We do report results for the performance-guaranteed version, OS+. We denote the variations as OSr+ representing a fixed rK RPM speed, where $r = 3, 6, 9, 12$.

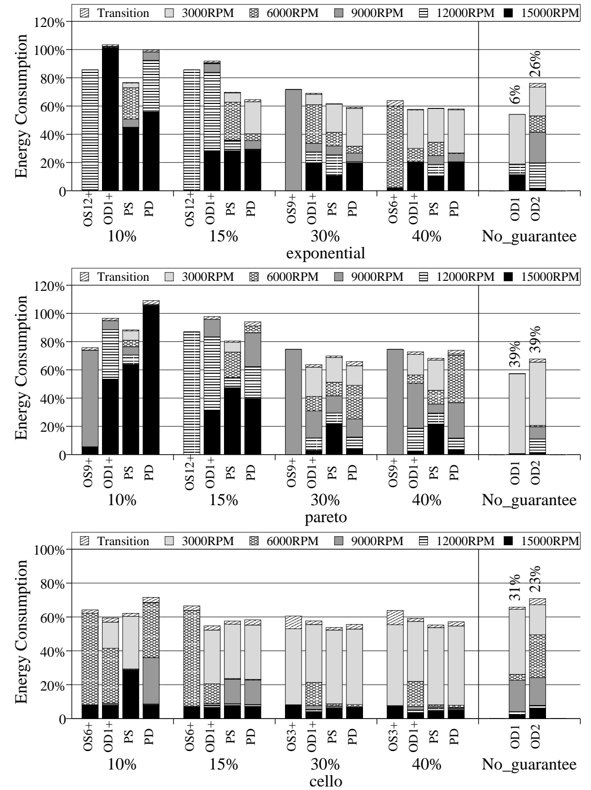

Figure 5: Disk energy consumption for different $Slowdown_{limit}$, normalized to the case without energy management. For OD, the numbers above the bars represent the % performance degradation.

Scheme	OD1	OD2
Exponential	6	26
Pareto	39	39
Cello	31	23

Table 9: Percentage execution time degradation for OD algorithms for disk

$Slowdown_{limit}$	10%	15%	30%	40%
PS vs. best OS+	-6 [-17, 3]	7 [-5, 19]	11 [6, 14]	10 [9, 13]
PD vs. best OD+	-10 [-20, 3]	9 [-6, 30]	5 [-3, 14]	0 [-2, 3]
PS vs. PD	18 [13, 23]	3 [-8, 14]	-2 [-6, 3]	3 [-1, 8]

Table 10: Relative comparison of energy consumption of different algorithms. The numbers are *average*, [*min*, *max*] *percentage* improvement in energy consumption of the first algorithm over the second.

6.4 Results

The results for disks are presented in Tables 9, 10 and Figure 5, analogous to the results for memory. Again, we see that the original dynamic algorithms can incur unacceptably large application- and threshold-dependent performance degradations (up to 39%) (Table 9). In contrast, our performance guarantee algorithm is effective in *all* cases and never violates $Slowdown_{limit}$. For space reasons, we do not show the table of execution time degradation for the performance-guaranteed algorithms.

For energy results, we only briefly discuss high-level results, with a focus on differences from the memory case. First, in sharp

contrast to the memory case, PS is either comparable to or better than PD. The primary reason is the complexity of the dynamic algorithms in the disk case in terms of the number of parameters and the tuning required for them. PD can dynamically tune only two parameters while keeping the others fixed, so PD does not achieve its full potential. This is also the reason why PD does worse than OD+ in some cases – PD cannot compete with the hand-tuning of OD+ for some cases; however, it is important to note that this hand tuning makes OD+ far less practical.

Second, no algorithm is a clear winner across all cases although PS is the best or close to the best in all but one case. The exception case is for the Pareto trace with 10% $Slowdown_{limit}$, where PS consumes 17% more energy than OS9+. The reason is that this case has a particularly bursty distribution which results in poor predictions in the PS algorithm; the low $Slowdown_{limit}$ exacerbates the effect of this poor prediction. In the future, we plan to apply the phase-tracking work by others to enable better predictions [9, 34].

The primary reason for the somewhat inconclusive results in the disk case is that PD is not able to exploit its full potential as discussed above. We tried another optimization on PD as follows. Currently, PD uses the same parameter settings for all disks in the system. We used a PS-style knapsack optimization algorithm to determine a close-to-optimal apportioning of the total available slack among the individual disks (based on per-disk access counts). Customizing the slack for each disk allowed customizing the threshold values for each disk. We applied this hybrid optimization-heuristic based algorithm to the Cello trace for 10% $Slowdown_{limit}$ and found it to improve the energy savings of PS and PD by 9% and 21% respectively. (We also tried this technique on memory, but found no improvement since PD already does well with memory.)

7. RELATED WORK

This section discusses closely related work on control algorithms for energy management for memory, disk, and other subsystems.

Memory: In addition to the work described in Section 2, Delaluz et al. have also studied compiler-directed in [8] and operating-system-based approaches in [7] to reduce memory energy. Recently, Padmanabhan et al. proposed a power-aware virtual memory implementation in the OS to reduce memory energy [29]. Our work differs from all previous work in that it focuses on performance-guaranteed control algorithms.

Disk: Most of the previous disk energy work focuses on a single disk in mobile devices [13, 15, 31, 36, 37, 39, 40]. Recently, a few studies looked into energy management for high-end storage systems [5, 6, 14]. An analytical technique is to use 2-competitive benefit analysis to compute the threshold values [25]. Several previous studies have investigated some adaptive threshold adjustment schemes [10, 15, 22]. However, they focus on energy consumption without any explicit limits on the consequent performance degradation. Our PD algorithm can provide performance guarantees.

Other control algorithms for energy adaptation. There is also substantial work on control algorithms for adapting other parts of the system, in particular, the processor and cache [2, 4, 11]. Integrating this work with the storage system adaptations is a key part of our future work. Most work on the processor architecture has been similar to the dynamic algorithms studied here (i.e., threshold based) and requires a lot of tuning. Some exceptions are work by Huang et al. [16] and work in the area of multimedia applications [18] where adaptations occur at the granularity of sub-routines and multimedia application frames respectively. This granularity is analogous to our epochs, but none of this work provides a performance guarantee. Recently, there has been work on using optimization-based techniques for adapting the processor architecture for multimedia applications with the explicit purpose of eliminating tuning of thresholds for processor algorithms [17]. Thus, this work shares our goals and the optimization equations are similar to those for our PS algorithm. However, there are several significant differences. First, because it is difficult to estimate slowdowns due processor adaptations, the work in [17] relies on extensive off-line profiling that exploits certain special features of multimedia applications. Instead, we are able to make more elegant analytic estimates of the slowdowns due to adaptation in each storage device, and apply our work to general-purpose applications. Furthermore, we are also able to provide performance-guarantees, which the previous work does not provide. Finally, there has also been optimization-driven work in the area of dynamic voltage scaling in the processor [21]. The PS optimization framework shares similarities with such work, but applies the ideas to an entirely different domain (storage subsystems).

8. CONCLUSIONS AND FUTURE WORK

Current memory and disk energy management algorithms are difficult to use in practice because they require painstaking application-dependent manual tuning and can result in unpredictable slowdowns (more than 800% in one case). This paper overcomes these limitations by (1) proposing an algorithm to guarantee performance that can be coupled with any underlying energy management control algorithm, and (2) proposing a self-tuning, heuristics-based energy-management algorithm (PD) and an optimization-based (tuning-free) energy-management algorithm (PS). Over a large number of scenarios, our results show that our algorithms are effective in overcoming the current limitations, thereby providing perhaps the first practical means of using the low power modes present in commercial systems today and/or proposed in recent literature.

We envisage several directions for future work. First, we would like to work towards energy management algorithms that take all system components (e.g., processors, memory, and disk) into account. Second, our work will likely benefit from incorporating recent work on detecting predictable phases [9, 34], to improve the predictions used by our algorithms. Third, we would like to improve on the PD algorithm for disks by combining with an optimization based approach as mentioned earlier. Finally, we would like to combine energy management with thermal considerations.

9. REFERENCES

[1] Power, heat, and sledgehammer. White paper, Maximum Institution Inc., 2002.

[2] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual Symposium on Computer Architecture*, 2001.

[3] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, Univ. of Wisconsin-Madison, 1996.

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

[5] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th International Conference on Supercomputing*, June 2003.

[6] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Nov 2002.

[7] V. Delaluz, M. Kandemir, and I. Kolcu. Automatic data migration for reducing energy consumption in multi-bank

memory systems. In *the 39th Design Automation Conference*, June 2002.

[8] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramniam, and M. J. Irwin. Hardware and software techniques for controlling DRAM power modes. *IEEE Transactions on Computers*, 2001.

[9] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *36th Annual International Symposium on Microarchitecture*, 2003.

[10] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. 2nd USENIX Symposium on Mobile and Location-Independent Computing*, 1995.

[11] D. Folegnani and A. González. Energy-efficient issue logic. In *Proceedings of the 28th Annual Symposium on Computer Architecture*, 2001.

[12] G. R. Ganger, B. L. Worthington, and Y. N. Patt. The DiskSim simulation environment - version 2.0 reference manual.

[13] P. Greenawalt. Modeling power management for hard disks. In *the Conference on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Jan 1994.

[14] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proceedings of the International Symposium on Computer Architecture*, pages 169–179, June 2003.

[15] D. P. Helmbold, D. D. E. Long, T. L. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications*, 5(4):285–297, 2000.

[16] M. C. Huang, J. Renau, and J. Torrellas. Positional processor adaptation: Application to energy reduction. In *Proc. of the 30th Annual Intl. Symp. on Comp. Architecture*, 2003.

[17] C. J. Hughes and S. V. Adve. Spreading slack for optimal energy-performance tradeoffs for multimedia applications. In *Proceedings of the International Symposium on Computer Architecture*, June 2004.

[18] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th International Symposium on Microarchitecture*, Dec 2001.

[19] IBM hard disk drive - Ultrastar 36Z15.

[20] S. Irani, S. Shukla, and R. Gupta. Competitive analysis of dynamic power management strategies for systems with multiple power saving states. Technical report, UCI-ICS, September 2001.

[21] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, 1998.

[22] P. Krishnan, P. M. Long, and J. S. Vitter. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. In *12th International Conference on Machine Learning*, 1995.

[23] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–116, 2000.

[24] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, December 2003.

[25] K. Li, R. Kumpf, P. Horton, and T. E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *USENIX Winter*, pages 279–291, 1994.

[26] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[27] Martello and Toth. Knapsack problems: Algorithms and computer implementation. In *John Wiley and Sons Ltd*, 1990.

[28] F. Moore. More power needed. Energy User News, Nov 25th, 2002.

[29] H. H. Padmanabhan. Design and implementation of power-aware virtual memory. In *USENIX*, 2003.

[30] G. A. Paleologo, L. Benini, A. Bogliolo, and G. De Micheli. Policy optimization for dynamic power management. In *Proceedings of the 35th Annual Conference on Design Automation*, pages 182–187, 1998.

[31] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *the 18th International Conference on Supercomputing*, June 2004.

[32] Rambus. Rdram. http://www.rambus.com, 1999.

[33] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Proceedings of the Winter 1993 USENIX Conference*, 1993.

[34] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[35] Storage Systems Division. Adaptive power management for mobile hard drives. IBM White Paper, 1999.

[36] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[37] J. Zedlewski, S. Sobti, and N. G. et al. Modeling hard-disk power consumption. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*, 2002.

[38] L. Zhang, Z. Fang, M. Parker, B. Mathew, L. Schaelicke, J. Carter, W. Hsieh, and S. McKee. The impulse memory controller. *IEEE Transactions on Computers*, pages 1117–1132, 2001.

[39] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao. Reducing energy consumption of disk storage using power-aware cache management. In *10th International Symposium on High Performance Computer Architecture*, 2004.

[40] Q. Zhu, A. Shankar, and Y. Zhou. Power aware storage cache replacement algorithms. In *the 18th International Conference on Supercomputing*, June 2004.