Performance-Directed Energy Management for Storage Systems

MANY ALGORITHMS USE THRESHOLDS TO CONTROL A MEMORY OR DISK'S POWER MODE, WHICH CAN DEGRADE PERFORMANCE PAST AN ACCEPTABLE LIMIT AND REQUIRE PAINSTAKING PARAMETER TUNING. THESE DRAWBACKS MAKE THEM IMPRACTICAL FOR MANY REAL SYSTEMS. PERFORMANCE-DIRECTED ALGORITHMS OVERCOME BOTH THESE OBSTACLES.

Xiaodong Li
Zhenmin Li
Pin Zhou
Yuanyuan Zhou
Sarita V. Adve
University of Illinois at Urbana-Champaign

Sanjeev Kumar
Intel Laboratories

Energy consumption has become an important issue in the design of battery-operated mobile devices and sophisticated data centers.[1] The storage hierarchy, which includes memory and disks, is a major energy consumer in such systems, especially for high-end servers at data centers.[2-4] Indeed, recent measurements from IBM server systems show that memory can consume 50 percent more power than processors,[5] and a recent industry report shows that storage devices account for nearly 27 percent of a data center's total energy consumption.[6]

Low power modes are one way to reduce energy consumption. Modern memory such as RDRAM (www.rambus.com) lets an individual memory device transition into a range of low-power operating modes. Similarly, many disks also support multiple power modes.[3]

But while transitioning a memory chip or a disk to a low power mode *can* save energy, it can also degrade performance. For these low power modes to be as effective as possible, some control algorithm must decide which power mode each device should be in at a particular time. Much work has focused on energy control algorithms for storage systems that transition a device into a low power mode when a certain usage function exceeds a specified threshold.[2-4] These algorithms are difficult to use in real systems, however, because designers must painstakingly and manually tune threshold values, and even then a performance guarantee is difficult. The sidebar "Why Current Algorithms Don't Always Work" describes the issues in more depth.

To address these limitations, we developed three algorithms:

- a performance guarantee technique that designers can use with any underlying energy-control algorithm,
- a performance-directed control algorithm that periodically assigns a static configuration to different devices by solving an optimization problem, and
- another performance-directed control algorithm that dynamically self-tunes according to an optimal set of thresholds.

Together, these algorithms represent a significant step toward practical control

Published by the IEEE Computer Society

algorithms for memory and disk energy conservation, especially for systems that require service guarantees. Our algorithms eliminate painstaking, application-dependent parameter tuning, and users need not worry about whether the underlying energy conservation scheme will degrade performance by some unpredictable, unacceptable value.

To evaluate our algorithms' effectiveness, we compared them to various control algorithms with and without performance guarantee for memory and disk, covering more than 200 scenarios. (Because we do not have the space to describe our algorithms in the context of both memory and disk, we explain them for memory only, although we present simulation results for both memory and disks.) In every case, the performance guarantee technique successfully held performance degradation to the specified limit. Further, in most cases, our two new performance-directed control algorithms saved more energy than existing algorithms.

Why current algorithms don't always work

Current control algorithms have two main limitations. The first is the need for manual threshold tuning, which can be tedious and inaccurate. We found that reasonable threshold values depend heavily on the system and application. For the memory subsystem, a set of thresholds derived from competitive analysis[1] showed a performance degradation of 8 to 40 percent when applied to six SPEC benchmarks. Our hand-tuning efforts also showed that the best threshold values for a given application could cause high performance degradation in others. For memory, the best threshold values for a 10 percent performance degradation for the SPEC benchmark gzip gave 63 percent performance degradation when applied to the parser benchmark. Even for the same application, the best threshold values can change for different program phases and can depend on input data.

The second limitation is the lack of performance guarantee. Even if a system design included thresholds tuned for an expected set of applications, no mechanism bounds performance degradation for applications that deviate from the expected set. Our experiments show that the potential performance degradation arising from an inappropriate set of thresholds can be as high as several hundred percent. This type of unpredictable behavior or lack of a safety net is clearly a problem for all users, but it can be catastrophic for high-end servers in host data centers that must honor service-level contracts. Such data centers are becoming increasingly important consumers of high-end servers, and the ability to provide some form of performance guarantee is crucial to their business model. Further, these high-end servers are likely to reap the most cost savings from reduced memory and disk energy.

References

1. A.R. Lebeck et al., "Power Aware Page Allocation," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 00), ACM Press, 2000, pp. 105-116.

Providing a performance guarantee

We incorporated a new technique that guarantees the underlying control algorithm will not degrade performance beyond the specified limit. This technique dynamically monitors the performance degradation at runtime and forces all devices to full-power mode when the degradation exceeds the specified limit. It is possible to combine this performance-guarantee technique with any underlying control algorithm for managing memory or disk power modes, including our self-tuning and tuning-free algorithms. With this technique, the user specifies an acceptable slowdown, and the system seeks to minimize energy within that constraint. Users can thus make conscious, specific trade-offs of performance and energy savings. Because the technique works with any energy-control algorithm, both existing and new algorithms can limit performance degradation, making energy-saving schemes usable for workloads that demand a certain performance level.

The acceptable slowdown is based on the assumption that the user has been guaranteed some base "best" performance assuming no energy management, and can opt to save more energy by accepting a slowdown relative to this base performance. This acceptable slowdown, $\text{Slowdown}_{\text{limit}}$, is the percentage of increase in execution time relative to the base. The performance guarantee technique uses this as input and then ensures that the underlying energy management algorithm does not slow down execution beyond this acceptable limit.

The idea then is to track the slowdown and force all devices to active when the observed slowdown exceeds $\text{Slowdown}_{\text{limit}}$. The performance guarantee algorithm thus has two key tasks: estimate the actual slowdown directly attributable to energy management and ensure that the actual slowdown does not exceed the specified limit.

Estimating actual slowdown

At each memory access, the memory controller estimates the absolute delay in execution time attributable to energy management. The delay estimation should be as accurate as possible, but conservative—the estimated delay must be greater than or equal to the actual delay.

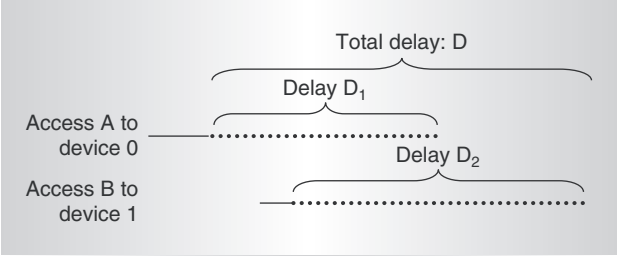

Figure 1. Example of overlapped requests and refinement of the delay estimation.

A simple way to estimate delay is to add the delay from energy management for each access, which would be the transition time from a low-power mode to an active one. This simple way is too conservative, however, because it does not consider the effect of overlapped accesses and other latency-hiding techniques in modern processors (out-of-order execution, prefetching, nonblocking caches, and write-buffers, among others). For that reason, we refined our slowdown estimation method to consider some of the major sources of inaccuracy.

First, our algorithm assumes that the processor sees the delay from energy management for a *single* load. In modern out-of-order processors, the latency of a cache miss that goes to memory cannot usually be fully overlapped with other computation. The additional delay from energy management simply adds to the existing stall time and delays the execution.

If the processor sends two loads to the memory system in parallel, their latencies overlap, therefore hiding some of the energy management delay. Our major refinement to the simple way of adding delay for each access is to exploit information about overlapped or concurrent requests. In Figure 1, accesses *A* and *B* overlap. Suppose both devices are in low-power mode and D_1 and D_2 are delays that occur to accesses *A* and *B*, respectively. The total delay in execution time is smaller than D_1 and D_2. A tighter bound is *D*, the result of subtracting the overlapped time from $D_1 + D_2$. This idea extends to multiple overlapped requests.

Further, most modern architectures can hide the extra delay for writebacks and reads attributable to store instructions unless there is memory contention.

Enforcing the specified limit

A simple way to enforce the performance guarantee is to ensure that the *slack*—the amount of allowed execution delay that would not violate the slowdown constraint—is never negative. If slack becomes 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 accumulates to reactivate the underlying control algorithm.

A simple method would be to update the delay and slack at each access, but that approach has two rather large limitations: It must rely on a new tunable parameter—"enough slack"—and it must check the actual *percentage* of the slowdown against the slack limit (at least one division and one comparison) after every access. These additional checks incur too much overhead.

Our idea is to break the execution time into epochs. An *epoch* is a relatively large time interval over which we assume the application execution is predictable; for our experiments, the epoch was a million instructions.

At the start of an epoch, the algorithm estimates the *absolute* available slack for the entire epoch. Now, after each access, the algorithm must check only the actual absolute delay at that point in the epoch against the estimated available slack for the entire epoch. If the actual delay is more than the available slack, the algorithm forces all devices to active power mode until the epoch ends.

The algorithm can estimate the available slack for the next epoch on the basis of the specified $\text{Slowdown}_{\text{limit}}$ and the execution time of the next epoch without power management. It predicts the latter as the same as for the last epoch and bases its prediction on the last epoch's measured execution time and slowdown estimation.

This estimation gives the fair share of available slack for the next epoch. To this, we add a correction to account for any underuse or overuse of available slack in the previous epoch. Thus, if the previous epochs have not used up their share of slack, the unused portion carries forward, and the next epoch can afford to use *more* than its fair share. Conversely, if the previous epoch used up too much slack because, say, the algorithm

incorrectly predicted the epoch length, the next epoch will attempt to make up that slack.

The overhead of our performance guarantee method consists of only one or two integer comparisons and fewer than four arithmetic additions or subtractions per access. The available-slack calculation requires some multiplication, but because it occurs only once each epoch, we can amortize this overhead over a large interval. It is therefore negligible.

PS algorithm

Because we were interested in finding an algorithm that was not threshold based, we explored the possibility of using an algorithm based on formal optimization. Our algorithm, which we call Performance-Directed Static (PS) because static algorithms[4] inspired it, also works on an epoch granularity. PS eliminates the thresholds-based nature of most energy-control algorithms by choosing a single power mode, or *configuration*, for each device for the entire epoch. Unlike the original static algorithm, which uses a fixed configuration for all devices throughout the entire execution, PS assigns different configurations to different devices. It also reassigns configurations when a new epoch starts.

Configuration choice

PS assigns a fixed configuration (power mode) to a device for the entire epoch, and the device transitions into active mode only to service a request. Because it changes the configurations at epoch boundaries according to the available slack, PS can adapt to epoch-scale changes in application behavior. It also allows different configurations for different devices, thereby exploiting the variability in the amount of traffic going to different devices. In essence, it has the flexibility to apportion total slack across devices.

The PS algorithm chooses the configuration for each epoch by mapping the problem to a constrained optimization problem. By applying standard optimization techniques, PS can achieve a close to optimal solution for fixed configurations through an epoch without complex heuristics.

For each device i, PS chooses configuration C_i that maximizes the total energy savings subject to the constraint of the total available slack for the next epoch. That is,

$$\text{maximize} \sum_{i=0}^{N-1} E(C_i)$$

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

where $E(C_i)$ and $D(C_i)$ are a prediction of the energy savings and the increase in execution time of keeping device i in configuration C_i in the next epoch. AvailableSlack is a prediction of the slack available for the next epoch (calculated in the same way as the performance guarantee algorithm). N is the total number of devices.

In essence, PS is simply adding the delay from each device to conservatively estimate the total delay. Unlike the performance guarantee algorithm, PS cannot account for overlapped accesses among devices, since it does not yet know which accesses will overlap in the next epoch. Nevertheless, the performance guarantee algorithm will carry any underused slack to the next epoch.

These equations represent the well-known multiple-choice knapsack problem (MCKP). Although finding the optimal solution for MCKP is NP-complete, we use a linear greedy approximation algorithm to find an acceptable solution.

For each device i, for each possible power mode, the optimization must estimate the energy saving and the delay for the next epoch. For an accurate estimation, it would need to predict the number and distribution of the accesses in the next epoch.

We assume that the number of accesses in the next epoch is the same as that in the last epoch, since the epochs are relatively long, and the number of accesses to each device changes slowly over each epoch. In Figure 2, for example, which shows the access count for the SPEC benchmark *vortex*, the access rate remains relatively stable (for the most part) for each device. Although some bursty periods occur, where the access count changes abruptly for a short time, resulting in suboptimal configurations, the performance guarantee algorithm compensates for this burstiness. If PS underpredicts the access count, and the power mode is too low, the performance guarantee algorithm will force the device to go active. Conversely, if PS overpredicts the access count and the epoch uses up

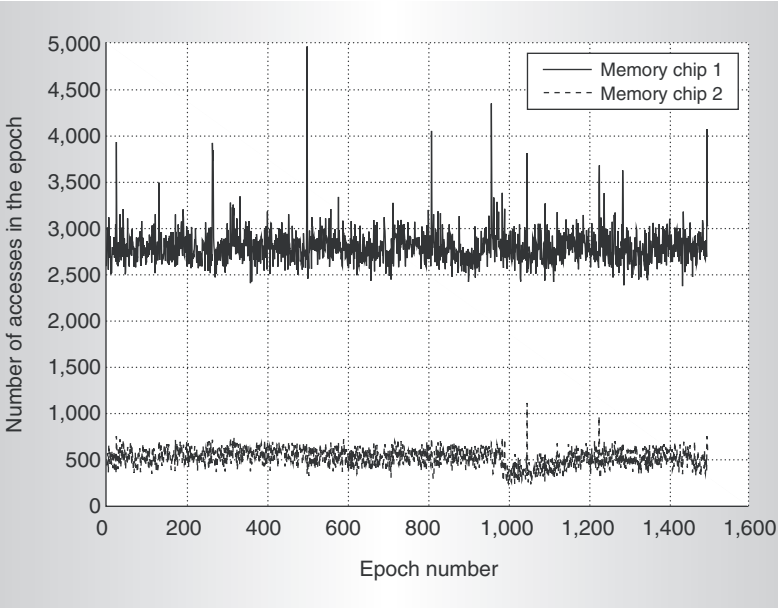

Figure 2. Access counts per epoch for two memory chips with the SPEC benchmark *vortex*. Each epoch is an interval of one million instructions.

too little slack, the performance guarantee algorithm will reclaim the leftover slack for the next epoch.

To estimate the temporal distribution of accesses, PS assumes that accesses to a given device are uniformly distributed in time and that accesses do not overlap. The assumption, although simplistic, is strictly conservative. The performance guarantee algorithm can again partly compensate for some of the suboptimal results by reclaiming any unused slack for the subsequent epoch.

PS can now calculate the contribution of device i in power mode P_k to the overall execution time delay as

$$D(P_k) = A_i \times [t_{\text{access}}(P_k) - t_{\text{access}}(P_{\text{active}})]$$

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

PS calculates the energy saved by placing device i in power mode P_k in a similar way.

Overhead

At the beginning of each epoch, PS must 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 and N is the number of chips ($M < 5$ and $N < 16$ in our system). Because we use a linear greedy algorithm to solve the MCKP, and because $D(C_i)$ and $E(C_i)$ are monotonically non-decreasing, evaluation requires $2MN$ computation steps, with each step comprising one or two integer comparisons and one or two subtractions. PS invokes such computation only at the beginning of each epoch, so we can amortize this overhead over the sizable epoch length (one million instructions).

PD algorithm

Our Performance-Directed Dynamic (PD) algorithm gained inspiration from previous dynamic algorithms that put a memory chip into lower power modes after being idle for a certain threshold period.[4] However, unlike other dynamic algorithms, PD automatically retunes its thresholds at the end of each epoch, according to available slack and workload characteristics.

Specifically, the PD algorithm maps the problem of threshold tuning to a constrained optimization problem using the same equations as PS. The difference is that PD describes configuration C_i for device i by thresholds $Th_1, Th_2, \ldots, Th_{M-1}$, where M is the number of power modes and Th_k is the time that the device will stay in power mode k–1 before going down to power mode k. The search space for PD is prohibitively large—M–1 threshold variables, and each variable could be any integer $[0, \infty]$.

Given that we have no efficient solution for this large space, we considered a heuristics-based technique that uses a linear interpolation and a simple feedback-based control mechanism.[7] The technique first curtails the space of solutions by using the same set of thresholds for all devices in a given epoch. It then exploits the first-order dependence that thresholds have on the slack available for the epoch and on the number of accesses.

For a larger slack, devices can go to lower power modes more aggressively, so thresholds can be smaller. Similarly, for given slack S, lower access counts allow for lower thresholds because they cause a smaller total delay. Thus, PD seeks 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 are predictable (as we showed earlier), we need only determine PD's Th_k as a function of S for a given access count.

Threshold heuristics

In general, if number of accesses A is fixed, $Th_k(S)$ is monotonically nonincreasing with respect to available slack S, as Figure 3 shows. To reduce the computation complexity, PD approximates $Th_k(S)$ using multiple linear segments. It first considers M key values of S and approximates Th_k for each. These values are $S = A \times t_i$ where $0 \le i \le M-1$ (we assume $t_0 = 0$). This divides the available-slack (S) axis into M distinct intervals: $[0; A \times t_1]$, $\ldots$, $[A \times t_{M-2}, A \times t_{M-1}]$, $[A \times t_{M-1}, \infty]$. PD uses the function's approximated values at the various $A \times t_i$'s to interpolate the values of the remaining points in the corresponding intervals. For $Th_k(S)$, PD determines these approximate values and interpolations in three steps:

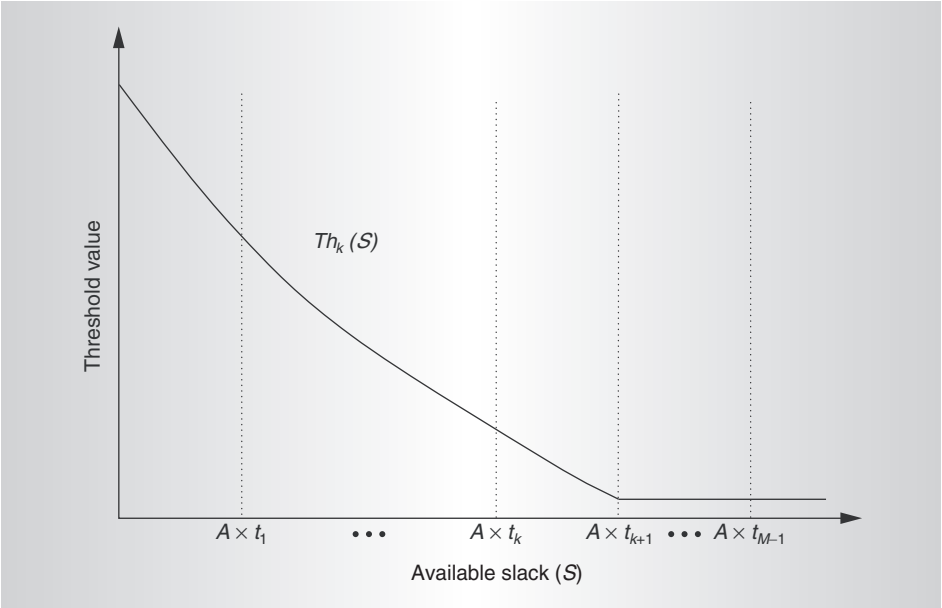

Figure 3. Threshold function $Th_k(S)$. A is the access count for the next epoch, and t_i is the transition time from power mode i to active mode.

- Consider key identified slack values $A \times t_i$, where $i > k$. These values imply available slack that is large enough to allow every access to wait for the transition time t_k. We set the minimal threshold for Th_k to be the energy breakeven point described in Irani, Shukla, and Gupta's competitive analysis[8] so that the device can aggressively go into a lower power mode. This also provides the two-competitive property—at most twice the energy as no energy management in the worst case.
- Consider the remaining identified slack values $A \times t_i$, where $0 \le i \le k$. For these cases, the available slack is either not enough or just barely enough for each access to wait for t_k. Therefore, we need to be conservative about putting a device in mode k—unless the device has already been idle for a long time, it should not be in mode k. PD thus sets threshold $Th_k(A \times t_i)$ to be much larger than for $Th_k(A \times t_{k+1})$; namely to $(C^{k-i}) \times t_k$, where C is a dynamically adjusted factor.
- After determining the M key points, for an available slack value S in any interval between two key points, determine value $Th_k(S)$ by linearly interpolating between the interval's endpoints. For available slack values S in the interval $(A \times t_{M-1}, \infty)$, the value of $Th_k(S)$ is the same as that at $S = A \times t_{M-1}$.

PD uses feedback-based control to dynamically adjust the constant value C at runtime. If during the last epoch, the system does not use up all its slack, the thresholds are too conservative. PD then reduces C to 95 percent of its current value to reduce the threshold values. The result is that, in the next epoch, the chip will go to lower power mode more aggressively. If during the last epoch, the system used up all its slack and the performance guarantee algorithm had to force all devices to become active to enforce the acceptable slowdown, the thresholds are too aggressive. In that case, PD doubles the constant to increase the threshold values.

Our experimental results show that this dynamic threshold-adjustment scheme works very well, and we never need to tune the adjustment speeds in decreasing and increasing C.

Table 1. Execution time degradation from energy management for original memory algorithms.

SPEC benchmark	Execution time degradation (percentage) for static algorithms			Execution time degradation (percentage) for dynamic algorithms		
	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	2,013	9	835	40
vortex	2	22	1,633	5	466	22
vpr	2	18	1,635	3	505	12

Overhead

At the beginning of each epoch, for each threshold Th_k, PD must first compare the current slack with the k key points to see which segment of $Th_k(S)$ it should use. This involves fewer than $M-1$ comparisons for each $Th_k(S)$ function, making the total comparisons fewer than M^2. PD then evaluates the linear functions, which takes four to five multiplications, divisions, and additions. The total computational complexity is thus smaller than M^2+5M (M being number of power modes, which is less than 5).

PD performs the threshold adjustment only at the beginning of each epoch, so as we did for PS, we can amortize its overhead over one million instructions.

Memory results

We evaluated PS and PD (with the performance guarantee algorithm) for memory using the SimpleScalar simulator with an aggressive out-of-order processor that has a nonblocking two-level cache hierarchy and RDRAM.[9] We ran our evaluation for the six SPEC benchmarks in Table 1.[7]

For comparison, we also implemented the original static and dynamic algorithms that Lebeck et al. describe.[4] The abbreviations in Table 1 are the original static (OS) algorithms. For the original dynamic (OD) algorithms, we use four settings for the required set of thresholds. Lebeck et al. suggested that the first set of thresholds, ODs, give the best $E\times D$ results for their simulation experiments with SPEC benchmarks on a system close to ours. The second set, ODn, also from the same authors, is the best setting for their Windows NT benchmarks on a system different from ours. We include these to show the thresholds' dependence on application and system. We calculated the third set, ODc, on the basis of their $E\times D$ competitive analysis.[4] Finally, we obtained the fourth set, ODt by extensive hand tuning, to account for the differences in the applications and in our system and the one Lebeck et al. studied.[4]

For tuning, we started with the thresholds described earlier and explored the space around them to find the set of best thresholds for each application—those that minimize energy within a 10-percent performance degradation.

Performance guarantee

As Table 1 shows, performance degradation is strongly sensitive to the power mode selected in the OS algorithms and to the thresholds in the OD algorithms. For the OD algorithms, thresholds tuned with competitive analysis (ODc) and those tuned for the Windows NT benchmarks (ODn) give unacceptable performance. This indicates that the original algorithms require painstaking manual tuning, which in turn depends heavily on the application and system. It also underlines the need for a performance guarantee.

We also enhanced the OS and OD algorithms to provide performance guarantees. For all the performance-guaranteed algorithms, we varied $\text{Slowdown}_{\text{limit}}$ from 5 to 30 percent. Table 2 shows the execution time degradation for the performance guaranteed algorithms. Across all 192 cases, the execution time degradation stays within the specified limit—evidence that our method for guaranteeing performance is indeed effective.

Energy savings

Figure 4 shows the energy consumption for parser and bzip. The results for the other SPEC

Table 2. Execution time degradation for performance-guaranteed (+) memory algorithms, including PS and PD.

Execution time degradation (percentage) for given algorithms Benchmarks	OSs+	OSn+	OSp+	ODs+	ODn+	ODc+	PS	PD
$Slowdown_{limit}$ = 5 percent								
bzip	1	5	5	3	4	4	4	4
gcc	1	4	4	3	4	3	3	3
gzip	1	4	5	3	3	3	3	3
parser	4	5	5	4	4	3	4	4
vortex	2	5	5	3	4	3	2	3
vpr	2	4	5	3	4	4	3	3
$Slowdown_{limit}$ = 10 percent								
bzip	1	9	10	6	8	7	8	8
gcc	2	8	9	6	7	6	7	6
gzip	1	6	9	4	7	6	6	6
parser	4	9	10	8	8	7	8	8
vortex	2	9	10	5	8	6	6	7
vpr	2	9	9	3	8	7	6	7
$Slowdown_{limit}$ = 20 percent								
bzip	1	9	19	6	17	14	16	16
gcc	1	14	17	6	14	11	15	12
gzip	1	6	18	4	13	8	11	12
parser	4	19	19	9	16	13	17	17
vortex	2	18	19	5	16	12	17	15
vpr	2	17	18	3	16	12	16	15
$Slowdown_{limit}$ = 30 percent								
bzip	1	9	29	6	25	20	24	24
gcc	1	14	26	6	21	17	23	19
gzip	1	6	26	4	19	8	18	19
parser	4	28	29	9	24	20	27	24
vortex	2	22	28	5	23	18	26	21
vpr	2	18	27	3	24	12	26	24

benchmarks, published elsewhere,[7] are similar. For OS+ and OD+, the figure shows the results of the setting with the least energy consumption for each application. Of all the algorithms that provide a performance guarantee, PD consumes the least energy in all cases. Across all the benchmarks, PS does better than OS+ in most cases,[7] but is never able to beat PD. PD and PS also compare favorably to the algorithms without performance guarantee in many cases. Overall, the results show that incorporating available slack into the control algorithm is effective in minimizing energy consumption given a specified performance degradation constraint.

Disk results

To evaluate our algorithms' performance in disk energy consumption, we used Dynamic Rotations Per Minute (DRPM), a multispeed disk model[3] that can service requests at a low rotational speed without the need to transition to full speed; however, the disk takes longer to service requests for all accesses at slower speeds. The performance-guarantee method and the PS and PD algorithms for disks are analogous to the memory case. Because no one has previously evaluated static algorithms for disks, we defined OS and OS+ algorithms analogous to the memory case: all disks stay at a fixed speed to service requests. The variation of OS+

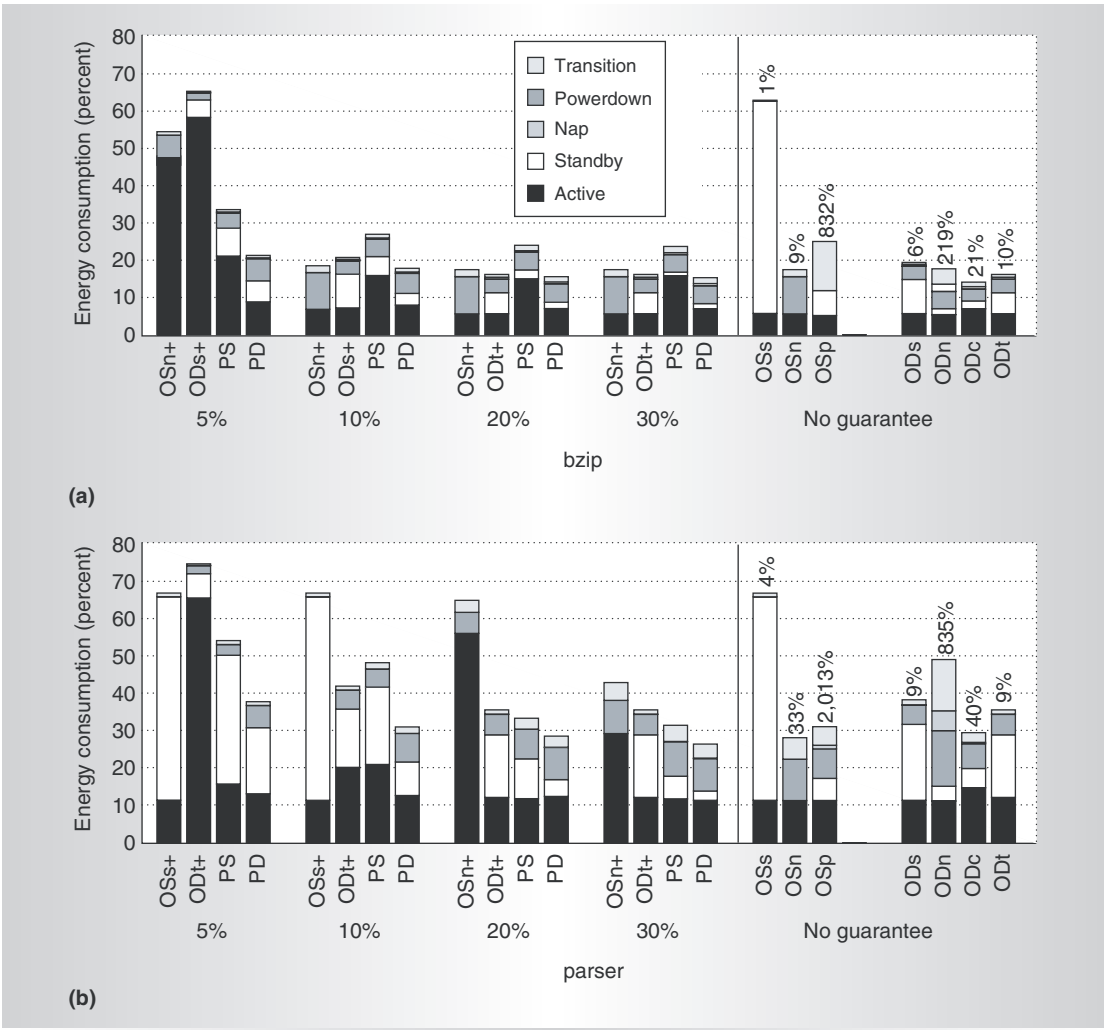

Figure 4. Memory energy consumption for bzip and parser with $\text{Slowdown}_{\text{limit}}$ varied at 5, 10, 20, and 30 percent from the case without energy management. For OD and OS (without performance guarantee), the numbers above the bars are the percentage of execution time degradation.

is OSr+ and represents a fixed *r*-thousand RPM speed, where $r = 3, 6, 9$, or 12. The dynamic algorithms, OD and OD+, are based on an algorithm that Gurumurthi et al. developed;[3] it dynamically transitions a disk from one speed to another, according to changes in the average response time and the request-queue length. The algorithm has five parameters; we explored these looking for ways to minimize the overall $E \times D$ for OD. OD and OD+ with this setting are OD1 and OD1+, respectively.

We also ran OD and OD+ at the parameter settings that Gurumurthi et al. used, which we call OD2 and OD2+. Our PD algorithm dynamically adjusts two of these parameters, the upper and lower thresholds for the change in average response time.

For our evaluation, we used both synthetic and real system traces, with the widely used DiskSim trace-driven simulator. The real system trace is the Cello trace collected from HP Cello File Servers in 1996.[10] We generated the synthetic traces, exponential and pareto, in way similar to the approach of Gurumurthi et al.[3]

Figure 5 presents results for two of the traces. It indicates that the original dynamic algorithms (OD1 and OD2) can incur unacceptably large application- and threshold-dependent execution time degradation (up to 39 percent across all traces[7]). In contrast, our performance-guaranteed algorithm is effective in *all* cases and

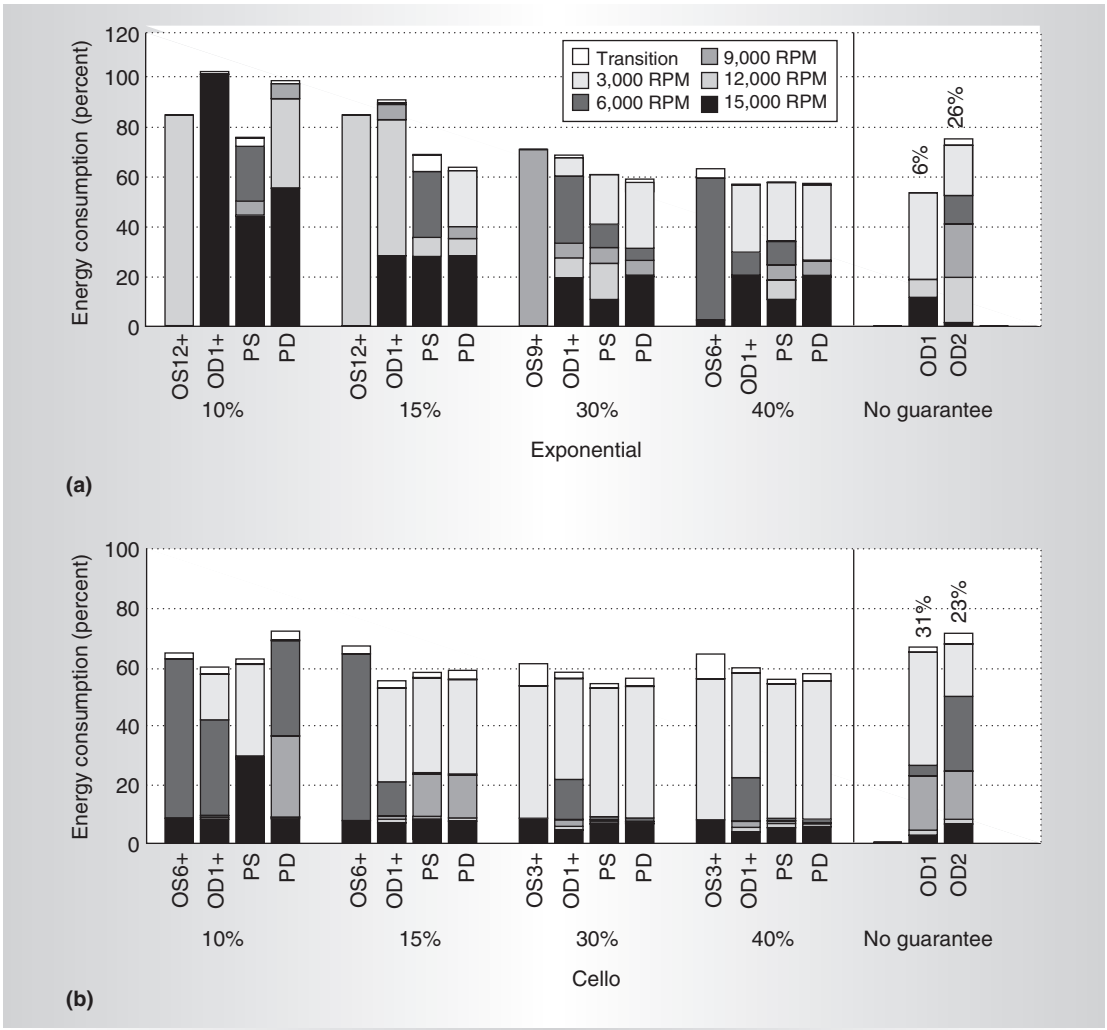

Figure 5. Disk energy consumption for synthetic and real system traces with $\text{Slowdown}_{\text{limit}}$ varied at 10, 15, 30, and 40 percent from the case without energy management. For OD and OS (without performance guarantee), the numbers above the bars are the percentage of execution time degradation.

never violated $\text{Slowdown}_{\text{limit}}$. PD and PS also save more energy than the corresponding (tuned best) original algorithms in most cases.

Thus, with little parameter tuning, these two methods can effectively reduce energy consumption while still providing a performance guarantee. However, in contrast to memory, across all traces,[7] PS is generally comparable to or better than PD because far more parameters are involved in dynamic algorithms. Thus, a simpler algorithm like PS is more beneficial for disks.

Both the PS and PD algorithms provide a performance guarantee and use the slack information that the performance-guarantee algorithm generates to guide energy management. For memory, PD consumes the least energy of all performance-guaranteed algorithms—up to 68 percent less than the best OD+. Even compared to the best hand-tuned OD (no performance guarantee), PD performs well in most cases. For disks, 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 cannot compete with the hand-tuned one in a few cases. Compared to all performance-guaranteed algorithms, PS cannot perform as well as PD for memory, but it is the best or close to the best in all but one case for disks.

Overall, the results show that our algorithms effectively overcome the limitations of current energy-control approaches, providing perhaps the first practical means of using the memory and disk low-power modes in current commercial systems and in designs that will eventually become the systems of the future. MICRO

Acknowledgments

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 grants CCR-0096126, IA-0103645, EIA-0224453, CCR-0209198, CCR-0205638, EIA-0224453, CCR-0305854, and CCR-0313286.

References

1. F. Moore, "More Power Needed," *Energy User News*, 25 Nov. 2002, http://www.energyusernews.com/CDA/ArticleInformation/features/BNP_Features_Item/0,2584,88121,00.html.
2. E.V. Carrera, E. Pinheiro, and R. Bianchini, "Conserving Disk Energy in Network Servers," *Proc. Int'l Conf. Supercomputing*, IEEE CS Press, 2003, pp. 86-97.
3. S. Gurumurthi et al., "DRPM: Dynamic Speed Control for Power Management in Server Class Disks," *Proc. Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 169-179.
4. A.R. Lebeck et al., "Power Aware Page Allocation," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 00), ACM Press, 2000, pp. 105-116.
5. C. Lefurgy et al., "Energy Management for Commercial Servers," *Computer*, vol. 36, no. 12, Dec. 2003, pp. 39-48.
6. Maximum Throughput Inc., "Power, Heat, and Sledgehammer," 2002, http://www.maxt.com/downloads/whitepapers/SledgehammerPowerHeat20411.pdf.
7. X. Li et al., "Performance Directed Energy Management for Main Memory and Disk," *Proc. Int'l. Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 04), ACM Press, 2004, pp. 271-283.
8. S. Irani, S. Shukla, and R. Gupta, *Competitive Analysis of Dynamic Power Management Strategies for Systems with Multiple Power Saving States*, tech. report UCI-ICS No. 01-50, Dept of Information and Computer Science, Univ. of California, Irvine, 2001.
9. D. Burger, T.M. Austin, and S. Bennett, *Evaluating Future Microprocessors: The SimpleScalar Tool Set*, tech. report CS-TR-1996-1308, Univ. of Wisconsin-Madison, 1996.
10. HP Labs, "Tools and Traces," 2004, http://www.hpl.hp.com/research/ssp/software/.

Xiaodong Li is a PhD student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include energy- and reliability-related issues in computer architecture. Li has an MSEE from Purdue University.

Zhenmin Li is a PhD student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include software debugging, data mining, and storage systems. Li has an ME in computer science from Tsinghua University, China.

Pin Zhou is a PhD student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. Her research interests include architectural support for software debugging, energy management for storage systems, and memory management. Zhou has an MS in computer science from Tsinghua University, China.

Yuanyuan Zhou is an assistant professor at the University of Illinois at Urbana-Champaign. Her research interests include database storage, architecture and operating system support for software debugging, power management, and memory management, storage systems. Zhou has an MA and a PhD, both in computer science, from Princeton University. She is a member of IEEE, ACM, and Usenix.

Sarita V. Adve is an associate professor in computer science at the University of Illinois at Urbana-Champaign. Her research interests include energy-, temperature-, and reliability-aware computer architectures and systems. Adve has a PhD in computer science from the University of Wisconsin-Madison. She is a recipient of an Alfred P. Sloan Fellowship; a

University Scholar of the University of Illinois; and a member of the IEEE, IEEE CS, and ACM SIGArch.

Sanjeev Kumar is a researcher at Intel Labs. His research interests include programming languages and computer architecture. Kumar has an MA and a PhD in computer science from Princeton University, an MS in computer science from Indiana University, and a BTech in computer science and engineering from the Indian Institute of Technology, Madras (now Chennai). He is a member of the IEEE, ACM, and Usenix.

Direct questions and comments about this article to Xiaodong Li, Siebel Center for Computer Science, Room 4111, 201 N. Goodwin Ave., Urbana, IL 61801; xli3@uiuc.edu.