Proceedings of the General Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

Energy Efficient Prefetching and Caching*

Athanasios E. Papathanasiou and Michael L. Scott
University of Rochester
{papathan,scott}@cs.rochester.edu
http://www.cs.rochester.edu/{~papathan,~scott}

Abstract

Traditional disk management strategies—prefetching and caching in particular—are designed to maximize performance. In mobile systems they conflict with strategies that attempt to save energy by powering down the disk when it is idle. We present new rules for prefetching and caching that maximize power-down opportunities (without performance loss) by creating an access pattern characterized by intense bursts of activity separated by long idle times. We also describe an automatic system that monitors past application behavior in order to generate appropriate prefetching hints, and a general system of kernel enhancements that coordinate I/O activity across all running applications.

We have implemented our system in the Linux kernel, and have measured its performance and energy consumption via physical instrumentation of a running laptop. We describe our implementation and present quantitative results. For workloads including a mix of sequential access to large files (multimedia), concurrent access to large numbers of files (compilation), and random access to large files (speech recognition), we report disk energy savings of 60–80%, with negligible loss in throughput or interactive responsiveness.

1 Introduction

Prefetching and caching are standard practice in modern file systems. They serve to improve performance—to increase throughput and decrease latency—by eliminating as many I/O requests as possible, and by spreading the requests that remain as smoothly as possible over time. This strategy results in relatively short intervals of inactivity. It ignores the goal of energy efficiency so important to mobile systems, and in fact can frustrate that goal. Magnetic disks, network interfaces, and similar devices provide low-power states that save energy only when idle intervals are relatively long. A smooth access pattern can eliminate opportunities to save energy even during such light workloads as MPEG and MP3 playback.

*This work was supported in part by NSF grants EIA-0080124, CCR-9988361, and CCR-0204344; by DARPA/AFRL contract F29601-00-K-0182; and by Sun Microsystems Laboratories.

The aim of our work is to create bursty access patterns for devices with non-operational low-power states, increasing the average length of idle intervals and maximizing utilization when the device is active, without compromising performance. At present we are focusing on hard disks.

Typical hard disks for mobile systems support at least four power states: Active, Idle, Standby, and Sleep. The disk only works in the Active state. In the Idle state the disk is still spinning, but the electronics may be partially unpowered, and the heads may be parked or unloaded. In the Standby state the disk is spun down. The Sleep state powers off all remaining electronics; a hard reset is required to return to higher states. Individual devices may support additional states. The IBM TravelStar, for example, has three different Idle sub-states.

One to three seconds are typically required to transition from Standby to Active state. During that spin-up time the disk consumes 1.5–2X as much power as it does when Active. The typical laptop disk must therefore remain in Standby state for a significant amount of time—on the order of 5–16 seconds for current laptop disks—to justify the energy cost of the subsequent spin-up. The energy savings in Idle state approaches that of Standby state, particularly in very small form factor devices, and the time and energy to move from Idle to Active state are minimal. Hence, even modest increases of the disk's idle interval can lead to significant energy savings.

In previous work we made the case for energy efficiency through burstiness and demonstrated the energy efficiency potential of aggressive prefetching for rate-based applications with sequential access patterns [28, 29]. In this paper, we provide a detailed description of the design and implementation of our prefetching and caching algorithms. In addition, we provide experimental results for more challenging workload mixes, including applications that make non-sequential accesses to multiple files.

We have implemented our system in the Linux 2.4.20 kernel. We have extended the memory management and file system of the operating system with algorithms and data structures to:

- Quickly identify the working set of the executing job mix and dynamically control the amount of memory used for aggressive prefetching and buffering of dirty data.

- Coordinate the generation of I/O requests among concurrently running applications, so that they are serviced by the device during the same small window of time.

In designing and evaluating our system we have focused on long-running applications that are commonly executed on mobile systems, and that generate a large number of I/O requests, separated by idle times too short to be exploited by the hard disk for energy savings. Examples include data transfer (copying); encoding, decoding, and compression; compilation and build; scripting utilities for system maintenance; and computationally demanding user interface tasks, e.g. speech recognition.

The following Section provides rules for optimal prefetching when the goal is energy efficiency. Sections 3 and 4 describe the design of our prefetching and request deferring mechanisms. Section 5 presents experimental results. Section 6 discusses previous work. Section 7 summarizes our conclusions.

2 Prefetching for Energy Efficiency

To illustrate the differences that arise when prefetching has the additional goal of improving disk energy efficiency, consider an application with reference string $\{\ A\ B\ C\ D\ E\ F\ G\ldots\ \}$ and a steady rate of one access every 10 time units. Assume that the buffer cache has room for three blocks, and that the disk requires one time unit to fetch a block that misses in the cache.[1]

Figures 1–3 illustrate the execution of the application while accessing the first 6 elements of the reference string, with an optimal replacement strategy and three different prefetching strategies: Figure 1 illustrates a *fetch-on-demand* strategy; Figure 2 illustrates a strategy that follows the prefetching rules of Cao et al. [1]; Figure 3 illustrates an energy-conscious prefetching strategy that attempts to maximize the length of the disk idle intervals. Under the *fetch-on-demand* strategy the application runs for 66 time units and experiences 6 misses. The disk idle time is spread across 6 intervals of 10 time units each. With a traditional prefetching strategy (Figure 2) run time decreases to 61 time units and the application experiences just one miss. The distribution of disk idle time remains practically unchanged, however: 5 intervals of 9 time units each and one of 8 time units. The energy-conscious prefetching strategy (Figure 3) achieves the same execution time and the same number of misses as traditional prefetching, but the disk idle time appears in two much longer intervals: one of 27 time units and one of 28 time units.

Time	Application	Disk	Cache State		
1		fetch(A)			
2−11	access(A)	idle	A		
12		fetch(B)	A		
13−22	access(B)	idle	A	B	
23		fetch(C)	A	B	
24−33	access(C)	idle	A	B	C
34		fetch(D)	A	B	C
35−44	access(D)	idle	D	B	C
45		fetch(E)	D	B	C
46−55	access(E)	idle	D	B	C
56		fetch(F)	D	B	C
57−66	access(F)	idle	D	B	C
67		fetch(G)	D	B	C

Figure 1: Optimal Replacement and Fetch-on-Demand. Accessing the reference string up to element F requires 66 time units. The disk idle time appears in 6 intervals of 10 time units each.

2.1 Rules for Optimal Prefetching Revisited

Traditional prefetching strategies aim to minimize execution time by deciding:

- when to fetch a block from disk,
- which block to fetch, and
- which block to replace.

Previous work [1] describes four rules to make these decisions in a performance-optimal fashion:

1. *Optimal Prefetching:* Every prefetch should bring into the cache the next block in the reference stream that is not yet in the cache.
2. *Optimal Replacement:* Every prefetch should discard the block whose next reference is farthest in the future.
3. *Do no harm:* Never discard block A to prefetch block B when A will be referenced before B.
4. *First Opportunity:* Never perform a prefetch-and-replace operation when the same operations (fetching the same block and replacing the same block) could have been performed previously.

The first three rules answer the questions of what to prefetch (rules 1 and 2) and (partially) when to prefetch (rule 3), and apply equally well to energy-conscious prefetching. The fourth rule suggests that a prefetch operation should be issued when (a) a prior fetch completes,

[1]Note that the application in this example consumes data at a rate slower than the bandwidth of the disk. The goal of our work is to increase the length of disk idle interval for workloads that run for relatively long periods and do not require the disk to be active constantly.

Time	Application	Disk	Cache State		
1		fetch(A)			
2	access(A)	prefetch(B)	A		
3	access(A)	prefetch(C)	A	B	
4−11	access(A)	idle	A	B	C
12	access(B)	prefetch(D)	A	B	C
13−21	access(B)	idle	D	B	C
22	access(C)	prefetch(E)	D	B	C
23−31	access(C)	idle	D	E	C
32	access(D)	prefetch(F)	D	E	C
33−41	access(D)	idle	D	E	F
42	access(E)	prefetch(G)	D	E	F
43−51	access(E)	idle	G	E	F
52	access(F)	prefetch(H)	G	E	F
53−61	access(F)	idle	G	H	F
62	access(G)	prefetch(I)	G	H	F

Figure 2: Optimal Replacement and Traditional Prefetching. Accessing the reference string up to element F requires 61 time units. The disk idle time appears in 5 intervals of 9 time units each and one of 8. In a long run the average idle interval length will be 9 time units.

Time	Application	Disk	Cache State		
1		fetch(A)			
2	access(A)	prefetch(B)	A		
3	access(A)	prefetch(C)	A	B	
4−11	access(A)		A	B	C
12−21	access(B)	idle	A	B	C
22−30	access(C)		A	B	C
31	access(C)	prefetch(D)	A	B	C
32	access(D)	prefetch(E)	D	B	C
33	access(D)	prefetch(F)	D	E	C
33−41	access(D)		D	E	F
42−51	access(E)	idle	D	E	F
52−60	access(F)		D	E	F
61	access(F)	prefetch(G)	D	E	F

Figure 3: Optimal Replacement and Energy-conscious Prefetching. Accessing the reference string up to element F requires 61 time units. The disk idle time appears in one interval of 27 time units and one of 28. In a long run the average idle interval length will be 28 time units.

or (b) the block that would be discarded was just referenced. Identifying this first opportunity can be difficult, and indeed most real systems are considerably less aggressive. As noted by Patterson et al. [31], the full performance benefit of prefetching will be achieved even if the prefetch completes just barely before the corresponding access.

We observe, however, that any uniform prefetch policy will tend to produce a smooth access pattern, with short idle times. As an example, consider a system with a cache size of k blocks, a reference string $\mathcal{R} = \{\ b_1\ b_2 \ldots\ b_k\ b_{k+1} \ldots\ b_n\ \}$, where $n > k$, and an inter-access time of $\mathcal{A}$. If we follow rule 4, we will fetch block b_{k+1} immediately after the reference to b_1, b_{k+2} immediately after the reference to b_2 and so on, breaking a possible disk idle interval of length $k \times \mathcal{A}$ into intervals of length $\mathcal{A} - \mathcal{F}$, where $\mathcal{F}$ represents the time to fetch a block from the disk. Assuming $\mathcal{F} < \mathcal{A}$, an energy-conscious prefetching algorithm should not initiate the prefetch of b_{k+1} until $\mathcal{F}$ time units prior to its reference. Then in a single burst it should prefetch blocks $\{\ b_{k+1} \ldots\ b_{2k-1}\}$, replacing blocks $\{\ b_1 \ldots\ b_{k-1}\}$. Intuitively, this policy alternates between "first opportunity" and "just in time" prefetching, depending on whether the disk is currently in the Active state.

Based on the above discussion, to accommodate the requirement of energy efficiency, we replace rule 4 with the following:

$4'$. *Maximize Disk Utilization:* Always initiate a prefetch operation after the completion of a fetch, if there are blocks available for replacement (with respect to Rule 3).

$5'$. *Respect Idle Time:* Never interrupt a period of inactivity with a prefetch operation unless the prefetch has to be performed immediately in order to maintain optimal performance.

Rule $4'$ guarantees that a soon-to-be-idle disk will not be allowed to become inactive if there are blocks in the cache that may be replaced by blocks that will be accessed earlier in the future. This way disk utilization is maximized and short intervals of idle time that cannot be exploited for energy efficiency are avoided. Rule $5'$ attempts to maximize the length of a period of inactivity without degrading performance. Note that the rule implies that the prefetching algorithm should take into account additional delays due to disk activation or congestion as well as the time required for a fetch to complete. An algorithm that follows rules $4'$ and $5'$ will lead to the same hit ratio and execution time as an algorithm following the rules of Cao et al., but will exhibit fewer and longer periods of disk inactivity whenever possible.

State	Value
Active Power	2.0 W
Idle Power	0.61 W
Idle-to-Active Energy	1.5 J
Idle-to-Active Time	0.55 s
Active-to-Idle Energy	2.4 J
Active-to-Idle Time	0.85 s
Standby Power	0.15 W
Spin up Energy	5.0 J
Spin up Time	1.6 s
Spin down Energy	2.94 J
Spin down Time	2.3 s

Table 1: Abstract disk model parameters for computing the potential of aggressive speculative prefetching. Values are based on the characteristics of the Hitachi DK23DA hard disk.

2.2 Prefetching's Potential for Energy Savings

In comparison to traditional prefetching, which aims to reduce the latency of access to disks in the active state, prefetching for energy efficiency has to be significantly more aggressive in both quantity and coverage. A traditional prefetching algorithm can fetch data incrementally: its goal is simply to request each block far enough in advance that it will already be available when the application needs it. It will improve performance whenever its rate of "true positives" (prefetched blocks that turn out to indeed be needed) is reasonably high, and its "false positives" (prefetched blocks that aren't needed after all) don't get in the way of fetching the "false negatives" (needed blocks that aren't prefetched). By contrast, an energy-reducing prefetching algorithm must fetch enough blocks to satisfy all read requests during a lengthy idle interval. Minimizing "false negatives" is more important than it is in traditional prefetching, since the energy cost and performance penalty of power state transitions is very high. These differences suggest the need to fetch much more data, and much more speculatively, than has traditionally been the case. Indeed, prefetching for burstiness more closely resembles prefetching for disconnected operation in remote file systems [19] than it does prefetching for low latency.

Fortunately, avoiding a disk power-up operation through speculative prefetching can justify fetching a large number of "false positives" in terms of energy consumption. To show the energy saving potential of speculative prefetching, we present calculated values for an aggressive, speculative prefetching algorithm that follows the rules for energy-efficient prefetching (Section 2.1) with various "false-positive" to "true-positive" ratios, and compare to a prefetching algorithm that follows Rules 1–4 and prefetches only what is needed. We assume an abstract disk model based on the Hitachi DK23DA hard disk with the characteristics shown in Table 1 and an optimal power management policy (one that spins the disk down whenever it can save energy by doing so, without performance loss). We include in our calculations the cost of reading data into memory. Based on the specifications of Micron's 512 Mb $\times$ 16 SDRAM DDR dies [25], this is 100 μJ per page.

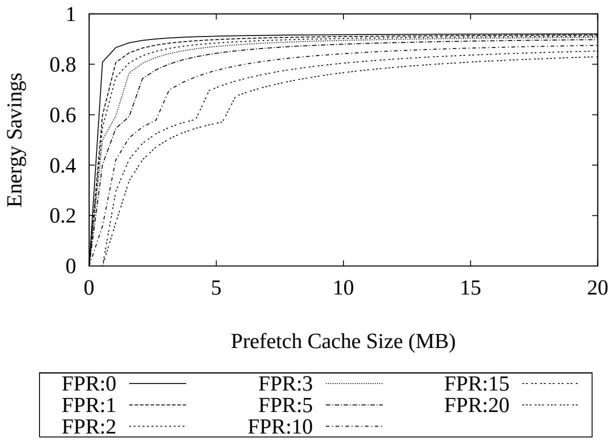

Figure 4: Energy savings of a speculative prefetching algorithm across various prefetch buffer sizes and "false-positive" to "true-positive" ratios (*FPR*) ranging from 0 to 20. An application with a "slow" data consumption rate of 16 KB/sec is assumed.

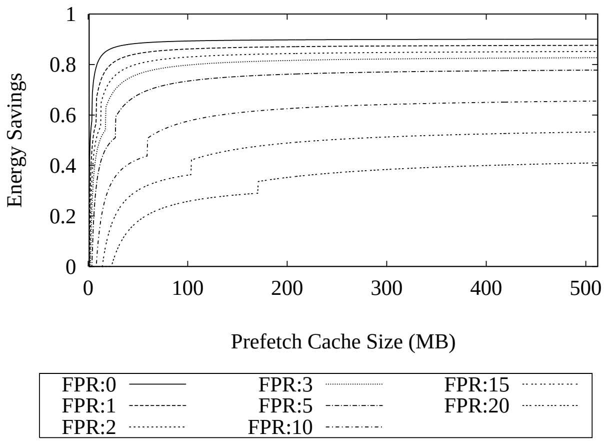

Figure 5: Energy savings of a speculative prefetching algorithm across various prefetch buffer sizes and "false-positive" to "true-positive" ratios (*FPR*) ranging from 0 to 20. An application with a "fast" data consumption rate of 240 KB/sec is assumed.

Figures 4 and 5 present results for "false-positive" to "true-positive" ratios (*FPR*) ranging from 0 to 20 for a pair of applications consuming data at 16 KB/s and 240 KB/s, respectively. For the slower application (Figure 4), with even a small amount of memory dedicated to prefetching, significant energy savings can be achieved. Just 5 MB of memory prefetching leads to over 50% energy savings, even for "false-positive" to "true-positive" ratios as high as 20 to 1, i.e. even if we prefetch more than 20 times as much data as we actually use. For the faster application (Figure 5), a 50% savings in disk + memory energy can be achieved with a 25 MB prefetch buffer for "false-positive" to "true-positive" ratios of up to 5 to 1. Larger ratios require significantly more prefetch memory.

3 Design of Energy-Aware Prefetching

As mentioned in the previous Section, any prefetching algorithm has to decide *when to fetch a block*, *which block to fetch*, and *which block to replace*.

3.1 Deciding When to Prefetch

Based on Rules $4'$ and $5'$ presented in the previous Section, our prefetching algorithm attempts to fetch from the disk as many blocks that are going to be accessed in the future as possible during periods when the disk is active, and to postpone prefetching operations until the latest opportunity during periods when the disk is idle. To approximate such behavior, we introduce an Epoch-based algorithm into the memory management mechanisms of the operating system. Each epoch consists of two phases: an *active* phase and an *idle* phase. Prefetching occurs during the active phase. To manage this prefetching, the OS must:

1. Predict future data accesses. Prediction is based on manual or automatic hints (Section 3.2).
2. Compute the amount of memory that can be used for prefetching or storing new data. This step requires identifying quickly the currently useful in-memory data: the workload's working set and cached files.
3. Free the required amount of memory by unmapping pages and flushing dirty, mapped pages.
4. Prefetch or reserve buffers for writing new data proportional to each executing application's memory resource requirements. The goal of this step is to coordinate I/O accesses across concurrently running applications so that they all generate their next demand miss at approximately the same time.

When the active phase completes, the idle phase of the epoch begins. During the idle phase, accesses to each active file are monitored by the operating system. Based on the access pattern and rate, and the state of the buffer cache, the operating system attempts to predict the next miss time for each file, and to initiate a new prefetching cycle early enough to achieve optimal performance.

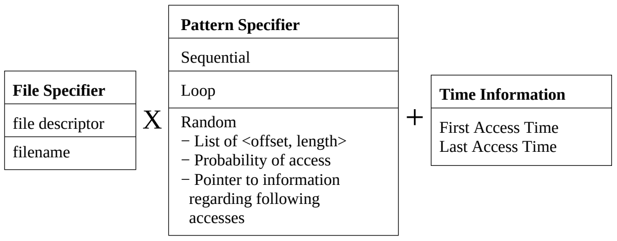

Figure 6: Hint Interface: Applications can disclose future file accesses using the hint interface. The interface specifies a file using a filename or a file descriptor and provides a pattern specifier that can be one of sequential, loop and random. In addition, an estimation of the time of first and last access can be given, if available.

Prefetching has to start well in advance in order to hide both the latency of the fetch itself and the delay of a possible disk reactivation (power-up).

The start of a new epoch is triggered by one of the following events:

1. A new prefetching cycle has to be initiated in order to maintain optimal performance.
2. A demand miss took place. In this case the prefetching algorithm has failed to load into memory all required data, or has mistakenly evicted useful pages from the buffer cache during the active phase. The application that issued the request may experience an increased delay in addition to the penalty of a demand miss if the disk has been placed into a low-power state.
3. The system is low on memory resources. The page freeing logic has to be executed.

3.2 Deciding What to Prefetch

To achieve as high a degree of accuracy as possible, prediction is based on *hints*. Our hint interface is an extension of that described by Patterson et al. [31]. The hint interface consists of a file specifier that can be a file descriptor or a filename, a pattern specifier that shows whether the file will be accessed in a sequential, loop or random way, and estimates of the times of the first and last accesses to the file. The time information can be represented as offsets from the start of the application execution or from the disclosure of the hint. For randomly accessed files the hint interface also provides a list of "hot" clusters within the file, the probability of access to each cluster, and an optional pointer to a file that provides sets of clusters that have a significant probability of being accessed within a certain time period of the access to a specific page. Such information can be generated through profiling. Figure 6 summarizes the hint interface. Hints are submitted to the operating system through a new set of system calls.

New applications can use the new system calls directly in order to provide hints to the operating system.

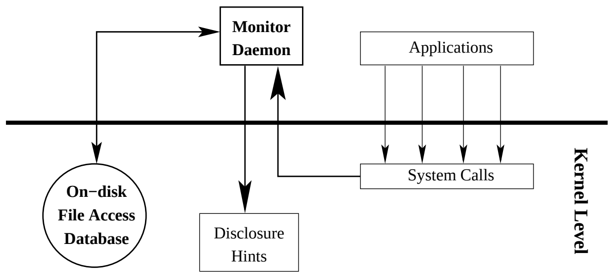

Figure 7: The monitor daemon provides hints automatically on behalf of applications that do not support the hint interface. It tracks file system use by monitoring system calls. It analyzes the collected information, and creates a hint database for each application whose access pattern may be harmful to energy efficiency. When an application with hints re-executes, its file accesses will automatically be disclosed to the operating system by the monitor daemon.

For existing applications, efficient file prediction may be achieved by monitoring past file accesses and taking advantage of the *semantic locality* that appears in user behavior [20]. In our current prototype, a *monitor daemon* tracks the file activity of all executing applications by tracing the `open`, `close`, `read`, `write` `execve`, `exit`, and `setpgid` system calls. Its goals are two-fold: to prepare a database describing file accesses for each application based on the collected information (*Access Analysis*) and to generate hints automatically on behalf of applications based on the information in the database (*Hint Generation*). Figure 7 illustrates the operation of the monitor daemon.

Since access analysis can be a computationally intensive operation, it takes place only at periods during which energy consumption is not a concern (when the mobile system is plugged in). The analysis utility discovers associations among file accesses and applications, and records the access pattern (sequential, loop, and random) and the time from the beginning of execution to the first and last access to each file. For randomly accessed files the utility also identifies clusters of pages within the file that tend to be accessed together, the probability of an access to a given cluster, and for each cluster X a list of clusters that tend to be accessed within a certain time (currently one minute) of the access to that cluster. The analysis utility stores this information for applications that use the file system for relatively long periods of time (currently at least 1 minute) and cause access patterns that spread a large amount of idle time across many short intervals. Occasional accesses that appear during interactive workloads can be handled adequately by previously proposed disk power management policies [4, 5, 15, 22].

In order to associate related file accesses that are generated by different processes (as an example consider accesses caused by the various invocations of the `gcc` compiler during the execution of a `make` operation), we take advantage of the process group structure of Unix. The analysis utility associates accesses with the process group leader of the process that caused the access. The database that maintains the hinting information is indexed by the absolute pathname of the executable with which the access was associated, the directory in which the access took place, and the arguments used to make the invocation.

During normal system operation, the monitor daemon tracks `execve` system calls. If there are hints available in the database for the specified application, the daemon automatically generates the hints on behalf of that application.

3.3 Deciding What to Replace

As mentioned in Section 3.2, the first step during the initiation of a new prefetching cycle is to compute the number of pages that can be used for prefetching. Unlike a traditional prefetching algorithm, which can make incremental page-out decisions over time, the energy-conscious prefetching algorithm must make a decision significantly ahead of time, predicting the number of cached pages that are not going to be accessed during a possibly long idle phase.

Intuitively, two parameters determine the number of pages that should be freed at the beginning of each epoch. First, the reserved amount of memory should be large enough to contain all predicted data accesses. Second, prefetching or future writes should not cause the eviction of pages that are going to be accessed sooner than the prefetched data. Since our goal is to maximize the length of the hard disk's idle periods, we use the type of the first miss during an epoch's idle phase to refine our estimate of the number of pages available for prefetching in the subsequent epoch.

We categorize page misses as follows:

1. Eviction miss: A miss on a page that used to reside in the buffer cache, but was evicted in favor of prefetching. Such a miss suggests that number of pages used for prefetching in the current epoch was too large.
2. Prefetch miss: A miss on a page for which there was a prediction (hint) that it was going to be accessed. Such a miss suggests that a larger prefetch cache size could have been used during the current epoch.
3. Compulsory miss: A miss on a page for which there is no prior information.

An eviction miss during the idle phase of an epoch suggests that the prefetching depth (the total number of pages prefetched) should decrease, while a prefetch miss suggests that the prefetching depth should increase. Controlling the prefetching depth based on eviction misses provides a way to protect the system from applications that may issue incorrect hints. Incorrect hints will increase the number of eviction misses and lead to a reduced prefetching depth. Section 4.4 describes in detail how our system adjusts the prefetching depth.

4 Implementation

We have implemented our epoch-based energy-efficient prefetching algorithm in the Linux kernel, version 2.4.20. We describe that implementation in this Section.

4.1 Hinted Files

Prefetching hints are disclosed by the monitor daemon or by applications themselves. In addition, the kernel automatically detects and generates hints for long sequential file accesses. The full set of hints, across all files is maintained in a doubly linked list, sorted by estimated first access time. In addition to the information listed in Section 3.2), the kernel keeps track of the following:

- *Memory Status*: Shows the caching state for a file. A file can be completely uncached, have only its metadata cached, be partially cached, or be fully cached. When a hint is disclosed using the filename of the file, the caching state of the corresponding file is unknown. In such cases the kernel assumes that the file is completely uncached, and aggressively attempts to prefetch metadata and data. The kernel also keeps track of `open` and `close` requests in order to associate hinted accesses with their corresponding file descriptors.
- *Rate of access*: Keeps track of the average rate in pages per second with which the file is being accessed.
- *Prefetch Depth*: Shows the number of pages that were used for prefetching data for the corresponding hinted file during the current epoch. The prefetching algorithm allocates memory for prefetching to each hinted file proportional to its average access rate.

4.2 Prefetch Thread

Idle interval length can be limited because of a lack of coordination among requests generated by different applications. Even if there are long idle periods in the access pattern of every application, we will be unable to power down the disk unless these patterns are in phase. The operating system must ensure that read and write requests from independent concurrently running applications are issued during the same small window of time. Write activity can easily be clustered because most write requests are issued by a single entity: the update daemon. Similarly, page-out requests are issued by the swap daemon. Read and prefetching requests, however, are generated within a process context independent of other applications. To coordinate prefetching requests across all running applications we introduce a centralized entity that is responsible for generating prefetching requests for all running applications: the prefetch daemon. The prefetch daemon is analogous to the update daemon and handles read activity. Through the prefetch thread the problem of coordinating I/O activity is reduced to that of coordinating three daemons.

During the active phase the prefetch thread goes through the list of hints and attempts to prefetch data in a way that will equalize the expected time to first miss across all applications. The prefetching algorithm sets an initial target idle period and for each hinted file that is predicted to be accessed within the target period it aggressively prefetches metadata and data proportional to the average access rate. The target idle time is then increased gradually until all hinted files are fully prefetched or the amount of memory available for prefetching is depleted.

The target idle times used by the prefetching algorithm are based on the “breakeven” times of modern mobile hard disks: the times over which the energy savings of low-power states exactly equal the energy cost of returning to the Active state. Currently, we are using target idle times of 2 seconds, which corresponds to the highest low-power state of the IBM TravelStar [17], 5 seconds, which corresponds to the IDLE-3 low-power state of the TravelStar and the Idle state of the Hitachi DK23DA, and multiples of the Standby state breakeven time (16 seconds for the DK23DA). When the prefetching cycle completes, the algorithm predicts that the length of the idle phase of the upcoming epoch will be the period for which the prefetch thread successfully prefetched all necessary data.

4.3 Prefetch Cache

We have augmented the kernel’s page cache with a new data structure: the prefetch cache. Pages requested by the prefetch daemon are placed in the prefetch cache. Each page in the prefetch cache has a timestamp that indicates when it is expected to be accessed. When a page is referenced, or its timestamp is exceeded, it is moved to the standard LRU list and is thereafter controlled by the kernel’s page reclamation policy.

4.4 Eviction Cache

To choose an appropriate size for the prefetch cache, we must keep track of pages that are evicted in favor of prefetching (Section 3.3). We do this using a new data structure called the *eviction cache*. This cache retains the metadata of recently evicted pages (though not their contents!) along with a unique serial number, called the *eviction number*. The eviction number counts the number of pages that have been evicted in favor of prefetching. During the idle phase, if an eviction miss takes place, the difference between the page’s eviction number and the current epoch’s starting eviction number indicates the number of pages that were evicted in favor of prefetching *without* causing an eviction miss. It can be used as an estimate of a suitable prefetching depth (prefetch cache size) for the upcoming epoch. The prefetch depth does not change in the case of compulsory misses or misses on pages that were evicted in prior epochs. It is increased by a constant amount if there were no misses, or if there were only prefetch misses.[2] In order to avoid significant oscillations in the prefetching depth we use a moving average function. During system initialization, when there is no prior information available, the prefetch depth is set to the number of idle pages in the system.

4.5 Handling Write Activity

The original Linux kernel uses a variant of the approximate interval periodic update policy [26]. The update daemon runs every 5 seconds, and flushes all dirty buffers that are older than 30 seconds. This policy is bad for energy efficiency: under even light write workloads idle time will appear in intervals of 5 seconds or less.

In our current implementation, we use a modified update daemon that flushes all dirty buffers once per minute. In addition, we have extended the `open` system call with an additional flag that indicates that write-behind of dirty buffers belonging to the file can be postponed until the file is closed or the process that opened the file exits. Such a direction is useful for several common applications, such as compilations and MP3 encoding, that produce files that do not have strict intra-application reliability constraints. For legacy code, users can specify in a configuration file the names of applications (`gcc`, for example) or file classes (e.g. `.o`) for which write-back can be delayed. The monitor daemon then provides a "flush-on-close" or "flush-on-exit" directive to the operating system.

A side effect of `create` and `write` accesses is that they can lead to unpredicted read operations of file metadata, directories, or file system metadata (e.g. the free block bitmaps). Such read operations are synchronous and can interrupt lengthy idle intervals. For this reason the monitor daemon keeps track of `write` and `create` accesses and generates hints for any files that may be written or created by a certain application. During the prefetching cycle of the epoch, the prefetch thread speculatively prefetches file system metadata for files associated with `write` or `create` hints. At the memory management level of the operating system, file system structure is not known, since it is file system dependent. In order to enable file system metadata prefetching we have extended the Linux virtual file system with two new function pointers: `emulate_create` and `emulate_write`. Both functions execute the low level file system code that is normally executed during file creation or disk block allocation without actually modifying the file system (only the corresponding read requests are issued). Currently, we have an implementation of the two functions for the Linux Ext2 file system.

Finally, write activity can lead to unexpected I/O requests during an idle phase if the system runs out of memory and the page daemon has to start paging. For this reason, during the active phase the prefetch thread reserves a portion of the available memory for future writes. The amount of memory allocated to each file is proportional to its write rate. At the end of the prefetching cycle, the prefetch threads clears a number of pages equal to the total number of reserved pages. Pages are reserved only for files that have been active for at least a certain time period (5 seconds) and have a fast write rate (at least 1 page per second).

4.6 Power Management Policy

At the completion of the prefetching cycle, the prefetch thread predicts the length of the upcoming idle phase (Section 4.2). This prediction is forwarded to the kernel's power management policy. If the predicted length is longer than the hard disk's Standby breakeven time, the disk is set to the Standby state within one second after it becomes idle (the disk may be servicing requests for several seconds after the prefetch thread completes its execution).

Since the prediction provided by the prefetching system is based only on file accesses associated with hints, there is a significant chance of decreased prediction accuracy during highly interactive workloads that are not handled efficiently by the monitor daemon. To avoid harmful spin-down operations, the power management algorithm monitors the accuracy of the prefetching system's predictions. If the prefetching system repeatedly mispredicts the length of the idle phase, providing predictions that specify idle periods longer than the disk's Standby breakeven time, when the actual idle period length is shorter than the breakeven time, the power management policy reverts to a dynamic-threshold spin-down policy, ignoring predictions coming from the prefetch thread until their accuracy is increased.

For rate-based and non-interactive applications, the same information that allows the operating system to identify opportunities for spin-down can also be used to predict appropriate times for spin-up, rendering the device available just in time to service requests. For this purpose during the idle phase of an epoch the operating system monitors the rate at which pages belonging to sequentially accessed files are consumed from the prefetch cache. Based on this rate, the number of pages remaining in the prefetch cache, and the disk's power-up delay, it computes the time at which the disk has to be activated in order to avoid any noticeable delays.

[2] In the current implementation we increase by the `pages_low` value, used by the pageout daemon. In Linux this is 1/128 of the total number of memory pages but not less than 20 or more than 255.

Disk	Hitachi
Capacity	10-30GB
Active	2.1W
Active Idle	1.6W (24%)
Low-Power Idle	0.6W (71%)
Standby	0.15W (93%)
Spin up	3.0W
Spin-up time	1.6s
Breakeven time	16s

Table 2: Energy consumption parameters for the Hitachi DK23DA hard disk. The disk supports three low power states: Active Idle (a portion of the electronics is off), Low Power Idle (the heads are parked) and Standby.

5 Experimental Evaluation

In this section, we compare our energy-conscious prefetching algorithm (called *Bursty*) to the standard *Linux* 2.4.20 policies across systems with memory sizes ranging from 64 MB to 492 MB. We use the power management policy described in Section 4.6 for the Bursty system, and a 10 second fixed threshold power management policy for Linux. Linux 2.4.20 supports a conservative prefetching algorithm that reads up to 128 KB (32 4KB pages) in advance and leads to very short periods of disk idle time for our experimental workloads. Hence, the power management policy degenerates to the No-Spin-Down policy.

Our experiments were conducted on a Dell Inspiron 4100 laptop with 512 MB of total memory and a Hitachi DK23DA hard disk. Table 2 presents the power consumption specifications of the disk. Power measurements were collected from the disk's two 5V supply lines. To measure power, both the voltage and current need to be known. The voltage is assumed to be fixed at 5V. We used $100m\Omega$ precision resistors in order to dynamically measure the current through the supply lines. The voltage drop across the resistors was measured through the following National Instruments setup: a SCXI-1308 voltmeter terminal block, a SCXI-1102C (32 channel multiplexer/amplifier and 10kHz filter) module, a SCXI-1000 chassis (for the mentioned modules), a SCXI-1349 card to chassis cable, and a PCI-6052E Analog-to-Digital converter card (capable of 16 bit resolution, 333Ksamples/second). The gain of the SCXI-1102C was set to 100 and the PCI-6052E was set to have a range of $+/-10$V. The sampling rate for each signal was 1000 samples/second. Measurements were collected and converted to current and power using National Instrument's LabView (version 6.0.2) software.

The idle interval histogram graphs (Figures 8 and 9) are based on traces collected from the ATA/IDE disk driver during the execution of our workloads scenarios. In order to avoid any disk activity caused by the tracing system, we used a pinned-down 20 MB memory buffer that was periodically transmitted to a logging system through the network.

We use four different workload scenarios, with different degrees of I/O intensity. The first, MPEG playback of two 76 MB files (referred to as *MPEG*), represents a relatively intensive read workload. The second (referred to as *Concurrent*) is a read and write intensive workload, which involves concurrent MP3 encoding and MPEG playback. The MP3 encoder reads 10 WAV files with a total size of 626 MB and produces 42.9 MB of data. During the MP3 encoding process, the MPEG player accesses two files with a total size of 152 MB. Our third workload (referred to as *Make*) is a full Linux kernel compilation. Finally, in order to evaluate our system for workloads that consist of random accesses to files, we use the SPHINX-II Speech Recognition utility [16] from CMU (referred to as *SPHINX*). During speech recognition SPHINX accesses a 128 MB language model file in an apparently random way. As input we use a set of recorded dialogues that were used in the evaluation of the TRIPS Interactive Planning System [8]. We used a subset of the dialogues to prepare the access pattern database (Section 3.2), and evaluated the system on a different subset.

The metrics used in the comparisons are:

Length of idle periods Longer idle periods can be exploited by more power-efficient device states. Increasing the length of idle periods can improve any underlying power management policy.

Energy savings We compare the energy savings achieved for Linux and our Bursty system for various memory sizes.

Slowdown A significant challenge for our Bursty system is to minimize the performance penalties that may be caused by increased disk congestion and disk spin-up operations.

Figures 8-11 show the distribution of idle time intervals for our workload scenarios. We present results for our Bursty system using various memory sizes. For the first three workloads, the memory size ranges from 64 MB to 492 MB. For SPHINX two sizes are used: 256 MB and 492 MB. Executing SPHINX on systems with less than 256 MB of memory leads to thrashing. In all graphs the straight vertical line represents the 16 second break-even point of the Hitachi hard disk. When executing on the standard Linux kernel (not shown in the graphs), the first three workloads lead to access patterns in which 100% of the disk idle time appears in intervals of less than 1 second, independent of memory size, preventing the use of any low-power mode. In contrast, larger memory sizes lead to longer idle interval lengths for the Bursty system, providing more opportunities for the disk to transition to a low-power mode. During the Linux kernel compilation,

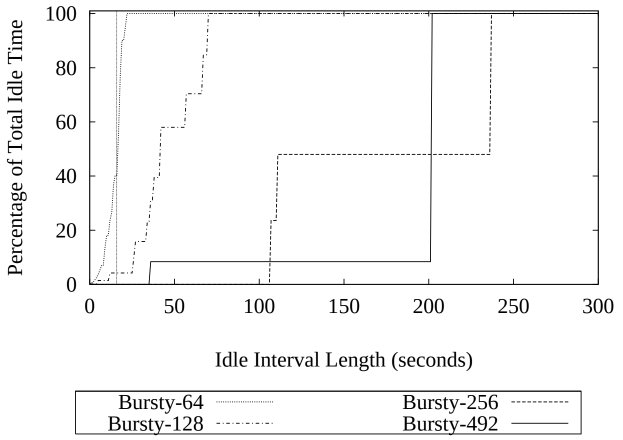

Figure 8: Cumulative distribution of disk idle time intervals during MPEG playback. On the standard Linux kernel (not shown), 100% of the disk idle time appears in intervals of less than 1 second for all memory sizes.

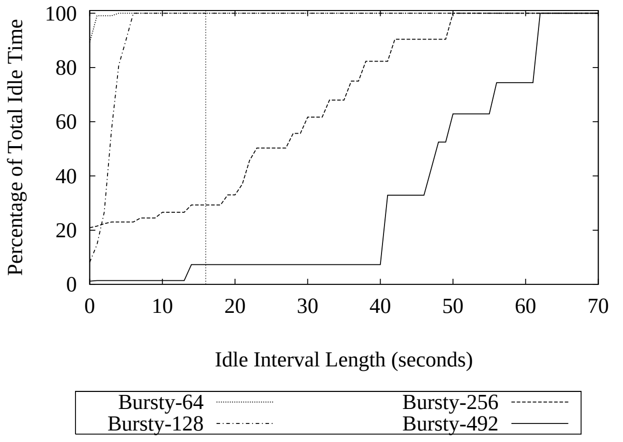

Figure 9: Distribution of disk idle time intervals during concurrent MPEG playback and MP3 encoding. On the standard Linux kernel (not shown), 100% of the disk idle time appears in intervals of less than 1 second for all memory sizes.

the Bursty system manages to prefetch most of the accessed data when system memory exceeds 128 MB. At 96 MB our energy-aware prefetching algorithm slowly increases the size of the prefetch cache, eventually achieving idle periods that are longer than the disk's breakeven time. The algorithm behaves similarly on a 64 MB system. However, it also leads to increased paging that has a negative effect on both energy and performance. For the speech recognition workload at 492 MB the Bursty system prefetched the whole language model file leading to long idle phases. At 256 MB it prefetched up to 33% of the file, leading to idle interval lengths only slightly longer

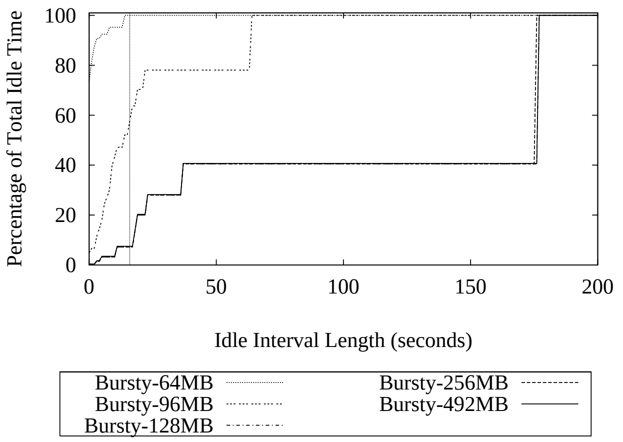

Figure 10: Distribution of disk idle time intervals during a full Linux kernel compilation. On the standard Linux kernel (not shown), 100% of the disk idle time appears in intervals of less than 1 second for all memory sizes. At a memory size of 128 MB and over all accessed files are prefetched by the Bursty system leading to increased idle interval lengths.

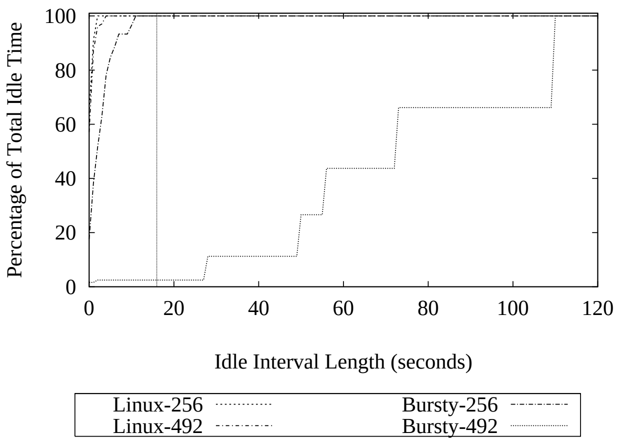

Figure 11: Distribution of disk idle time intervals during speech recognition by SPHINX. On the standard Linux kernel with 256 MB, 100% of the disk idle time appears in intervals of less than 1 second.

than Linux, due to accesses to the uncached portion of the file.

Figure 12 presents disk energy savings as a function of total system memory size. The base case used for the comparisons is the standard Linux kernel on a 64 MB system. For Linux, increasing the system's memory size has only a minor impact on the energy consumed by the disk, because of the lack of long idle intervals. In contrast, the savings achieved by the Bursty algorithm depend on the amount of memory available. For the first workload, sig-

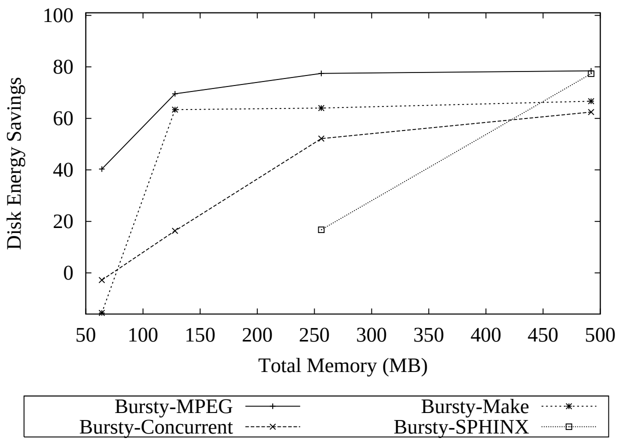

Figure 12: Disk energy savings as a function of total memory size. Results are shown for all experimental workloads: *MPEG* playback, *Concurrent* MPEG playback and MP3 encoding, *Make* (the Linux kernel compilation) and *SPHINX* executing on our Bursty system. When executing on the standard Linux kernel (not shown), increasing the total memory size to 492 MB leads to at most 5.25% in disk energy savings across all workloads.

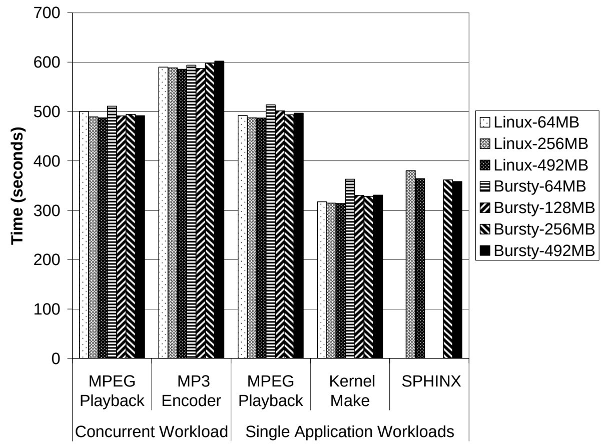

Figure 13: Execution time (in seconds) of the workloads on Linux and the Bursty system with various memory configurations.

nificant energy savings are achieved for all memory sizes. Even on the 64 MB system, the energy consumed by the disk is reduced by 40.3%. Despite the fact that most disk idle intervals are not long enough to justify a spin-down operation, they allow the disk to make efficient use of the low-power idle state that consumes just 0.6 W. With 492 MB, the Bursty system loads the required data in just three very intensive I/O bursts, allowing the disk to transition and remain in the spin-down state for significant periods of time, and leading to 78.5% disk energy savings.

Results for the second workload are similar. However, because of the increased I/O intensity the energy savings are less pronounced. Energy consumption is reduced after the memory size exceeds 128 MB (15.9% energy savings). On a system with 492 MB energy savings reach 62.5%. For the Linux kernel compilation, our Bursty system achieved significant energy savings for memory sizes of 128 MB and larger: up to 66.6%. However, despite the increase in idle interval lengths, on a 64 MB system our algorithm leads to increased energy consumption (15.5%) because of excessive paging. Finally, for the speech recognition workload, disk energy savings reach 77.4% for a 492 MB system. With 256 MB, the energy-conscious prefetching algorithm saves 16.7% through more efficient use of the active idle power mode.

Figure 13 presents the execution time for the workload scenarios. For the *Concurrent* workload (left side of the graph), the slowdown of the MP3 encoding process is 2.8% or less. The performance of the MPEG player stays within 1.6% of that on the Linux system in all cases except the 64 MB system (Bursty-64MB), where it experiences a slowdown of 4.8%. For the *MPEG* playback workload, the Bursty system experiences a slowdown of 1% (for the 64 MB case) or less, when compared to Linux with 492 MB of memory (Linux-492MB). For the Linux kernel compilation the Bursty system stays within 5% of the execution time of Linux across all memory sizes larger than 128 MB. On a 64 MB system Bursty experiences a performance penalty of 15% mostly due to increased paging and disk congestion. Using a priority-based disk queue that gives higher priority to demand requests than prefetching requests could lead to improved performance. Finally, during speech recognition aggressive prefetching of the language model file leads to slightly improved performance for *SPHINX* due to the reduction in page cache misses. Our performance results show that the prefetching algorithm manages to avoid successfully most of the delay that may be caused by disk spin-up operations. In addition, it can lead to improved performance because of an increased cache hit ratio.

6 Related Work

Power Management. The research community has been very active in the area of power-conscious systems during the last few years. Golding et al. [11] hint at the idea of conserving such non-renewable resources as battery power during idle periods by disabling unused resources. Ellis et al. [6] suggest making energy efficiency a primary metric in the design of operating systems. ECOSystem [37] provides a model for accounting and for fairly allocating the available energy among competing applications according to user preferences. Odyssey [10, 27] provides operating system support for application-aware resource management. The key idea is to trade quality for resource availability.

Several policies have been proposed to decrease the power consumption of processors that support dynamic voltage and frequency scaling. The key idea is to schedule so as to "squeeze out the idle time" in rate-based applications. Several researchers have proposed voltage schedulers for general purpose systems [9, 12, 35, 32]. Lebeck et al. [21] explore power-aware page allocation in order to make more efficient use of memory chips supporting multiple power states, such as the Rambus DRAM chips.

Hard Disks. The energy efficiency of hard disks is not a new topic. The cost and risks of Standby mode played a role in the early investigation of hard-disk spin-down policies [4, 5, 15, 22]. Concurrently with our own work [28], several groups have begun to investigate the deliberate generation of bursty access patterns. Heath et al. [14] and Weissel et al. [36] propose user-level mechanisms to increase the burstiness of I/O requests from individual applications. Lu et al. [24] report that significant energy can be saved by respecting the relationship between processes and devices in the CPU scheduling algorithm. (We note, however, that given the many-second "break-even" times for hard disks, process scheduling can increase burstiness only for non-interactive applications, which can tolerate very long quanta.) Zeng et al. [38] propose "shaping" the disk access pattern as part of a larger effort to make energy a first-class resource in the eyes of the operating system.

Like Lu et al. and Zeng et al., we believe that the effective management of devices with standby modes requires global knowledge, and must be implemented, at least in part, by the operating system. Our work differs from that of Lu et al. by focusing on aggressive read-ahead and write-behind policies that can lead to bursty device-level access patterns even for interactive applications. Our work is more similar to that of Zeng et al., but without the notion of energy as a first-class resource. While we agree that energy awareness should be integrated into all aspects of the operating system, it is not clear to us that it makes sense to allocate joules to processes in the same way we allocate cycles or bytes. Rather than say "I'd like to devote 20% of my battery life to MP3 playback and 40% to emacs," we suspect that users in an energy-constrained environment will say "I'd like to extend my battery life as long as possible without suffering more than a 20% drop in sound quality or interactive responsiveness." It will then be the responsibility of the operating system to manage energy *across applications* to meet these quality-of-service constraints.

Recently, researchers have begun to explore methods to reduce the power consumption of large-scale storage systems. Carrera et al. [7] compare design schemes for conserving disk energy in network servers. Colarelli et al. [2] explore massive arrays of idle disks, or MAID, as an alternative to conventional mass storage systems for scientific computing. Papathanasiou et al. [30] suggest replacing server-class disks with power-efficient arrays of laptop disks.

Disk access pattern "shaping" techniques, such as our prefetching algorithm, can be applied to server storage systems and improve their power efficiency. Zhu et al. [39] propose power-aware storage cache management algorithms that provide additional opportunities for a large-scale storage system to save energy. Our prefetching algorithm complements nicely their power-aware cache replacement policy. Finally, Gurumurthi et al. [13] suggest the use of DRPM [13], an approach that would dynamically modulate disk speed, decreasing the power required to keep the platters spinning when the load is light.

Prefetching. Prefetching has been suggested by several researchers as a method to decrease application perceived delays caused by the storage subsystem. Previous work has suggested the use of hints as a method to increase prefetching aggressiveness for workloads consisting of both single [31] and multiple [34] applications. Cao et al. [1] propose a two-level page replacement scheme that allows applications to control their own cache replacement, while the kernel controls the allocation of cache space among processes. Kaplan et al. [18] explore techniques to control the amount of memory dedicated to prefetching. Curewitz et al. [3] explore data compression techniques in order to increase the amount of prefetched data. To the best of our knowledge, previously proposed prefetching algorithms do not address improved energy efficiency. In general, they assume a non-congested disk subsystem, and they allow prefetching to proceed in a conservative way resulting in a relatively smooth disk usage pattern.

7 Conclusion

In our study we investigated page prefetching and caching strategies that increase the burstiness of I/O patterns in order to save energy in disks with non-operational low-power states. In addition, we presented methods to predict future accesses automatically and aggressively, to balance the memory requirements of prefetching and caching, and to coordinate accesses of concurrently running applications so that requests are generated and arrive at the disk at roughly the same time.

We have implemented our ideas in the Linux 2.4.20 kernel. Experiments with a variety of applications show that our techniques can increase the length of idle phases significantly compared to a standard Linux kernel leading to disk energy savings of 60–80%. The savings depend on the amount of available memory, and increase as the system's memory size increases. Even relatively short increases in the average idle interval length can lead to significant energy savings, mostly by making more efficient use of intermediate low-power states. Published studies [23, 5] attribute 9-32% of the total laptop energy consumption to the hard disk. These figures imply that our prefetching algorithm may increase battery lifetime by up to 25%. The exact fraction depends on the system configuration and the executing workload.

Though our current work has focused on hard disks, increased burstiness could be used to improve the energy efficiency of other devices with non-operational low-power states. Network interfaces—for wireless networks in particular—are an obvious example, but they introduce new complications. First, in addition to standby states, several wireless interfaces support multiple active states, with varying levels of broadcast power suitable for communication over varying distances. Second, while a disk never initiates communication, a wireless network does. Third, the energy consumed by a wireless interface depends on the quality of the channel, so communication bursts should be scheduled, when possible, during periods of high channel quality.

Over time, we speculate that burstiness may become important in the processor domain as well. In a processor with multiple clock domains, for example [33], one can save dynamic power in a floating-point application by slowing down the (lightly-used) integer unit. Alternatively, by scheduling instructions for burstiness, one might save both dynamic *and* static power by gating off voltage to the integer unit during periods of inactivity. This tradeoff between operational scale-down, with a smooth access pattern, and non-operational power-down, with a bursty access pattern, arises in multiple situations (including DRPM [13]), and is likely to be a recurring theme in our future work.

References

[1] CAO, P., FELTEN, E. W., AND LI, K. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the 1995 ACM Joint Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRCIS'95/PERFORMANCE'95)* (1995), pp. 188–197.

[2] COLARELLI, D., AND GRUNWALD, D. Massive Arrays of Idle Disks for Storage Archives. In *Proc. of the 2002 ACM/IEEE Conf. on Supercomputing (SC'02)* (Nov. 2002), pp. 1–11.

[3] CUREWITZ, K. M., KRISHNAN, P., AND VITTER, J. S. Practical Prefetching via Data Compression. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'93)* (May 1993), pp. 257–266.

[4] DOUGLIS, F., KRISHNAN, P., AND BERSHAD, B. Adaptive Disk Spin-down Policies for Mobile Computers. In *Proc. of the 2nd USENIX Symp. on Mobile and Location-Independent Computing* (Apr. 1995).

[5] DOUGLIS, F., KRISHNAN, P., AND MARSH, B. Thwarting the Power-Hungry Disk. In *Proc. of the 1994 Winter USENIX Conf.* (Jan. 1994), pp. 293–306.

[6] ELLIS, C. S. The Case for Higher Level Power Management. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems (HotOS VII)* (Mar. 1999).

[7] ENRIQUE V. CARRERA, EDUARDO PINHEIRO, R. B. Conserving Disk Energy in Network Servers. In *Proc. of the 17th Annual ACM Intl. Conf. on Supercomputing (ICS'03)* (June 2003), pp. 86–97.

[8] FERGUSON, G., AND ALLEN, J. TRIPS: An Intelligent Integrated Problem-Solving Assistant. In *Proc. of the 15th National Conf. on Artificial Intelligence (AAAI-98)* (July 1998), pp. 567–573.

[9] FLAUTNER, K., AND MUDGE, T. Vertigo: Automatic Performance-Setting for Linux. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation (OSDI'02)* (Dec. 2002), pp. 105–116.

[10] FLINN, J., AND SATYANARAYANAN, M. Energy-Aware Adaptation for Mobile Applications. In *Proc. of the 17th ACM Symp. on Operating Systems Principles* (Dec. 1999), pp. 48–63.

[11] GOLDING, R., II, P. B., STAELIN, C., SULLIVAN, T., AND WILKES, J. Idleness is not sloth, Jan. 1995.

[12] GOVIL, K., CHAN, E., AND WASSERMAN, H. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. In *Proc. of the 1st Annual Intl. Conf. on Mobile Computing and Networking (MobiCom'95)* (Nov. 1995).

[13] GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., AND FRANKE, H. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proc. of the 30th Intl. Symp. on Computer Architecture (ISCA'03)* (June 2003), ACM Press, pp. 169–181.

[14] HEATH, T., PINHEIRO, E., HOM, J., KREMER, U., AND BIANCHINI, R. Application Transformations for Energy and Performance-Aware Device Management. In *Proc. of the 11th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'02)* (Sept. 2002).

[15] HELMBOLD, D. P., LONG, D. D. E., AND SHERROD, B. A Dynamic Disk Spin-down Technique for Mobile Computing. In *Proc. of the 2nd Annual Intl. Conf. on Mobile Computing and Networking (MobiCom'96)* (Nov. 1996).

[16] HUANG, X., ALLEVA, F., HON, H.-W., HWANG, M.-Y., AND ROSENFELD, R. The SPHINX-II speech recognition system: an overview. *Computer Speech and Language 7*, 2 (1993), 137–148.

[17] IBM Corporation. OEM Hard Disk Drive Specifi cations for DARA-2xxxxx (6 GB – 25 GB). 2.5-Inch Hard Disk Drive with ATA Interface. Revision (2.1), Nov. 1999.

[18] KAPLAN, S. F., MCGEOCH, L. A., AND COLE, M. F. Adaptive Caching for Demand Prepaging. In *Proceedings of the 3rd Intl. Symp. on Memory Management* (2002), ACM Press, pp. 114–126.

[19] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[20] KUENNING, G. H., AND POPEK, G. J. Automated Hoarding for Mobile Computers. In *Proc. of the 16th ACM Symp. on Operating Systems Principles* (Oct. 1997), ACM Press, pp. 264–275.

[21] LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. S. Power Aware Page Allocation. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)* (Nov. 2000), pp. 105–116.

[22] LI, K., KUMPF, R., HORTON, P., AND ANDERSON, T. Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proc. of the 1994 Winter USENIX Conf.* (Jan. 1994), pp. 279–291.

[23] LORCH, J. R., AND SMITH, A. J. Energy consumption of Apple Macintosh computers. *IEEE Micro 18*, 6 (Nov. 1998), 54–63.

[24] LU, Y.-H., BENINI, L., AND MICHELI, G. D. Power-Aware Operating Systems for Interactive Systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems 10*, 1 (Apr. 2002).

[25] Micron Technology Inc. Micron 256Mb and 512Mb: x16 TwinDie SDRAM (Revision A 5/03 EN), May 2003.

[26] MOGUL, J. C. A Better Update Policy. In *Proc. of the USENIX Summer 1994 Technical Conf.* (June 1994).

[27] NOBLE, B., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile Application-Aware Adaptation for Mobility. In *Proc. of the 16th ACM Symp. on Operating Systems Principles* (Oct. 1997).

[28] PAPATHANASIOU, A. E., AND SCOTT, M. L. Increasing Disk Burstiness for Energy Effi ciency. Tech. Rep. 792, Computer Science Departmeny, University of Rochester, Nov. 2002.

[29] PAPATHANASIOU, A. E., AND SCOTT, M. L. Energy Effi ciency Through Burstiness. In *Proc. of the 5th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'03)* (Oct. 2003), pp. 44–53.

[30] PAPATHANASIOU, A. E., AND SCOTT, M. L. Power-effi cient Server-class Performance from Arrays of Laptop Disks. Tech. Rep. 837, Computer Science Departmeny, University of Rochester, Apr. 2004.

[31] PATTERSON, R. H., GIBSON, G., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed Prefetching and Caching. In *Proc. of the 15th ACM Symp. on Operating Systems Principles* (Dec. 1995), pp. 79–95.

[32] PERING, T., BURD, T., AND BRODERSEN, R. Voltage Scheduling in the lpARM Microprocessor System. In *Proc. of the 2000 Intl. Symp. on Low Power Electronics and Design (ISLPED'00)* (July 2000), pp. 96–101.

[33] SEMERARO, G., MAGKLIS, G., BALASUBRAMONIAN, R., ALBONESI, D. H., DWARKADAS, S., AND SCOTT, M. L. Energy Effi cient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *Proc. of the 8th Intl. Symp. on High Performance Computer Architecture (HPCA-8)* (Feb. 2002), pp. 29–40.

[34] TOMKINS, A., PATTERSON, R. H., AND GIBSON, G. Informed multi-process prefetching and caching. In *Proc. of the 1997 ACM Joint Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRCIS'97)* (1997), ACM Press, pp. 100–114.

[35] WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. Scheduling for Reduced CPU Energy. In *Proc. of the 1st USENIX Symp. on Operating Systems Design and Implementation (OSDI'94)* (Nov. 1994).

[36] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O: A Novel I/O Semantics for Energy-Aware Applications. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation (OSDI'02)* (Dec. 2002).

[37] ZENG, H., FAN, X., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)* (Oct. 2002).

[38] ZENG, H., FAN, X., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. Currentcy: Unifying Policies for Resource Management. In *Proc. of the USENIX 2003 Annual Technical Conf.* (June 2003).

[39] ZHU, Q., DAVID, F., ZHOU, Y., DEVARAJ, C., AND CAO, P. Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management. In *Proc. of the 10th Intl. Symp. on High Performance Computer Architecture (HPCA-10)* (Feb. 2004).