Nov. 2002

Increasing Disk Burstiness for Energy Efficiency

Athanasios E. Papathanasiou and Michael L. Scott

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 792

Abstract

Hard disks for portable devices, and the operating systems that manage them, incorporate *spin-down* policies that idle the disk after a certain period of inactivity. In essence, these policies use a recent period of inactivity to predict that the disk will remain inactive in the near future. We propose an alternative strategy, in which the operating system deliberately seeks to *cluster* disk operations in time, to maximize the utilization of the disk when it is spun up and the time that the disk can be spun down. In order to cluster disk operations we postpone the service of non-urgent operations, and use aggressive prefetching and file prediction to reduce the likelihood that synchronous reads will have to go to disk. In addition, we present a novel predictive spin-down/spin-up policy that exploits high level operating system knowledge to decrease disk idle time prior to spin-down, and application wait time due to spin-up. We evaluate our strategy through trace-driven simulation of several different workload scenarios. Our results indicate that the deliberate creation of bursty activity can save up to 55% of the energy consumed by an IBM TravelStar disk, while simultaneously decreasing significantly the negative impact of disk spin-up latency on application performance.

This work was supported in part by NSF grants CCR–9988361, CCR–0204344, and EIA–0080124, and by DARPA/ITO under AFRL contract F29601-00-K-0182.

1 Introduction

Power efficiency has become a major concern for computing systems. Processors such as the Transmeta Crusoe [Halfhill, 2000a; Halfhill, 2000b] and the Intel StrongARM [Intel Corporation] support frequency and voltage scaling, allowing a clever scheduling algorithm to save energy by "squeezing out the idle time" in rate-based applications. Memory, hard disk, and networking devices typically support a small number of discrete states, with varying power consumption and performance characteristics. The device typically operates only in the highest energy state; other states consume progressively less power, but take increasing amounts of time, and often energy, to return to the active state.

Traditional power management policies attempt to move a device into one of its low power states during periods of idle time. The principal challenge is to identify the state that will maximize energy savings for a given level of acceptable performance degradation. Since the optimal power state depends not only on the device's characteristics, but also on the length of the idle period, which is not known in advance, choosing the optimal state becomes a difficult prediction problem. The most common policies employ a fixed or adaptive *threshold*, moving to a low power state after a certain period of device inactivity. More complex policies monitor the utilization pattern of the underlying device, keep track of the length and frequency of idle periods, and attempt to select a power mode based on prediction of the duration of the upcoming idle period.

Unfortunately, power management policies that simply *react* to observed access patterns suffer from a fundamental limitation: if the time between device accesses is too small to justify moving to a low power mode, then the most efficient reactive policy will keep the device constantly active, and no energy will be saved. A more promising approach, we believe, is to *change* the access pattern. Specifically, we propose that the operating system deliberately attempts to create *bursty* access patterns for devices with low power modes. In this paper we focus in particular on hard disks.

Though most disk requests are driven by applications, the temporal distribution of those requests is to a large extent an artifact of the scheduling, memory management, and buffering decisions made by the OS kernel. Operating systems have traditionally attempted to generate as smooth an access pattern as possible, by spreading requests over time. This smoothing reduces the likelihood of I/O bottlenecks, and can be expected to maximize throughput for I/O intensive workloads, but it hides opportunities to save energy. If the operating system attempted instead to cluster requests in time it might often be able to merge several short intervals of inactivity into a single, longer interval, during which the disk could profitably be moved to a low power ("spun down") state.

In the remainder of the paper, we present and evaluate mechanisms to increase the burstiness of disk usage patterns, by delaying asynchronous read and write requests (moving them forward in time) and by prefetching aggressively to reduce as much as possible the number of synchronous reads that have to come from disk (moving them backward in time). We focus in our current work on interactive workloads and workloads consisting of rate-based and multimedia applications. We also present preliminary results that quantify the positive impact of file prediction on energy efficiency. Ongoing work (not reported here) will accommodate more random and interactive access patterns, using more sophisticated prefetching mechanisms, and spooling synchronous writes through FLASH RAM. To quantify the benefits of burstiness, we model the energy consumed by a standard laptop hard disk, driven by disk access patterns from multimedia applications.

To identify opportunities to move requests in time, we argue that a power management system must track the access patterns of applications over time, generating application-specific predictions of future accesses [Patterson *et al.*, 1995], and must integrate the power state and performance characteristics of

underlying devices into such traditional resource management policies as memory management and disk scheduling.

For mobile systems, whose workloads are not usually I/O intensive, the principal downside to a bursty access pattern is a potential increase in the application-perceived latency of synchronous reads, due to spin-up operations. For interactive applications aggressive prefetching serves to reduce the number of visible delays. 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. It can also be used deactivate the disk after the end of a burst of activity.

Data usage prediction and prefetching have been studied extensively in the past as a method to improve system performance. The nature of power management, however, suggests a significantly more aggressive approach. When the goal is simply to minimize application perceived latency, the benefits of prefetching are limited by the application's data consumption rate [Patterson *et al.*, 1995]: it is useless to prefetch earlier than the point at which a future access will yield a zero stall time. With respect to power management, however, any future data access that does not hit in memory is equally costly after the device has entered a low power state, since it will lead to a spin-up operation. To maximize energy efficiency we should prefetch as aggressively as possible, as long as prefetching does not exceed our memory capacity, or evict data that will be used sooner than the prefetched data.

The rest of this paper is structured as follows. Section 2 provides the motivation for an energy-conscious file system, while Section 3 describes the design of our prefetching and request deferring mechanisms. Section 4 presents experimental results. In a series of experiments we report on (1) delaying asynchronous requests for mixed workloads, (2) prefetching for applications with sequential access patterns, and (3) prefetching in more general applications. Our results indicate that the deliberate creation of bursty activity can save up to 55% of the energy consumed by an IBM TravelStar disk, while simultaneously decreasing significantly the negative impact of disk spin-up latency on application performance. Section 5 discusses previous work. Section 6 summarizes our conclusions.

2 Motivation

2.1 Energy Efficiency of Hard Disks

Following the guidelines of the Advanced Configuration and Power Interface Specification (ACPI) [ACPI, 2000], modern hard disks for mobile systems support four different power states: Active, Idle, Standby, and Sleep. 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 [IBM, 1999].

The time and energy required to move from Idle to Active state are minimal for most devices. We therefore assume in the rest of this paper that the disk moves automatically to Idle state when there are no pending requests. The time and energy required to move from Sleep to any other state are high enough that we assume that state is used only when the machine is shut down or has been idle for a significant period of time. Standby state lies in the middle. Contemporary disks require on the order of one to three seconds to transition from Standby to Active state. During that spin-up time they consume 1.5–2X as much power as they do when Active. The typical laptop disk must therefore remain in Standby state for a certain amount of

Disk	M-1994	IBM-2000	T-2001	IBM-micro
Capacity	105MB	6-18GB	2GB	340MB-1GB
Active	1.95W	2.1W	1.3W	0.73W
Idle	1.0W	1.85W	0.7W	0.5W
Active Idle	NA	0.85W	NA	NA
Low Power Idle	NA	0.65W	0.5W	0.22W
Standby	0.025W	0.25W	0.23W	0.066W
Spin up	3.0W	3.33W	3.0W	0.66W
Spin down	0.025W	NA	NA	NA
Spin up time	2.0s	1.8s	1.2s	0.5s
Spin down time	1.0s	NA	NA	NA
Breakeven time	6.2s	6.2s	7.6–13.3s	0.7–2s

Table 1: Energy consumption parameters for various disks. M-1994 stands for MAXTOR MXL-105 III (1994), IBM-200 stands for IBM Travelstar (2000), T-2001 stands for Toshiba MK5002 MPL (2001), and IBM-micro stands for IBM Microdrive DSCM.

time to justify the energy cost of the subsequent spin-up. This *break-even time* is currently on the order of 5–15 seconds for laptop disks. Table 1 presents the break-even time and the power characteristics for one older and three more recent hard disks. Note that the IBM Microdrive, a one-inch device intended for use in cameras and PDAs, is smaller than the typical laptop disk.

The principal task of an energy-conscious disk management policy is to force transitions to Standby state when this is likely to save significant energy. Inappropriate spin-downs can waste energy (if the disk remains idle for less than the break-even time), frustrate the human user (if the computer must frequently spin up the disk in order to service an interactive application), and reduce the lifetime of the disk through wear and tear. On the other hand, even significant amounts of unnecessary disk traffic (e.g. prefetching of data that are never actually used) can be justified if they allow us to avoid spin-up operations, or to leave the disk spun down for longer periods of time. Our goals are thus to (1) *maximize the length of idle phases* by prefetching aggressively and by postponing write requests, (2) *minimize wait time* in the Idle state during idle phases long enough to justify spin down, (3) *avoid unnecessary spin downs* during short idle phases, and (4) *minimize application perceived delays* through disk pre-activation.

2.2 Linux Delayed-Write and Prefetching Algorithms

As a concrete example of disk management policies oriented toward smoothing, consider the delayed-write and prefetching algorithms employed in Linux. Like many modern operating systems, Linux uses a unified virtual memory and file buffer system to cache disk contents in memory. Write operations create dirty pages in memory, but unless the application explicitly requests a synchronous operation, they do not immediately access disk. Rather, a kernel daemon called *Kupdate* executes every five seconds and initiates write requests for every buffer that has been dirty for more than thirty seconds. The 30 second delay decreases the system's write activity: files that are deleted within 30 seconds never actually reach the disk. The 5 second repeat interval serves to spread disk traffic over time, and ensures that a system crash can never destroy data more than 35 seconds old. (In a non-preemptive kernel such as Linux, frequent execution of *Kupdate* also prevents the work associated with creating disk requests from becoming a noticeable "hiccup" in user responsiveness. In our energy-aware version of Linux, the daemon still runs frequently, but we postpone the activation time of the disk request queue.)

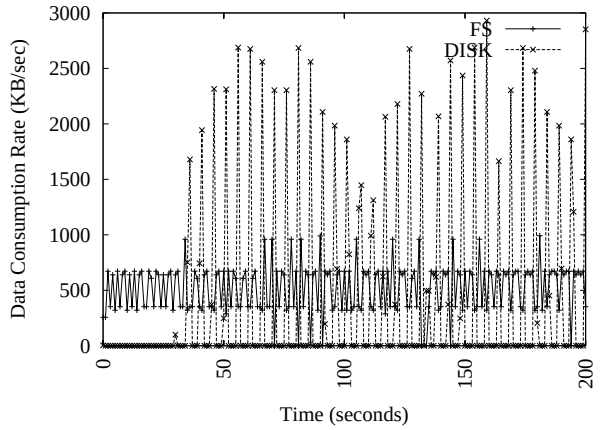

Figure 1: Data production rate as seen by the Linux disk and the file system during a CD copy operation. Differences are due to the behavior of *Kupdate*.

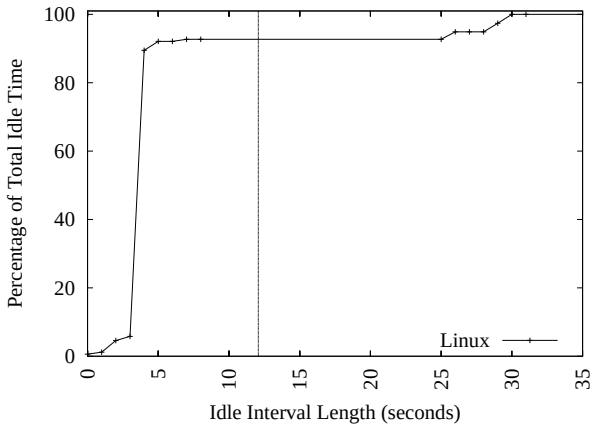

Figure 2: Distribution of idle time intervals for the Linux disk during a CD copy operation.

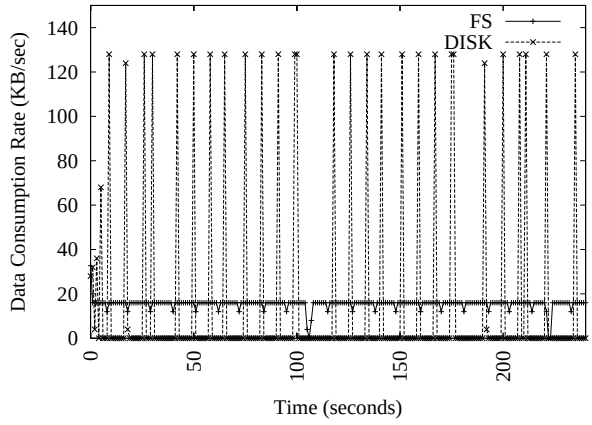

Figure 3: Data consumption rate as seen by the Linux disk and file system during mp3 playback. Differences are due to prefetching.

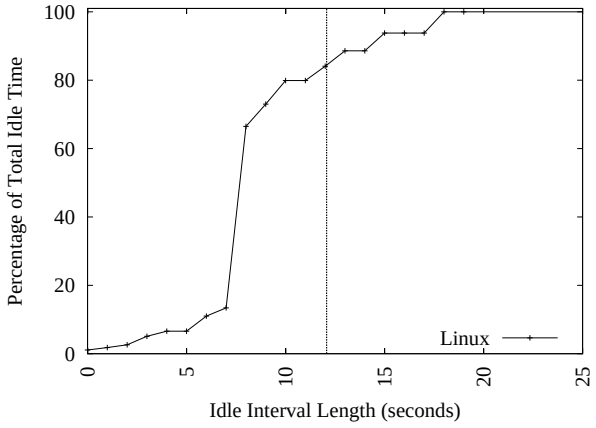

Figure 4: Distribution of idle time intervals for the Linux disk during mp3 playback.

Kupdate has a negative effect on energy consumption. Unless the system is completely idle, write requests are generated almost every time the daemon runs. The result can be seen in Figures 1 and 2, which show data production rates and disk idle times while copying a CD-ROM to disk. No idle time interval is longer than five seconds, leaving no opportunity to save energy by spinning down the disk. This is unfortunate given that the sustainable bandwidth of the disk significantly exceeds that of the CD drive.

For sequential read accesses, Linux supports a conservative prefetching algorithm that reads up to 128 KB (32 4KB pages) in advance. Figures 3 and 4 illustrate the resulting disk behavior for mp3 playback. The application consumes file data sequentially at a rate of approximately 16 KB/s (1MB/min), which the kernel translates into read requests for 128KB every 8 seconds. Since most modern laptop disks have a break-even time greater than 8 seconds, there is again no opportunity to save energy by spinning down the disk. In our experiment 66% of the total disk idle time (194 seconds out of 292) appears in intervals of less than eight seconds, and only 15% appears in intervals that are larger than 12 seconds (the break-even time for an IBM TravelStar disk).

Assuming there is enough available memory, an energy conscious prefetching algorithm could increase the amount of prefetched data. Doubling it to 256KB would move the inter-request interval above the break-even point for spin-down of many laptop disks. Additional increases would permit substantially greater energy savings. Moreover, a disk scheduling algorithm that has access to the additional information that the mp3 playback is the only currently active task can spin down the disk immediately after servicing the last request and spin it up again just in time to service the next bunch of prefetching requests.

The previous examples illustrate cases where smoothing conflicts with energy efficiency. They suggest the following guidelines for energy efficient design:

1. **Delayed updates**: Update policies should be modified in order to increase the burstiness of write activity. A simple way to increase write burstiness would be to increase the execution period of the kernel's update thread, but such a change could lead to decreased system responsiveness.

2. **Aggressive Prefetching**: Depth of data prefetching should be increased beyond an application's prefetch horizon in order to reduce the frequency with which read requests have to be serviced by the disk. File prediction should also be employed. The risks of aggressive prefetching may be decreased by the use of hints. Hints may be generated automatically by keeping track of past application access patterns.

3. **Urgency-based scheduling**: Increased burstiness can lead to disk congestion. To avoid increased application perceived delays and reduced system reliability, urgent requests (mostly synchronous reads associated with foreground applications and synchronous writes) should not wait for the completion of earlier non-urgent requests. To make urgency information available to the low-level disk driver, requests should be annotated when originally generated in higher levels of the system.

4. **Device Awareness**: Power consumption and performance characteristics of devices should be integrated into the operating system algorithms that determine the device's usage pattern. In the case of hard disks, the memory management system—specifically the prefetching and update algorithms—should reflect the break-even times for each low power state.

3 Design of an Energy-Conscious File System

Following the guidelines suggested in Section 2, we propose the following kernel subsystems for energy-conscious disk management (Figure 5):

- The **monitoring system** tracks file accesses of applications over time (including multiple runs), to learn their access patterns. It notes which files are read and written, at what rates, and with what recurring patterns (sequential or random access, once or multiple times).

- Using information from the monitoring system, the **hint generation system** predicts future disk accesses on the part of currently running applications, and passes these predictions as hints to the memory management and file system. A similar approach to prefetching was proposed by Patterson *et al.* [Patterson *et al.*, 1995], but in a less aggressive form. As noted in section 1, attempts to maximize performance do not lead to the depth of prefetching required to produce a bursty disk usage pattern.

- Based on the currently available memory and information from the hinting system, the **memory management system** and **file system** must decide which data buffers to keep in memory in order to maximize the length of idle phases. To accomplish these tasks, they are augmented with algorithms for

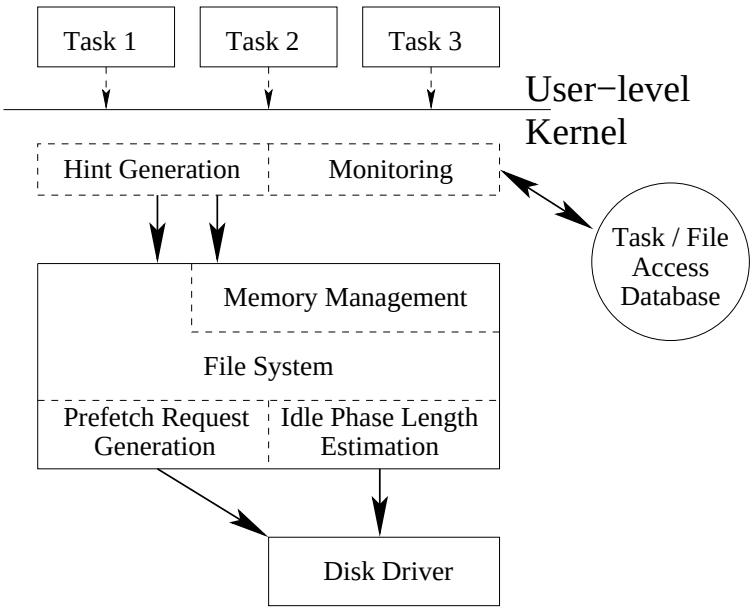

Figure 5: Design of the Proposed System.

prefetch request generation and **idle phase length estimation**. During each brief period of disk activity, these algorithms determine which memory pages are going to be accessed in the next idle phase, which files and how many buffers from each file have to be prefetched, which dirty pages have to be flushed in order to provide space for new data and meet application expectations regarding synchronous writes, and how many pages have to be reserved in order to hold the data that will be produced during the next idle phase. All requests are annotated with an indication of their urgency.

- Based on information from the prefetch request generation and idle phase length estimation algorithms, the **disk driver** decides whether to keep the device active or place it in a low power state. Non-urgent requests that are issued after a disk deactivation do not lead to an immediate disk reactivation.

3.1 Annotations for Delaying Requests

As a first step towards the implementation of the proposed design we present a scheme for annotating I/O requests in the Linux kernel. As mentioned above, request annotations can facilitate an efficient reshaping of the disk usage pattern and avoid significant performance degradation due to disk congestion. Particularly for applications with unpredictable access patterns, where prefetching cannot be used to increase disk burstiness, delaying selectively requests can avoid undesirable disk reactivations.

The decision to delay the service time of a synchronous request and the amount of time for which it can be delayed depends on its criticality or urgency and on the current status of the system.[1] Such information is available in the higher levels of the operating system where the request's origin and the causes that lead to the request are known, but it is lost before it reaches the level at which power-management policies are typically implemented. The disk driver may know the type of the operation (read or write), the size of the

[1] As an example of the effect of the system's status on request criticality, consider write requests that are generated during periods of low memory availability. Such requests are more urgent than similar requests generated during periods of low memory demand.

data, and the blocks of the disk that have to be written, but it does not know the purpose of the request or the identity of the process that initiated it. This information is unimportant to a pure threshold-based spin-down policy, but it may help us to distinguish urgent operations or operations that should not be delayed significantly from less important requests.

An inventory of I/O operations in the Linux 2.4 kernel suggests seven different categories of requests, listed here in increasing order of urgency:

1. Asynchronous requests from user-level processes.[2]
2. Synchronous requests from non-interactive user-level processes.
3. Asynchronous requests from kernel threads.
4. Synchronous requests from interactive user-level processes.
5. Synchronous requests from kernel threads.
6. Requests created by a *sync* operation.
7. Requests that have to be serviced in order to free system resources under heavy load.

Because the Linux scheduler does not attempt to distinguish between interactive and non-interactive processes, our initial implementation is unable to distinguish between requests in categories (2) and (4). We consider delaying requests in categories (1) and (3) ("non-urgent requests"), but service requests in categories (2) and (4)–(7) ("urgent requests") immediately. More specifically, our implementation employs two parameters: the spin-down threshold T and the maximum delay time D. We spin the disk down after T seconds in which no requests are pending. We spin the disk up when there is an urgent request, or when a D seconds have elapsed since the oldest still-unserviced non-urgent request. Once the disk is spun up, we service all outstanding requests, from all categories, and continue to service newly arriving requests (from all categories) until we again exceed the spin-down threshold T.

It is important to note that the generation of requests is independent of the time that the low-level driver will service the request. Synchronous requests are generated in direct response to an application system call. Asynchronous requests are generated by upper levels of the kernel periodically or when the processor is idle. Additional requests are generated by various modules of the kernel itself. The initiation of request servicing depends on the load of the hard disk and the power management policy. Request annotations keep the generation of requests independent of their actual service time and provide enough information to the low-level driver to respect the request's semantics.

The Linux I/O Mechanisms

I/O requests in Linux may be initiated through either the memory management system or the file system. Independent of the source, all I/O requests end up in a call to the function *generic_make_request()* in *drivers/block/ll_rw_blk.c*. The function inserts the request into a private queue associated with the device. After the insertion into the task queue has completed, the bottom half of the device driver visits the queue and services every pending request until the queue becomes empty.

[2]Currently, Linux does not support asynchronous I/O by user-level processes. All user level read requests will block until the requested data have been loaded into memory. Write system calls are effectively asynchronous: they return calls return as soon as the request has been placed in the driver's queue.

Linux supports deferred processing through the notion of a *task queue.* Specifically, requests may be placed in a request queue without invoking the bottom half of the associated device driver. The bottom half of the driver may be activated at a later time. The function *run_task_queue()* is used in order to activate the bottom halves of all the drivers that have requests associated with them in the task queue. An example in which deferred processing is used is delaying I/O requests to the hard disk so that they may be merged or re-ordered with new requests.

Deferred processing is supported by the kernel through a mechanism called *plugging*. Any device driver that supports deferred processing registers a *plug* function with the kernel at initialization time. The *plug* function is responsible for inserting the request queue of the device driver into the task queue, and delaying arriving requests until a call to *run_task_queue()* takes place. A *plug* function is always associated with a corresponding *unplug* function, which is responsible for activating the bottom half of the driver, and removing its request queue from the task queue.

After initialization, *plugging* works as follows. Upon the arrival of a request to a driver with an empty request queue, its request queue is plugged into the task queue. Additional requests are stored in the plugged request queue, but no processing takes place. A call to the function *run_task_queue()* by some component of the system has as a result a call to the associated *unplug* function and hence the activation of the bottom half of the driver. Processing of the requests is initiated and new requests are placed into the request queue until serviced by the bottom half of the driver. When a driver's queue becomes empty the bottom half of the driver is deactivated. A new request will result in re-plugging the request queue of the driver.

Plugging for Power Savings

The plugging mechanism of Linux may be used in order to implement delayed processing of disk requests. The key idea behind the implementation is to substitute the default *unplug* routine associated with the disk driver with a new routine, called *power_saving_unplug_device()*, which unplugs its request queue and activates the bottom half of the driver only if a request that has to be processed immediately has entered the request queue or the hard disk has not yet spun down. Using this method, requests directed to a spun-down disk may be delayed until it becomes urgent to service them.

Since all requests directed to a specific device are placed into the same request queue, and since once the hard disk is spun up there is no reason to delay other pending requests, it is not necessary to keep importance information per request. Instead the request queue may keep track of the importance of the most urgent request currently in the queue. Consequently, the request queue structure is augmented with two additional fields. The first one, called *ps_unplug_priority* shows the current level of importance (priority) of the request queue, and takes any of the values presented near the beginning of this section. The second field, named *ps_queue_max_delay* indicates the time at which the request queue should be unplugged, and hence determines the maximum time for which the processing of the request queue may be delayed.

Insertion into a device's request queue takes place at the lowest level of the block device driver, where information related to the urgency of the request is not known. A method that could be used to make this information available at the low level driver implementation would be to make the urgency of the request a parameter in all functions found in the control flow path from the function that generates the request (for example the implementation of the *write()* system call) to the routine that inserts requests into a device request queue. However, such an implementation would require modifying a significant number of kernel, file system and driver functions, which makes this solution undesirable.

The fact that the request insertion routine runs in the process context that generated the request provides a more elegant alternative solution. The task structure of a process or thread is used to communicate the information about the urgency of a request to the low level driver implementation. For this purpose the

task structure is augmented with two fields. The field *ps_unplug_priority* is a pair that indicates if and for how long the current request may be delayed. An additional field, named *ps_unplug_priority_init*, provides default values for this pair, and is used to re-initialize it after the request has been inserted into the request queue. Note that a field describing the time period for which a request may be delayed is not necessary, since in our current implementation requests of the same urgency have the same maximum delay period.

Similar annotations are used for the *run_task_queue()* function, which unplugs a device request queue. Such annotations are necessary because in certain situations, such as cases of limited resource availability or a call to the *sync()* system call, the *run_task_queue()* function is invoked in order to unplug devices, force the processing of any pending I/O and hence reclaim resources immediately. In order to avoid delaying the processing of any pending requests in such cases, the annotations in the task structure are updated to indicate the increased importance of the new request. The unplug function *power_saving_unplug_device()* checks the urgency annotations of the current process. If those annotations indicate that the process's current request is urgent, the queue is unplugged and the bottom half of the driver is activated regardless of annotations already in the request queue.

3.2 Prefetching for Applications with Sequential Access Patterns

In an initial attempt to evaluate the benefits of aggressive prefetching, we have focused on I/O intensive applications with a more or less steady rate of sequential accesses and no reuse. Such applications include mp3 and mpeg playback and encoding, compression and decompression programs, and data copy applications (ftp, cd copying). These applications make it easy to generate hints that predict future I/O requests. Hint generation for more complex applications is a subject of future work (see section 3.3). We believe it to be feasible, particularly given the numbers involved: as noted in section 1, an algorithm that prefetches large amounts of extraneous data can still save energy if it avoids expensive spin-ups. In this respect, energy-conscious disk prefetching more closely resembles prefetching for disconnected operation in remote file systems [Kistler and Satyanarayanan, 1992] than it does data prefetching in processors.

Knowing in advance that an application accesses files only once allows the memory management system to evict pages immediately after use, rather than evicting the more useful, but less recently used, data that might be chosen by a traditional LRU replacement policy. In addition, knowing that an application accesses files sequentially allows us to prefetch more aggressively, since it is almost certain that prefetched data are going to be used. The prefetch horizon is constrained only by the available memory. As long as prefetching does not cause additional disk activity, because of swapping or paging, it will have a beneficial effect both on latency and energy consumption. For the rest of this description we assume that a portion of physical memory of known size is used to service the requests of all applications under concern. In Section 4 we present results for various memory sizes.

Our disk management algorithm operates in two phases. During a *request generation phase* the disk is activated and prefetching requests for each active file and write requests for dirty pages are generated. The amount of prefetched data for each active file is determined by the data consumption rate of the application accessing the file. Intuitively, a larger amount of memory should be reserved for prefetching the data of applications that exhibit a higher data consumption rate, so that all applications run out of prefetched data at about the same time. A certain portion of memory is also reserved for data that will be produced during the upcoming idle phase.

Upon completion of a request generation phase, an *idle phase* begins. During an idle phase a prediction algorithm predicts the length of the idle phase. The prediction algorithm executes periodically in order to update the data rates associated with each active file and to recalculate the predicted idle phase length based

on the new rates. If at any point during the idle phase the predicted length is larger than the disk's break-even time, then the disk is spun down. Moreover, after a disk spin down the prediction algorithm keeps recalculating the idle phase length in order to pre-activate the disk and initiate the next request generation phase, so that applications do not experience delay due to the spin-up time overhead.

During each request generation phase, the prefetching algorithm:

1. Computes the data production or consumption rate for each open file for which a sequential access pattern has been detected.
2. Calculates the aggregate rate imposed by all files under consideration.
3. Sets a target value for the length of the upcoming *idle phase* based on the aggregate data rate and the available memory.
4. Initiates prefetching for files associated with read activity for a total size equal to the file's data consumption rate multiplied by the predicted duration of the upcoming idle phase. For files associated with write activity, no additional accesses are necessary. Dirty pages produced during the idle phase will be stored in memory and flushed to the disk at the beginning of the next request generation phase.

The *idle phase length prediction algorithm* is similar to the prefetching algorithm. When invoked during the active phase it:

1. Updates the data production or consumption rate of each active file.
2. Computes the cumulative data production rate of all files associated with write activity. Based on this rate it calculates the time at which all free memory resources will be consumed (and hence a request generation phase will be initiated in order to free memory pages).
3. For all files associated with read activity, computes the minimum time at which the first access for which the data are not in memory will take place, based on each file's data consumption rate and the amount of data that has been prefetched.
4. Sets the prediction of idle phase length to the minimum of the values calculated in steps 2 and 3.

As mentioned above the prediction algorithm attempts to initiate a request generation phase before an application accesses data that have not been fetched in memory in order to avoid performance degradation. Two main causes can lead to decreased performance. First, the reactivation of a spun-down disk is associated with a significant time overhead. Second, our algorithm for increasing the burstiness of the disk usage pattern can lead to increased disk congestion. Consequently, applications associated with read requests that are placed at the end of a heavily loaded disk request queue suffer increased delay. In order to avoid such delays, the prediction algorithm generates prefetching requests before the last in-memory page of a file is consumed. In order to calculate the file access at which prefetching should start, the following equation is used:

$$P = RA_{end} - R * (T_{act} + \frac{PendingPages}{Throughput}) \quad (1)$$

where P is the file page at which prefetching has to be initiated in order to avoid any application perceived delays, RA_{end} and R represent the last in-memory page and the data consumption rate respectively for the file being accessed, T_{act} is the activation time for the file being checked, $PendingPages$ denotes the number of pages for which disk requests are pending, and $Throughput$ represents the average disk throughput in pages/second. To minimize the likelihood that any application will run out of data before any more has been prefetched, read requests for multiple files are prioritized, and data are read first for files that are most likely to run out.

3.3 File Prediction

Aggressive prefetching within a single file can increase disk file system burstiness in cases where relatively large files are being accessed. However, many applications access multiple small files. Such applications include compilers, computer games and editors. For these applications a more efficient re-shaping of the disk usage pattern may be achieved through file prediction.

Currently, we do not have a complete solution that solves the problem of accurate file prediction. However, we believe that efficient file prediction may be achieved by monitoring past file accesses of applications. Upon initiation of the execution of an application its working set of files can be loaded into memory. An alternative method is to aggressively load into memory all small files found in the working directory of the application. In Section 4.3, we present the energy savings that can be achieved by the latter file prediction method during a game playing workload scenario.

4 Experimental Evaluation

This section evaluates the benefits of using the proposed algorithms. In three sets of experiments we present:

1. The effect of delaying the service time of asynchronous write requests on energy consumption (Section 4.1).

2. The effect of aggressive prefetching and postponing writes for workloads consisting of multimedia and rate-based applications (Section 4.2).

3. The impact of file prediction and prefetching on the disk energy consumption of applications accessing multiple small files (Section 4.3).

We assume that a fixed threshold policy (spin down after a fixed number of seconds of idle time) is used as the disk power management policy. In the first and third sets of experiments, we use a large range of spin-down thresholds, ranging from zero seconds (immediate spin-down if there are no pending disk requests) to seven minutes. In the second set, we focus on the most aggressive thresholds, in the range of one to ten seconds. Larger threshold policies lead to no or very small energy savings for the workloads we consider. In all cases, the base case for comparison is the *No-spin-down* policy, in which the disk never spins down.

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.[3]

Energy Consumption We compare the energy savings achieved by all techniques against the No-spin-down policy.

Number and Type of Spin-Down Operations Spin-up operations have a negative effect on a disk's lifetime. In addition, if a spin-down operation results in spinning up the disk before the break-even time for the disk has elapsed, the operation leads to increased energy consumption. Hence, it is important to minimize the number of unnecessary spin-down operations.

State	Energy(J)	Power(J)	Latency
Active	NA	2.1W	NA
Idle	NA	0.65W	NA
Standby	NA	0.25W	NA
Sleep	NA	0.1W	2.8s
Spin up	6	3.33W	1.8s
Spin down	0	0	0s
Breakeven	NA	NA	12.0s

Table 2: Disk Model Characteristics. Transitions between the Active state and the Idle state are assumed to be instantaneous and do not require any additional energy consumption. The break-even time for transitioning to the standby state is 12.08 seconds.

In order to evaluate our disk management policies, we use traces that contain information from both the file system and the disk driver. For each file system or disk request the trace records the request generation time, the completion time, the type (read or write), the generating process, and the name of the relevant file. Traces were collected during actual execution of our workloads. The execution images of the applications were loaded into memory before the initiation of the trace collection process, in order to avoid disk activity due to page faults. In addition, we made sure than no other disk activity was generated during the trace collection as a result of paging or swapping.

For the first and third sets of experiments, results were generated by running the disk traces through a trace-driven disk simulator that emulates each of the fixed-threshold power management policies. For the second set of experiments the file system traces were run through a simple memory/file system simulator that executed on top of our disk simulator. We experiment with various buffer memory sizes ranging from 2 MB to 128 MB. In addition to the fixed-threshold policies, we emulate a predictive policy that makes spin-down decisions using our idle phase length prediction algorithm.

We use a disk model with four power states based on the IBM TravelStar disk. We do not use the Sleep state, and we assume that the disk enters the Idle state as soon as there are no pending requests. Because the real disk transitions into its various levels of Idle state somewhat more slowly, our results are appropriately conservative: they underestimate the energy consumed by the base (No-Spin-Down) policy. The parameters of our disk model are shown in Table 2.

4.1 Delaying Asynchronous Operations

This section presents the impact of delaying asynchronous operations on energy consumption for various fixed-threshold policies. Our experiments employ a trace representing 220 minutes of interactive laptop use by a CS graduate student. The workload includes a mix of programming (editing), debugging and compiling code.

All figures in this section present results for five different policies:

No-SD: The No-Spin-Down policy.

Linux: The original Linux disk scheduler with a fixed-threshold spin down policy.

[3] An exception is the case of fixed-threshold policies that use inappropriate thresholds. For example, using a fixed spin-down threshold of fifteen seconds on a disk with a ten second break-even time will exhibit increased energy consumption if the length of idle periods is increased from ten seconds to twenty seconds.

M-NoLimit: Our disk scheduler with a fixed-threshold spin down policy without any limit on the time for which a request may be delayed. This strategy results in delaying all asynchronous requests for an indefinite period of time. Postponed requests are serviced at the arrival time of the next synchronous request. This policy provides an upper bound of the energy savings that a load-change disk scheduler may achieve.

M-10, M-20, M-30, M-60: Our disk scheduler with a fixed-threshold spin down policy. Synchronous requests are never delayed; asynchronous requests may be delayed for up to 10, 20, 30, or 60 seconds, respectively.[4]

Energy Savings

Figure 6 displays the relative energy savings of the spin down policies with respect to the No-Spin-Down Policy. Figure 7 provides a close up for thresholds in the range of 0–60 seconds. Table 3 shows the best case energy savings. The policies that delay asynchronous write requests achieve their largest savings with aggressive thresholds in the range of 1–6 seconds. In addition they provide 0.7%–24.6% more energy savings than a strategy that does not rearrange the disk request load. A limit on the period of time that a request may be delayed of 30 seconds leads to average energy savings of 25.0%, or 14.1% additional energy savings when compared to the original Linux disk scheduling algorithm.

Slowdown

Figure 8 shows the slowdown of various policies in comparison to the No-Spin-Down policy. The measurements are based on the total time required by the hard disk in order to complete the workload. We assume that the disk requests will be scheduled in the same order even in the presence of disk re-activation overheads. For all thresholds shorter than five seconds, our load-change algorithms perform better than the original Linux scheduler. The reason behind this performance improvement is that the clustering of requests achieved by our load-change strategy results in fewer spin-down operations. A 10 second threshold, which for the original Linux disk scheduler produces the best average energy savings (10.9%), leads to a slowdown of 4%. For the same threshold, our load change strategy with a 30 second write delay results in a comparable average slowdown (3.9%) with 16.5% energy savings. A 60 second write delay leads to an average slowdown of 4.3% and average energy savings of 25.5%. Increased energy savings are obtained by our load-change algorithm, when more aggressive thresholds in the range of 2–5 seconds are used, but with a slowdown in the range of 4.5–7.6%.

Spin-Down Operations

Figure 9 presents the number and type of spin down operations. As write delay increases, the total number of spin down operations and the number of spin down operations that lead to increased energy consumption, labeled *Bad*, decrease. The number of beneficial spin down operations also decreases as the write delay increases. The reason is that by increasing the write delay, the total number of idle phase is reduced. For thresholds in the range of 1–10 seconds, which provide the best energy savings, the load change algorithms with write delays of 20 seconds or more substantially decrease the number of spin down operations.

[4]The delay imposed by our disk scheduler is in addition to the thirty second delay of the Kupdate daemon. Hence, a 60 second delay limit in our load change strategy may lead to a maximum of a 90 second delay.

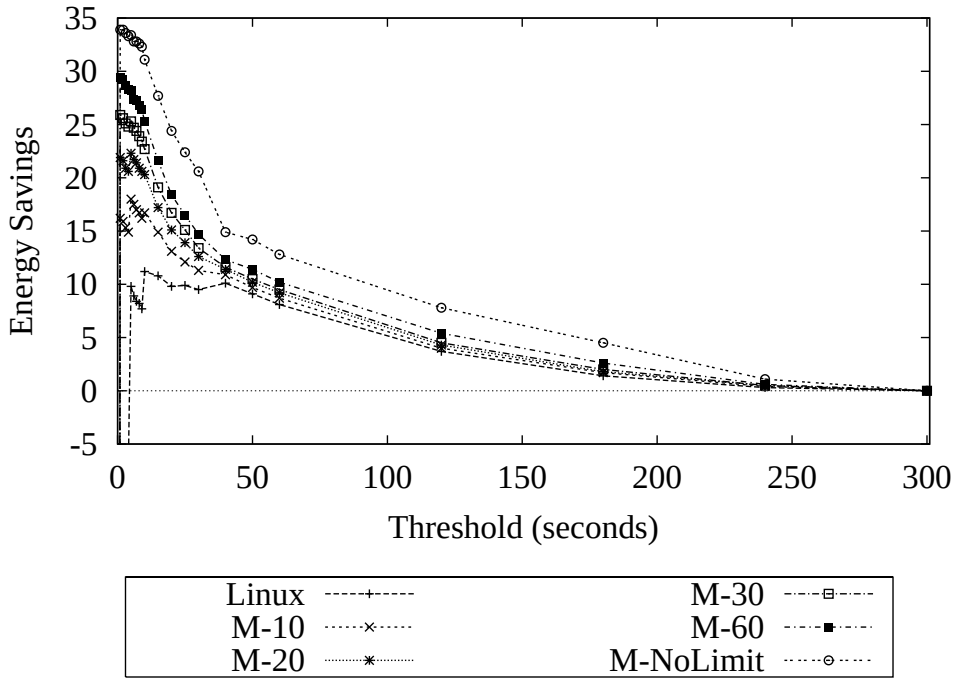

Figure 6: Energy savings of the simulated strategies during the mixed workload.

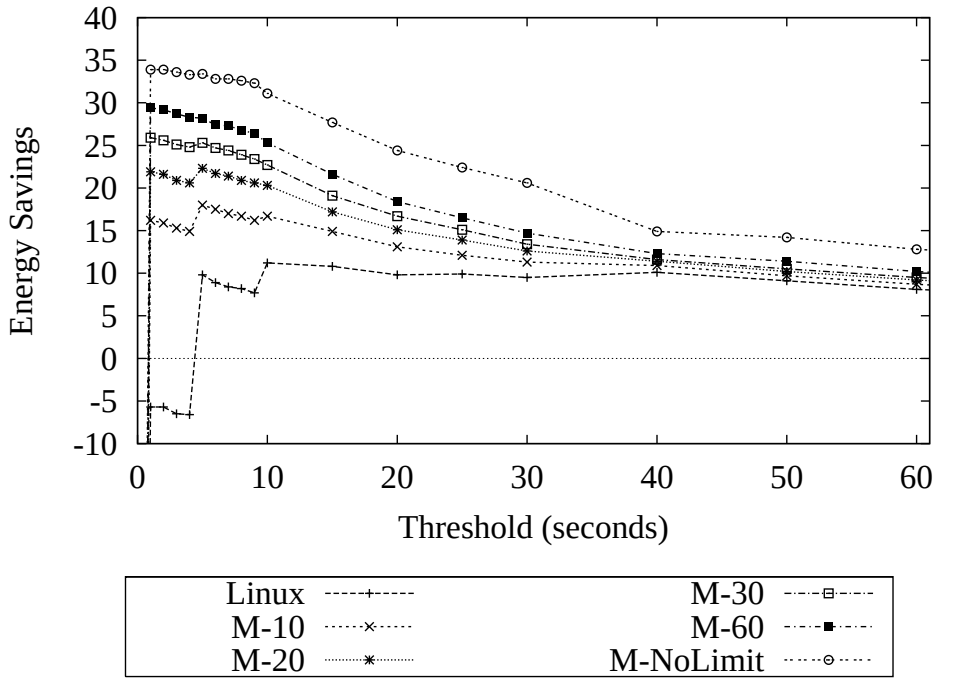

Figure 7: Energy savings of the simulated strategies for thresholds of 0–60 seconds.

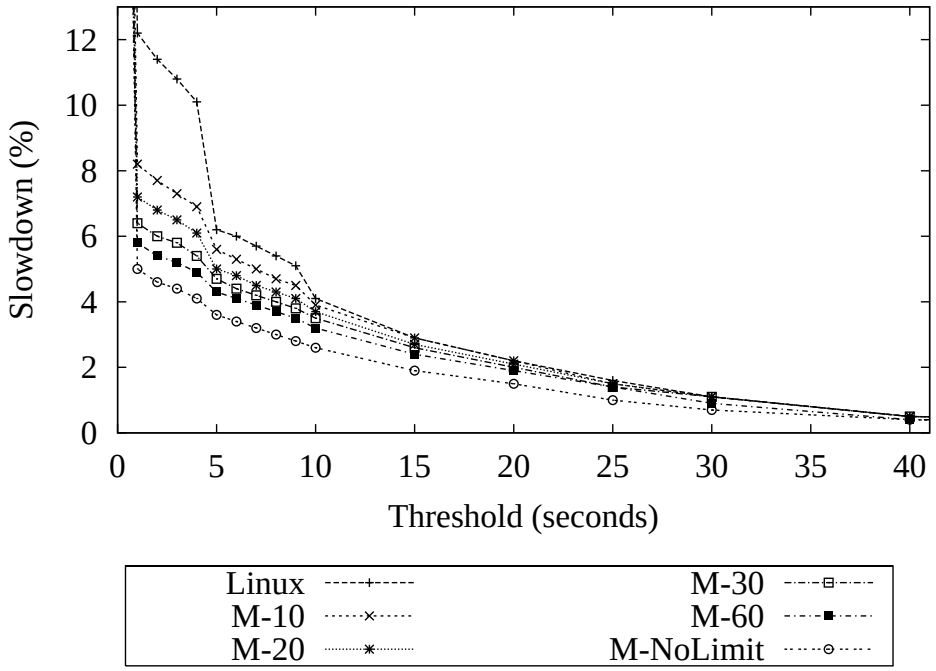

Figure 8: Slowdown relative to the No-Spin-Down policy during the mixed workload.

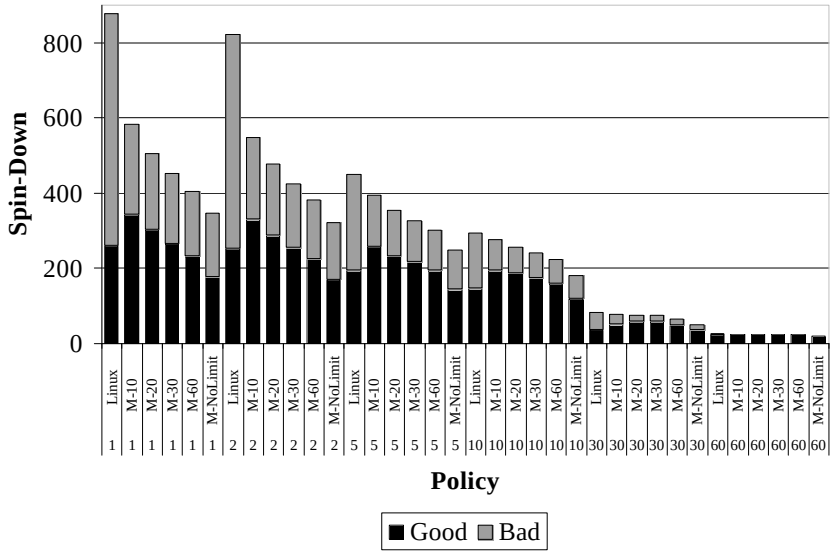

Figure 9: Number of “good” and “bad” spin down operations during the mixed workload.

Policy	Savings	Thres.
Original	10.8%	15
M-10	18.0%	5
M-20	22.3%	5
M-30	25.9%	1
M-60	29.4%	1
M-NoLimit	33.9%	1

Table 3: Best case energy savings for each policy. The policies that delay asynchronous requests achieve their largest energy savings with aggressive thresholds in the range of 1-6 seconds. In addition they provide 7.2%-23.1% more energy savings than a strategy that does not rearrange the disk request load. A low 30 second limit on the period of time that a request may be delayed leads to 25.9%, or 15.1% additional energy savings when compared to the original Linux disk scheduling algorithm.

Name	Description
NoSD	Never spin-down
T1	Fixed threshold of 1 second
T2	Fixed threshold of 2 seconds
T5	Fixed threshold of 5 seconds
T10	Fixed threshold of 10 seconds
Pred	Proposed predictive policy
Opt	Optimal: uses future knowledge
Bursty	Proposed load-change algorithm
Linux	Linux prefetching/update algorithms

Table 4: Power management policies and pattern-shaping algorithms used in the prefetching experiments. Throughout the evaluation each power management policy is combined with a pattern-shaping algorithm (for example Linux-Opt or Bursty-T1). Note that the Bursty algorithms always operate with a limited amount of memory while we do not set a limit on the Linux prefetching and periodic update algorithms.

4.2 Prefetching for Multimedia and Rate-based Applications

In this section, we compare our energy-conscious (*Bursty*) memory/disk management policy, which attempts to increase the burstiness of the disk's usage pattern through aggressive prefetching and postponing writes to the standard *Linux* policy. The Linux results represent the disk usage pattern created by the workloads used on a 512 MB (total) system. For the *Bursty* algorithms we experiment with various buffer memory sizes ranging from 2 to 128 MB. Since the data produced by the applications used are not associated with strict reliability constraints,[5] the limit on the period of time that write requests can be delayed is determined only by the amount of available memory. We assume a fixed threshold power management policy and experiment with aggressive thresholds in the range 1 to 10 seconds. The use of such aggressive thresholds is risky, since it can lead to a significant number of unnecessary disk spin-downs. However, for the usage patterns studied only aggressive thresholds can lead to substantial energy savings. Conservative threshold policies degenerate to the No-Spin-Down policy. In all instances of the *Bursty* algorithm, prefetching is initiated in advance in order to minimize application perceived delays. We also present results for a

[5]The written data can be reproduced again automatically in case of a system crash for all the applications tested.

Applications	Files Read	Files Written	Bytes Read	Bytes Written	Time (sec)	Disk Idle Time (sec)
Mp3 Playback	1	0	4.6 MB	0.0 MB	293.1	291.6
CD Ripping	0	15	0.0 MB	705.8 MB	1359.2	1190.5
Mp3 Encoding	15	15	705.8 MB	64.1 MB	1106.7	912.2
Mp3 Enc/Play	20	15	724.3 MB	64.1 MB	1227.4	1027.1

Table 5: Workload scenarios used in the experimental evaluation. Only the files/data that were associated with disk activity are shown.

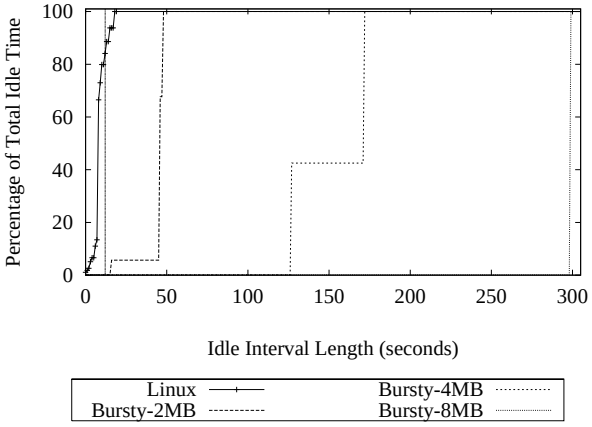

Figure 10: Cumulative Percentage of idle time intervals during mp3 playback.

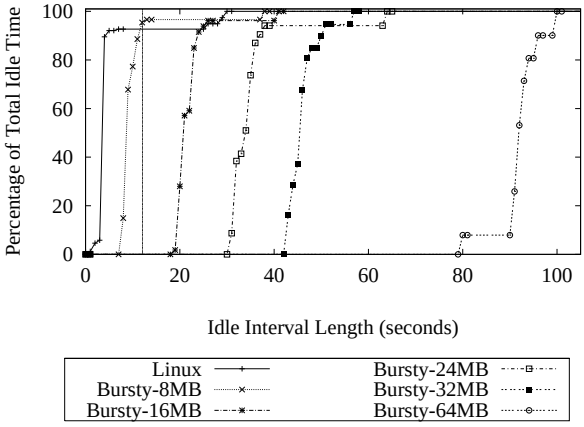

Figure 11: Cumulative Percentage of idle time intervals during CD-ROM ripping.

predictive algorithm, which spins down the disk immediately when the predicted idle phase length is greater than the disk's break-even time (but does not alter the access pattern), and for an *optimal* algorithm, which uses future knowledge in order to decide whether to spin down the disk. Our goals are to show how usage pattern re-shaping can facilitate aggressive power management policies, leading to increased energy savings and minimizing the number of unnecessary power state transitions, and to evaluate the efficiency of predictive techniques that base their predictions on high level operating system knowledge. Table 4 summarizes the above information.

Workload Scenarios

In the experimental evaluation we use four different workload scenarios with different degrees of I/O intensity. The first, mp3 Playback of one file, represents a light read workload. The second, CD ripping, represents a write-intensive workload during which 15 files with a total size of 705.8 MB were written to disk. The third scenario, mp3 encoding, is a read and write intensive workload during which 15 WAV files with a total size of 705.8 MB are encoded into mp3 format, producing 64.1 MB of data. Finally, our most intensive benchmark involves concurrent mp3 encoding and mp3 playback. The input to the mp3 encoder is the same as in the third scenario. During the mp3 encoding process, the mp3 player accesses 5 files with a total size of 18.5 MB. Table 5 summarizes the above information. It also indicates the duration of each workload scenario (from the first to the last disk access—not the application execution time) and the total disk idle time during the execution of the workload.

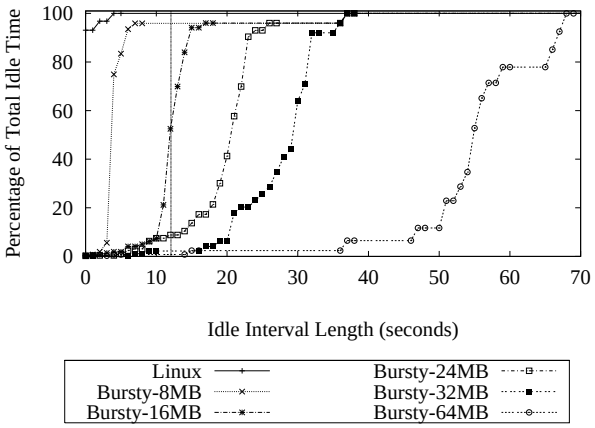

Figure 12: Cumulative Percentage of idle time intervals during mp3 encoding.

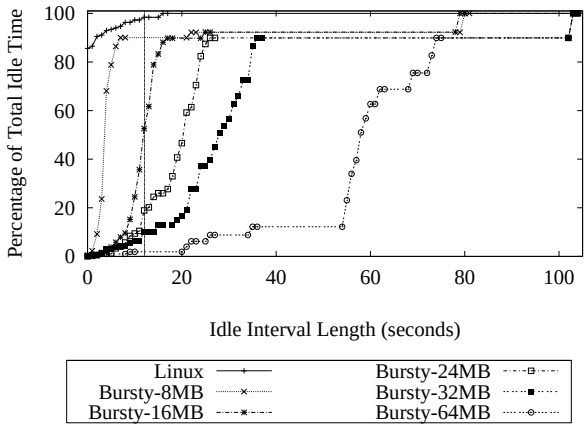

Figure 13: Cumulative Percentage of idle time intervals during mp3 encoding and Mp3 playback.

Idle Phase Length

Figures 10-13 show the distribution of idle time intervals for our workload scenarios. We present results for the standard Linux algorithm and our *Bursty* algorithm using various memory sizes. For the mp3 playback workload the memory size ranges from 2 MB to 8 MB. Using larger memory sizes has the same effect as the 8 MB case, since the whole file (4.6 MB) is prefetched in a single disk operation. For the rest of the workload scenarios, we present graphs for memory sizes ranging from 8 MB to 64 MB. Using less than 8 MB of memory for these scenarios leads to very short idle phases. In all graphs the straight vertical line represents the 12 second break-even point of our disk model. The main conclusion from the idle phase length graphs is that when a sufficient amount of memory is available, our algorithm controls the disk usage pattern so that idle time appears in intervals of approximately equal length that are longer than those that are generated by a standard Linux kernel. Ideally, the graphs for the idle phase length distribution of our algorithm would be a straight vertical line. Variations from the straight vertical line are due to accesses to new files that cause disk activations, since no data are available in memory for such files at the time of the first access.

Energy Consumption

Figures 14-17 present the energy savings results. For the Linux algorithms, we only present energy savings for the *Optimal* and the best realistic policy, which is the *No-Spin-Down* policy for all workloads except the CD-copy scenario, where the 10-second threshold policy performs better. It is important to note that for the disk usage patterns of the Linux algorithms the energy savings are never more than 1.2%, even when the optimal policy is used. The savings achieved by the *Bursty* algorithm depend on the underlying policy and the amount of available memory. When the memory is limited (less than 16 MB), the fixed threshold policies lead to increased energy consumption, because the length of idle phases is usually less than the disk's break-even time. However, with larger memory sizes, idle phase lengths increase and the aggressive fixed-threshold policies lead to significant energy savings ranging up to 55%. The *Predictive* algorithm manages to avoid the mistaken spin-down operations of the fixed threshold policies, when the available memory is limited, and still provides comparable energy savings for increased memory sizes. It follows very closely the behavior of the *Optimal* policy, achieving savings that are always within 5% of those achieved by the *Optimal* policy. For increased memory sizes it leads to decreased energy savings

when compared with the one and two second fixed threshold policies. The explanation is that when a file is first accessed, the predictive algorithm requires a certain period of time, usually greater than two seconds, in order to stabilize. During that period the disk is kept in the idle state in order to avoid unnecessary spin-down operations.

Number and Type of Spin-Down Operations

Figures 18-20 present the number and type of spin-down operations under various power-management policies and memory sizes. We do not show the results for the mp3 playback workload, because it leads to a small number of spin-down operations for all policies (less than three). Also we do not present results for the Linux algorithms: under the usage pattern created by Linux the static threshold algorithms degenerate to the No-Spin-Down policy in most cases.

For each case presented in the graphs, we show the number of spin-down operations that lead to energy savings, labeled *Good*, the number of counterproductive spin-down operations, labeled *Bad*, and the number of occasions where the idle phase length was longer than the disk's break-even time, but the power management policy failed to spin down the disk, labeled *Missed*. The *Predictive* policy has the least number of mistaken spin-down operations across all experiments and makes a comparable number of energy efficient spin-down operations as the most aggressive fixed threshold policies. It misses a few opportunities to spin the disk down, the missed opportunities correspond to relatively short idle phases that could lead to only minor energy savings.

Application Perceived Delay

Currently, our simulator is not accurate enough to predict the effect of our aggressive prefetching algorithms on application execution time. However, we were able to capture the effects of spin-up overhead and disk congestion. The principal result is that applications reading files for which at least one prefetching cycle has completed do not suffer any performance penalties due to disk activation overhead or disk congestion. However, the first read access to a file suffers a significant penalty of 1.9 to 3 seconds, depending on the amount of memory being used. The first 1.8 seconds of the delay are caused by the disk reactivation overhead of our disk model, while the remaining penalty is caused by congestion in the disk queues. In addition, the penalty for the first read access to a file increases slightly as the memory size increases, because larger memory sizes allow an increased number of requests to remain in the disk queue. Using a prioritized disk queue that gives priority to synchronous reads over asynchronous writes will decrease the effect of disk congestion on the completion time of read misses.

4.3 File Prediction

The limit on energy savings achieved for the applications in Section 4.2 was caused by accesses to new files. In addition, most of the application-perceived delays were experienced during the first access to a file. Additional energy savings and improved system responsiveness may be possible by reading files in advance during periods that the disk is spinning. Since file access tends to exhibit spatial and temporal locality, we believe that aggressively prefetching files found in the same directory or even the same directory tree as currently accessed files may lead to significant disk power savings. In order to get a first order approximation of the possible energy savings, we used a 95 minute trace collected during playing a commercial computer game (Sid Meier's Alien Crossfire Demo – Version 6.0 by Loki Software Inc.[6]). We assume that all files in

[6]Loki Software Inc. ports commercial computer games produced for Microsoft Windows platforms to Linux platforms.

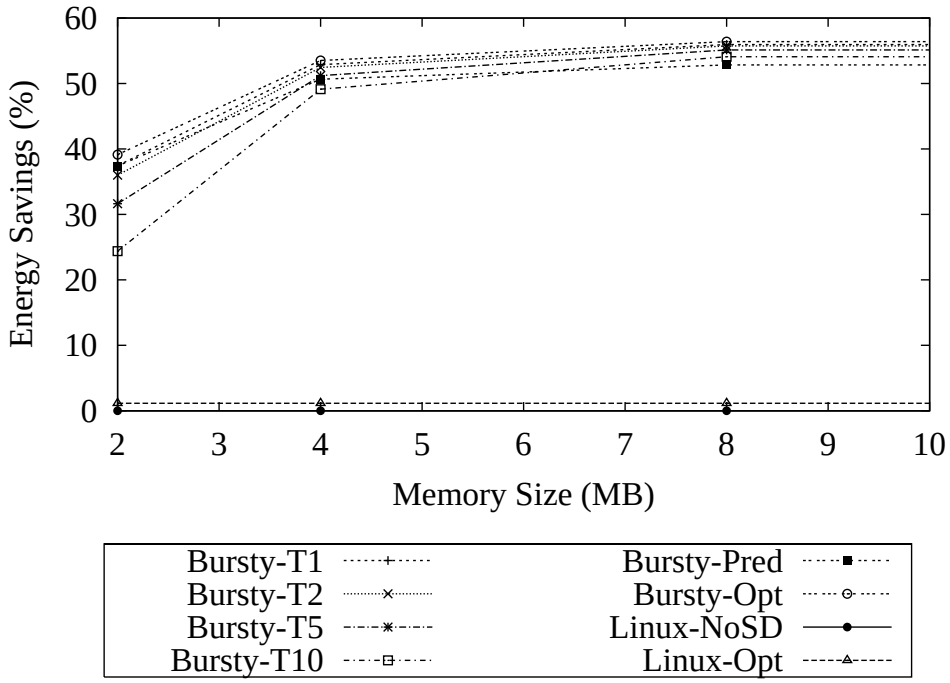

Figure 14: Energy savings during the mp3 playback workload scenario.

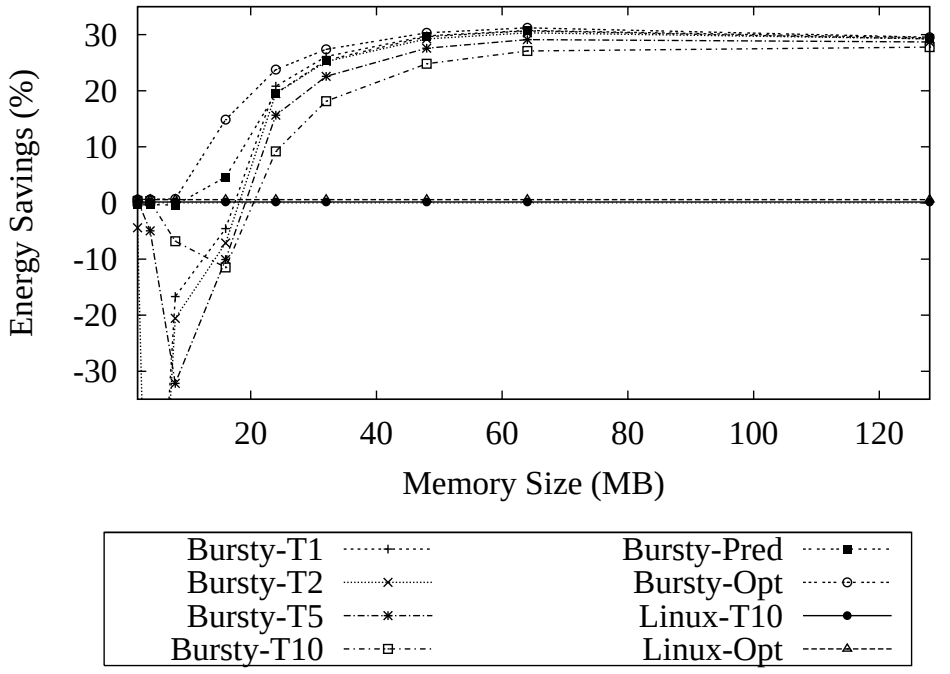

Figure 15: Energy savings during the CD copy workload scenario.

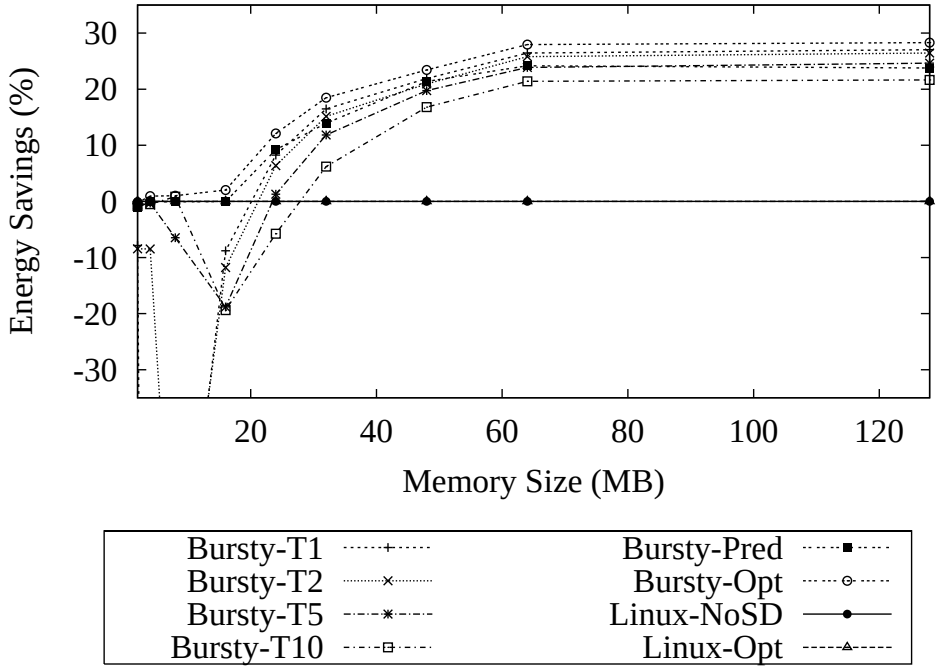

Figure 16: Energy savings during the mp3 encoding workload scenario.

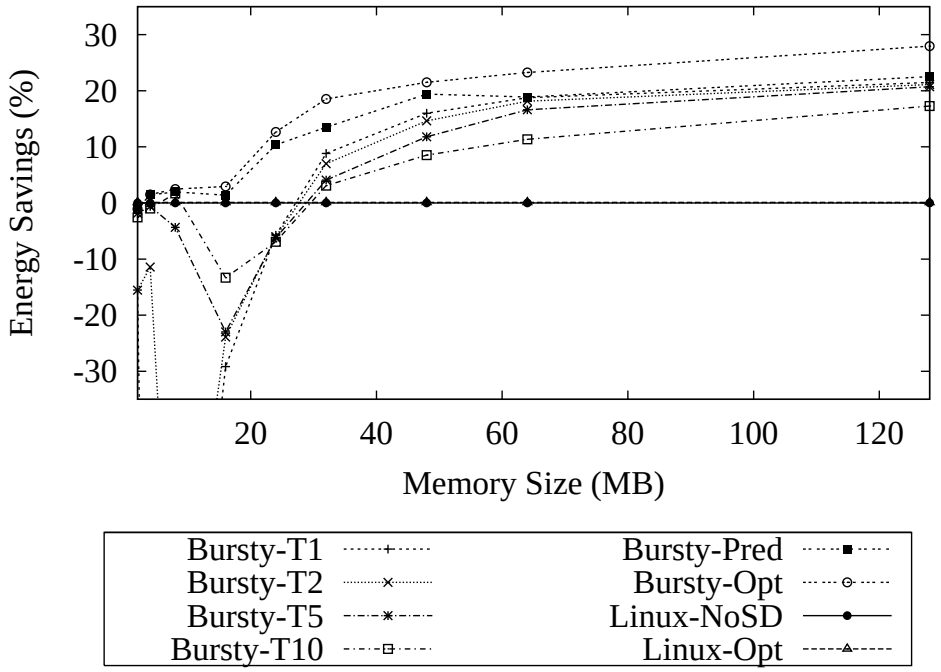

Figure 17: Energy savings during the mp3 encoding/mp3 playback workload scenario.

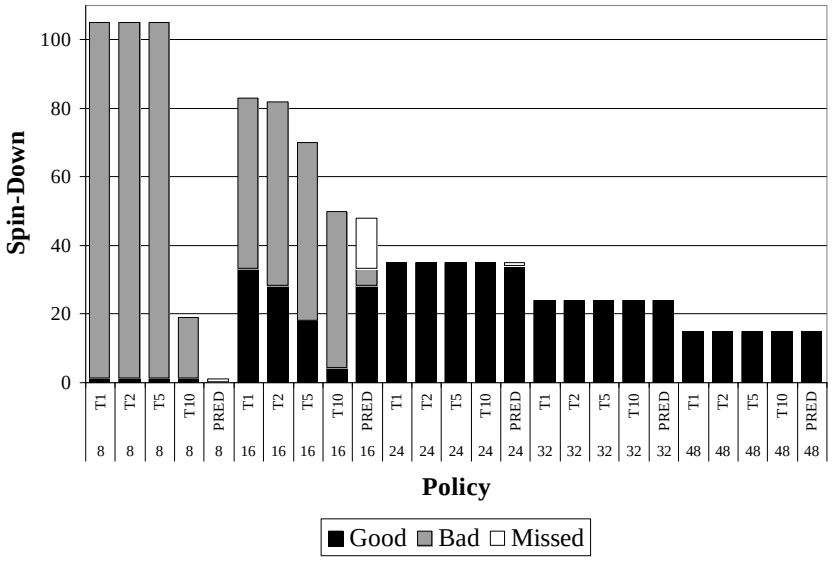

Figure 18: Number of spin-down operations during the CD copy workload scenario.

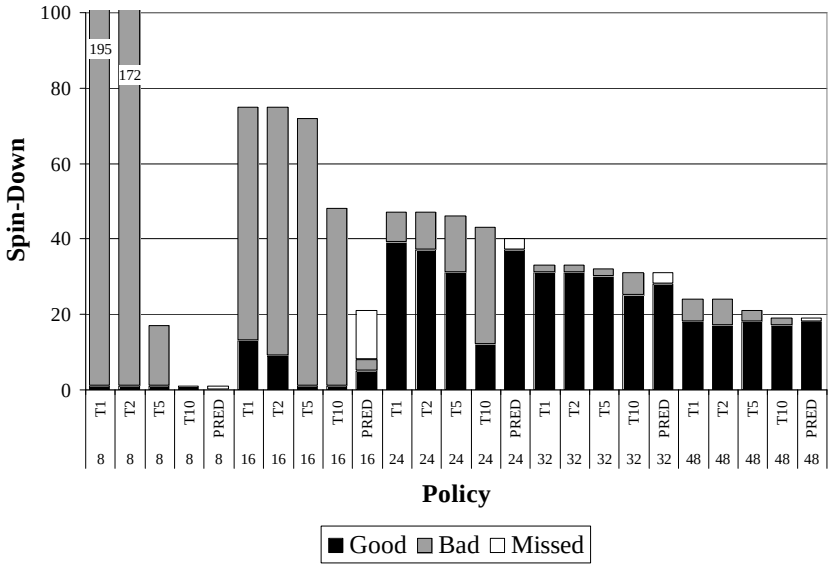

Figure 19: Number of spin-down operations during the mp3 encoding workload scenario.

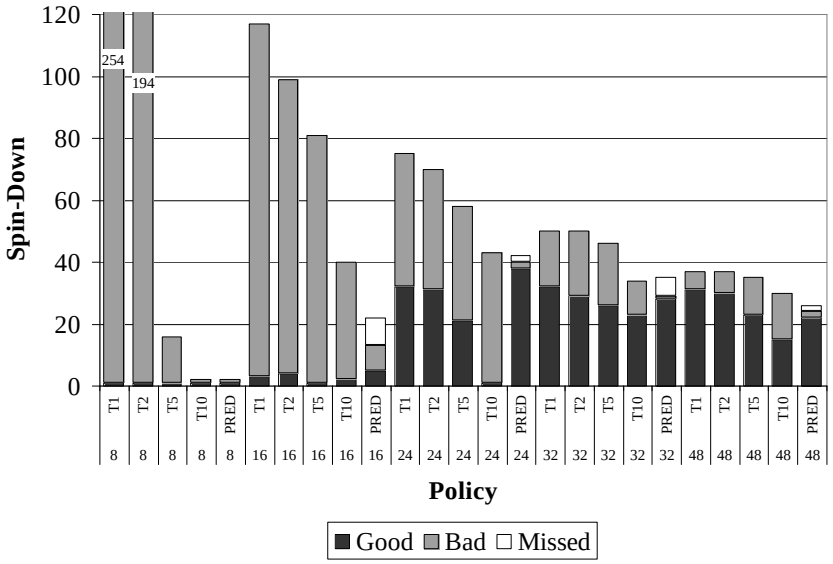

Figure 20: Number of spin-down operations during the mp3 encoding/playback scenario.

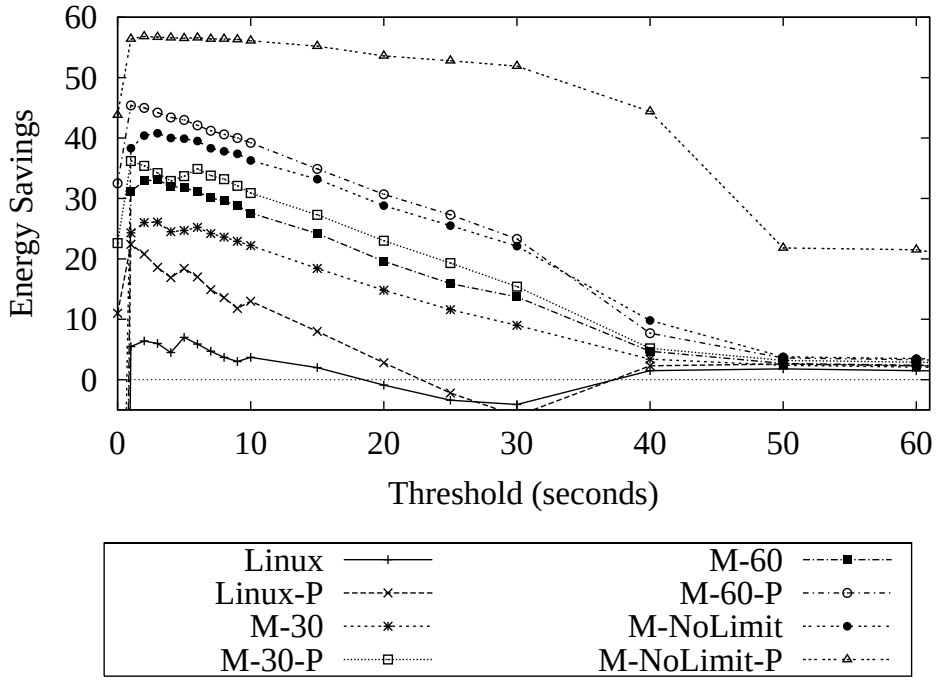

Figure 21: Disk Energy Savings under various policies while playing a computer game.

the directory holding the data of the game are prefetched at the beginning of the execution. Such aggressive prefetching requires the use of a 40 MB memory buffer. Figure 21 shows the energy savings achieved by the original Linux disk scheduler (*Linux*) and our disk scheduler (M-30, M-60, M-NoLimit for 30, 60 seconds and unlimited write delay respectively) under various fixed threshold policies, with (Linux-P, M-30-P, M-60-P, M-NoLimit-P) and without prefetching. Note that in the case of unlimited write delay with prefetching the energy savings increase 45% for an aggressive threshold of 1 second. Since the write activity produced while playing a game does not have significant reliability constraints, delaying it for a significant amount of time will not cause loss of important data in case of a system crash.

In this experiment, the entire game was loaded into memory at start-up. In larger games, whose data do not fit in memory, smart policies could use game locality to prefetch neighboring scenes.

5 Related Work

5.1 Power Management

The research community has been very active in the area of power-conscious systems during the last few years. Ellis *et al.* [Ellis, 1999][Vahdat *et al.*, 2000] emphasized the importance of energy efficiency as a primary metric in the design of operating systems. ECOSystem [Zeng *et al.*, 2002b] provides a model for accounting and fairly allocating the available energy among competing applications according to user preferences. In a more recent report, Zeng *et al.* [Zeng *et al.*, 2002a] propose pricing and bidding techniques based on the currentcy metric used in ECOSystem, in order to coordinate disk accesses and increase the energy efficiency of the hard disk. Odyssey [Flinn and Satyanarayanan, 1999][Noble *et al.*, 1997] provides operating system support for application-aware resource management. The key idea is the ability to trade quality with resource availability. Both Odyssey and ECOSystem aim to achieve a target battery lifetime.

Several policies for decreasing the power consumption of processors that support dynamic voltage and frequency scaling have been proposed. The key idea in such cases is allowing a clever scheduling algorithm to save energy by "squeezing out the idle time" in rate-based applications. Proposed schedulers for general purpose systems are either *interval-based* [Govil *et al.*, 1995] [Grunwald *et al.*, 2000][Weiser *et al.*, 1994] or *process-based* [Flautner *et al.*, 2001][Pering *et al.*, 2000]. Voltage scheduling algorithms have also been designed for real-time systems and have achieved significant energy savings both for uniprocessor [Pillai and Shin, 1999] and multiprocessor systems [Zhu *et al.*, 2002]

Lebeck *et al.* [Lebeck *et al.*, 2000] explore power-aware page allocation in order to make a more efficient use of memory chips supporting multiple power states, such as the Rambus DRAM chips. They compare policies for selecting where (in which memory chip) a page should be stored in terms of an $Energy * Delay$ metric. The idea of increasing the burstiness of the usage pattern can be applied in the case of memory accesses and lead to additional savings. For example, an energy-conscious compiler could reorder instructions, so that load or store operations to the same memory chip appear in short bursts. Tradeoffs between energy and performance in such a case are beyond the scope of this paper.

In the area of power management for hard disks, most of the related work studies transition strategies for changing the power state of the hard disk with a minimal impact on performance. Douglis *et al.* compared predictive transition policies, fixed threshold transition policies using various thresholds, and an optimal policy that uses future knowledge in order to decide when to spin down and when to spin up the disk [Douglis *et al.*, 1994]. Their results showed that fixed threshold policies with short thresholds approach the power consumption achieved by the optimal policy, but the number of delayed operations may increase substantially. Moreover, the best compromise between power consumption and response time is workload

dependent. The predictive policies performed slightly worse than the fixed threshold policies in terms of power both consumption and response time. Douglis' results are also supported by Li *et al.* [Li *et al.*, 1994], who found that the optimal threshold for a 1994 Maxtor MXL-105 III hard disk is only two seconds, while the industry recommended threshold is three to five minutes.

An alternative to fixed threshold and predictive techniques is an adaptive threshold policy. Douglis *et al.* [Douglis *et al.*, 1995] suggest a policy that attempts to reduce the frequency of short spin-down periods, on the assumption that these are the most annoying to the user. Helmbold *et al.* [Helmbold *et al.*, 1996] suggest the use of a machine learning algorithm that dynamically re-weights the recommendations of several prediction algorithms. Our work differs from such previous approaches in that we modify the disk usage pattern rather than simply adapting to it. Moreover, our prediction technique is based on high-level operating system knowledge and application behavior, instead of low level disk usage patterns.

Lu *et al.* [Lu *et al.*, 2000; Lu *et al.*, 2002] proposed the idea to base power state transitions on the behavior of the currently running processes. They suggest monitoring the utilization of every available device by each running task, and combining the individual utilization values in order to predict the future utilization for each device. In addition, they describe a process scheduler that takes into account the predicted device utilization for each running task, and attempts to schedule tasks in a way that maximizes utilization of a certain device and minimizes the utilization of another. We believe that our work complements Lu's. Both ideas try to modify the usage pattern observed by a power-aware device in order to reduce its energy consumption, but they attempt it at a different level of the system. We believe that combining the two algorithms will lead to additional energy savings.

Heath *et al.* [Heath *et al.*, 2002a] investigated the potential benefits of application supported device management for optimizing energy and performance. Possible application transformation include grouping read requests in order to increase the time between requests and providing the operating system with hints about future read requests. In a more recent paper [Heath *et al.*, 2002b], they present a compiler framework for transforming applications automatically. Experiments based on a real implementation and physical measurements suggest energy savings on the order of 55% to 89%. Their approach is complementary to ours. They suggest application modifications in order to cluster requests, while we attempt to solve the same problem at the operating system level. Their approach can achieve significant energy savings for workloads consisting of a single executing application. For workloads where multiple I/O intensive applications are executing concurrently, we believe that a more general memory management algorithm in the kernel of the operating system that can coordinate the generation of disk requests is required.

Finally, Weissel *et al.* [Weissel *et al.*, 2002] are also addressing the issues of disk usage pattern reshaping for energy efficiency in the "Cooperative I/O" project. They show that by providing an API that allows applications to characterize their requests as deferrable or abortable and by increasing the burstiness of the update policy significant energy savings may be achieved.

5.2 Buffer Cache Management

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 [Patterson *et al.*, 1995] and multiple applications [Tomkins *et al.*, 1997]. Cao *et al.* [Cao *et al.*, 1994][Cao *et al.*, 1995] 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. Such a scheme leads to significant performance improvements for applications that have knowledge of their future access patterns through

aggressive prefetching. Curewitz *et al.* [Curewitz *et al.*, 1993] explore data compression techniques in order to increase the amount of prefetched data.

In 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. They avoid prefetching beyond an application's prefetch horizon in most cases, and they do not deal with increased disk overheads caused by disk reactivations. Prefetching for energy efficiency requires a very aggressive prefetching scheme able to cover the data demands of applications for periods longer than 5–15 seconds, depending on the disk's specifications, to coordinate prefetching requests from multiple applications, and to take into account overheads associated with disk congestion and disk reactivation when deciding when prefetching should be initiated.

Several methods have been explored in order to disclose future application access patterns to the operating system. Chang *et al.* suggest generating hints automatically through speculative execution during periods of idle processor time [Chang and Gibson, 1999]. Such an approach cannot be used when the goal is energy efficiency, since speculative execution will increase the amount of energy consumed by the processor. Griffioen *et al.* propose a future access prediction technique based on past file accesses [Griffioen and Appleton, 1994]. Yeh *et al.* describe a method for improved file predictions, based on past associations among files [Yeh *et al.*, 2001b]. They also note the positive impact of more accurate prediction schemes on energy efficiency [Yeh *et al.*, 2001a], without however making an attempt to create a more power-friendly disk access pattern or taking into account the underlying power management policies.

Previous work has also explored periodic update techniques and compared them with write-through policies [Carson and Setia, 1992][Mogul, 1994]. The main conclusion was that periodic updates can lead to degraded performance if they increase significantly the burstiness of the disk usage pattern. Solutions in such cases focus on methods to smooth the usage pattern generated by periodic updates. However, in the case of energy efficiency the goal *is* to increase the burstiness of the disk usage pattern. The performance shortcomings of increased burstiness should be avoided through priority or criticality based disk scheduling algorithms [Ganger and Patt, 1998], aggressive prefetching and accurate file prediction. We explore aggressive prefetching in our current work. We will focus on criticality-based disk scheduling and file prediction in the future.

6 Conclusion

In our study, we investigated the potential benefits of increasing the burstiness of disk usage patterns in order to improve the energy efficiency of the disk spin-down policy. We suggested the use of aggressive prefetching and the postponement of non-urgent requests in order to increase the average length of idle phases. The prefetching algorithm is guided by hints from higher levels of the OS, which monitor program behavior. Annotations are being used in order to notify the low driver about the maximum delay that can be employed on non-urgent requests.

In addition, we presented a method to coordinate accesses of several concurrently executing tasks competing for limited memory resources so that requests are generated and arrive at the disk at roughly the same time. Based on our aggressive prefetching strategy we proposed an algorithm to preactivate the disk, so that application perceived delays due to disk reactivation and disk congestion are minimized. Finally, we designed a predictive algorithm for immediate disk deactivation that takes into account the status of the memory management system and the access patterns of the executing applications.

We evaluated the proposed algorithms through trace-driven simulations. Our results show that our techniques can increase the length of idle phases significantly compared to a standard Linux kernel. Postponing asynchronous requests lead to energy savings of up to 30% and aggressive prefetching can increase the savings up to 55%. The savings depend on the amount of the available memory and the urgency/reliability constraints of the requests. They increase as the memory size increases and decrease as request urgency increases. The limit on the achieved energy savings comes from the requirement to service unpredicted synchronous read requests. A significant number of such requests for the workloads tested is created by the first access to a file, and can be avoided through file prediction. We show that file prediction during workloads that access multiple small files, such as game playing, can achieve up to 55% energy savings. In our current design we consider all synchronous requests to have high urgency. Additional savings could be achieved by recognizing non-urgent synchronous requests and delaying them if necessary.

Though we focused on disks in this paper, we believe that burstiness can help to save energy in any device with low power modes, such as the wireless network. Currently we are working on an implementation of our ideas in the Linux kernel. In our future work we plan to experiment with applications that have irregular access patterns and stricter requirements for synchronous writes. For the latter, we expect to exploit non-volatile (e.g. FLASH) RAM.

We believe that in order to improve the power efficiency of computing systems significantly, power management policies should exploit high level operating system knowledge. In addition, traditional operating system algorithms, such as disk scheduling and memory and buffer cache management, should take into account the power and performance specifications of underlying devices, in order to generate usage patterns that facilitate power management.

References

[ACPI, 2000] "Advanced Configuration and Power Interface Specification (ACPI)," July 2000, Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation.

[Cao *et al.*, 1994] Pei Cao, Edward W. Felten, and Kai Li, "Implementation and Performance of Application-Controlled File Caching," In *Proc. of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 165–177, November 1994.

[Cao *et al.*, 1995] Pei Cao, Edward W. Felten, and Kai Li, "A Study of Integrated Prefetching and Caching Strategies," In *Proc. of the 1995 ACM Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRCIS'95/PERFORMANCE'95)*, pages 188–197, 1995.

[Carson and Setia, 1992] Scott D. Carson and Sanjeev Setia, "Analysis of the Periodic Update Write Policy For Disk Cache," *IEEE Transactions on Software Engineering*, 18(1):44–54, January 1992.

[Chang and Gibson, 1999] Fay Chang and Garth A. Gibson, "Automatic I/O Hint Generation Through Speculative Execution," In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[Curewitz *et al.*, 1993] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter, "Practical Prefetching via Data Compression," In *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, pages 257–266, May 1993.

[Douglis *et al.*, 1995] Fred Douglis, Pillaipakkamnatt Krishnan, and Brian Bershad, "Adaptive Disk Spin-down Policies for Mobile Computers," In *Proc. of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, April 1995.

[Douglis *et al.*, 1994] Fred Douglis, Pillaipakkamnatt Krishnan, and Brian Marsh, "Thwarting the Power-Hungry Disk," In *Proc. of the 1994 Winter USENIX Conference*, pages 293–306, January 1994.

[Ellis, 1999] Carla S. Ellis, "The Case for Higher Level Power Management," In *Proc. of the 7th Workshop on Hot Topics in Operating Systems (HotOS VII)*, March 1999.

[Flautner *et al.*, 2001] Krisztian Flautner, Steve Reinhardt, and Trevor Mudge, "Automatic Performance Setting for Dynamic Voltage Scaling," In *Proc. of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom'01)*, May 2001.

[Flinn and Satyanarayanan, 1999] Jason Flinn and Mahadev Satyanarayanan, "Energy-aware adaptation for mobile applications," In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[Ganger and Patt, 1998] Gregory R. Ganger and Yale N. Patt, "Using System-Level Models to Evaluate I/O Subsystem Designs," *IEEE Transactions on Computers*, 47(6):667–678, June 1998.

[Govil *et al.*, 1995] Kinshuk Govil, Edwin Chan, and Hal Wasserman, "Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU," In *Proc. of the 1st Annual International Conference on Mobile Computing and Networking (MobiCom'95)*, November 1995.

[Griffioen and Appleton, 1994] James Griffioen and Randy Appleton, "Reducing File System Latency using a Predictive Approach," In *Proc. of the USENIX Summer 1994 Technical Conference*, June 1994.

[Grunwald *et al.*, 2000] Dirk Grunwald, Philip Levis, Keith I. Farkas, Charles B. Morey III, and Michael Newufeld, "Policies for Dynamic Clock Scheduling," In *Proc. of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI'00)*, October 2000.

[Halfhill, 2000a] Tom R. Halfhill, "Top PC Vendors Adopt Crusoe," *Microprocessor Report*, 14(7):8–12, October 2000.

[Halfhill, 2000b] Tom R. Halfhill, "Transmeta Breaks x86 Low-Power Barrier," *Microprocessor Report*, 14(2):9–18, February 2000.

[Heath *et al.*, 2002a] Taliver Heath, Eduardo Pinheiro, and Ricardo Bianchini, "Application Supported Device Management," In *Proc. of the 2002 Workshop on Power-Aware Systems (PACS'02)*, pages 114–123, February 2002.

[Heath *et al.*, 2002b] Taliver Heath, Eduardo Pinheiro, Jerry Hom, Ulrich Kremer, and Ricardo Bianchini, "Application Transformations for Energy and Performance-Aware Device Management," In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, September 2002.

[Helmbold *et al.*, 1996] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod, "A Dynamic Disk Spin-down Technique for Mobile Computing," In *Proc. of the 2nd Annual International Conference on Mobile Computing and Networking (MobiCom'96)*, November 1996.

[IBM, 1999] "International Business Machine Corporation (IBM). OEM Hard Disk Drive Specifications for DARA-2xxxxx (6 GB – 25 GB). 2.5-Inch Hard Disk Drive with ATA Interface. Revision (2.1)," November 1999.

[Intel Corporation] "Intel SA-1110 Processor. Intel Corporation".

[Kistler and Satyanarayanan, 1992] James J. Kistler and Mahadev Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992, Earlier version presented at the 13TH *SOSP*, Oct. 1991.

[Lebeck *et al.*, 2000] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla S. Ellis, "Power Aware Page Allocation," In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, pages 105–116, November 2000.

[Li *et al.*, 1994] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson, "Quantitative Analysis of Disk Drive Power Management in Portable Computers," In *Proc. of the 1994 Winter USENIX Conference*, pages 279–291, January 1994.

[Lu *et al.*, 2000] Yang-Hsiang Lu, Luca Benini, and Giovani De Micheli, "Requester-Aware Power Reduction," In *Proc. of the 13th International Symposium on System Synthesis (ISSS 2000)*, pages 18–23, September 2000.

[Lu *et al.*, 2002] Yang-Hsiang Lu, Luca Benini, and Giovani De Micheli, "Power-Aware Operating Systems for Interactive Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(1), April 2002.

[Mogul, 1994] Jeffrey C. Mogul, "A Better Update Policy," In *Proc. of the USENIX Summer 1994 Technical Conference*, June 1994.

[Noble *et al.*, 1997] Brian Noble, Mahadev Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker, "Agile Application-Aware Adaptation for Mobility," In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[Patterson *et al.*, 1995] R. Hugo Patterson, Garth Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka, "Informed Prefetching and Caching," In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[Pering *et al.*, 2000] Trevor Pering, Tom Burd, and Robert Brodersen, "Voltage Scheduling in the lpARM Microprocessor System," In *Proc. of the 2000 International Symposium on Low Power Electronics and Design (ISLPED'00)*, pages 96–101, July 2000.

[Pillai and Shin, 1999] Padmanabhan Pillai and Kang G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 89–102, October 1999.

[Tomkins *et al.*, 1997] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson, "Informed multi-process prefetching and caching," In *Proc. of the 1997 ACM Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRCIS'97)*, pages 100–114. ACM Press, 1997.

[Vahdat *et al.*, 2000] Amin Vahdat, Alvin R. Lebeck, and Carla S. Ellis, "Every Joule is Precious: A Case for Revisiting Operating System Design for Energy Efficiency," In *Proc. of the 9th ACM SIGOPS European Workshop*, September 2000.

[Weiser *et al.*, 1994] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker, "Scheduling for Reduced CPU Energy," In *Proc. of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI'94)*, November 1994.

[Weissel *et al.*, 2002] Andreas Weissel, Bjorn Beutel, and Frank Bellosa, "Cooperative I/O: A Novel I/O Semantics for Energy-Aware Applications," In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[Yeh *et al.*, 2001a] Tsozen Yeh, Darrell Long, and Scott Brandt, "Conserving Battery Energy through Making Fewer Incorrect File Predictions," In *Proc. of the IEEE Workshop on Power Management for Real-Time and Embedded Systems at the IEEE Real-Time Technology and Applications Symposium*, pages 30–36, May 2001.

[Yeh *et al.*, 2001b] Tsozen Yeh, Darrell Long, and Scott Brandt, "Using Program and User Information to Improve File Prediction Performance," In *Proc. of the International Symposium on Performance Analysis of Systems and Software (ISPASS '01)*, November 2001.

[Zeng *et al.*, 2002a] Heng Zeng, Xiaobo Fan, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat, "Currentcy: Unifying Policies for Resource Management," May 2002.

[Zeng *et al.*, 2002b] Heng Zeng, Xiaobo Fan, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat, "ECOSystem: Managing Energy as a First Class Operating System Resource," In *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, October 2002.

[Zhu *et al.*, 2002] Dakai Zhu, Rami Melhem, and Bruce Childers, "Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time System," June 2002.