Application Transformations for Energy and Performance-Aware Device Management [*]

Taliver Heath, Eduardo Pinheiro, Jerry Hom, Ulrich Kremer, and Ricardo Bianchini

Department of Computer Science
Rutgers University
Piscataway, NJ 08854-8019

{taliver,edpin,jhom,uli,ricardob}@cs.rutgers.edu

Abstract

Energy conservation without performance degradation is an important goal for battery-operated computers, such as laptops and hand-held assistants. In this paper we determine the potential benefits of application-supported device management for optimizing energy and performance. In particular, we consider application transformations that increase device idle times and inform the operating system about the length of each upcoming period of idleness. We assess the potential energy and performance benefits of this type of application support for a laptop disk. Furthermore, we propose and evaluate a compiler framework for performing the transformations automatically for a disk device. Our experimental results demonstrate that unless applications are transformed, they cannot accrue any of the predicted benefits. In addition, they show that our compiler can produce almost the same performance and energy results that we obtain by hand-modifying applications. Overall, we find that the transformations we propose can reduce disk energy consumption from 55% to 89% with only a small degradation in performance.

1 Introduction

Recent years have seen a substantial increase in the amount of research directed towards battery-operated computers. The main goal of this research is to develop hardware and software that can improve energy efficiency and, as a result, lengthen battery life.

The most common approach to achieving energy efficiency is to put idle resources or entire devices in low-power states until they have to be accessed again. The transition to a lower power state usually occurs after a period of inactivity (an *inactivity threshold*), and the transition back to active state usually occurs on demand. Unfortunately, the transitions to and from the low-power state can consume significant time and energy. Nevertheless, this strategy works well when there is enough idle time to justify incurring such costs.

Previous studies of device control for energy efficiency have shown that some workloads do exhibit relatively long idle times. However, these studies were limited to interactive applications (or their traces), slow microprocessors, or both. Recent advances in fast, low-power microprocessors and their use in battery-operated computers are increasing the number of potential applications for these computers. For instance, non-interactive applications, such as movie playing, decompression, or encryption, are now commonly run on laptop computers. For most of these non-interactive applications, fast processors reduce device idle times, which in turn reduce the potential for energy savings. Furthermore, incurring re-activation delays in the critical path of the microprocessor now represents a more significant overhead (in processor cycles), as re-activation times are not keeping pace with microprocessor speed improvements.

Thus, to maximize our ability to conserve energy without degrading performance under these new circumstances, we need ways to increase device idle times, eliminate inactivity thresholds, and start re-activations in advance of device use. Device idle times can be increased in several ways and at many levels, such as by energy-aware scheduling or prefetching in the operating system, by performing loop transformations during compilation, etc. The set of possibilities for achieving the other two goals, namely elimination of inactivity thresholds and device pre-activation, is more limited. In fact, those goals can only be achieved with fairly accurate predictions of future application behavior, which can be produced consistently with programmer or compiler involvement. For these reasons, we advocate that programmers or compilers, i.e. applications, should be directly involved in device control in single-user, battery-operated systems such as laptops.

[*]This research was partially supported by NSF CAREER award CCR-9985050.

To demonstrate the benefits of this involvement, in this paper we evaluate the effect of transforming explicit I/O-based applications to increase their idle times. These transformations can be performed by a sophisticated compiler, but can also be implemented by the programmer after a sample profiling run of the application. For greater benefits, the transformations must involve an *approximate* notion of the original and target idle times. Thus, we also evaluate the effect of having the application inform the duration of each idle period (hereafter referred to as a CPU run-length, or simply *run-length*) to the operating system. With this information, the operating system can apply more effective device control policies. (For simplicity, we focus on the common laptop or hand-held scenario that only one application is ready to run at a time; other applications, such as editors or Web browsers, are usually blocked waiting for user input.) In particular, we study two kernel-level policies, *direct deactivation* and *pre-activation*, that rely on run-length information to optimize energy and performance.

As a concrete example of the use of our proposed transformations, we apply them in the management of a laptop disk. Our experiments show that several common laptop applications do not exhibit long enough run-lengths to allow for disk energy savings. To evaluate the potential of application transformations and application/operating system interaction, we manually transform applications, implement the policies in the Linux kernel, and collect experimental energy and performance results. The results demonstrate that the transformed applications can conserve a significant amount of disk energy without incurring substantial performance degradation. Compared to the unmodified applications, the transformed applications can achieve disk energy reductions ranging from 55% to 89% (70% on average) under our most sophisticated energy management policy with only a small performance degradation.

Encouraged by these results, we implemented a prototype compiler based on the SUIF2 compiler infrastructure that automates the manual code transformations, and in addition performs run-time profiling to determine buffer sizes and run-lengths. Our preliminary results for the compiler-based application transformations are as good as those for the hand-modified applications.

Based upon our hand-optimized and compiler-based experimental results, we conclude that application-supported device management can be very useful in terms of energy and performance.

In summary, we make the following contributions:

- We propose transformations to explicit I/O-based applications that increase their run-lengths. Another transformation informs the operating system about the upcoming run-lengths. Our main goal is to evaluate these transformations in detail.
- We implement and experimentally evaluate our transformations for real applications and a real laptop disk, as they are applied by the programmer or with compiler support. We also consider the effect of operating system-directed prefetching.

The remainder of this paper is organized as follows. The next section discusses the related work and highlights the aspects that distinguish our contributions. Section 3 describes the type of application transformation we advocate. Section 4 details the different policies we consider. Section 5 describes the disk and application workload we consider, and presents the results of our analyses and experiments. Finally, section 6 summarizes the conclusions we draw from this research.

2 Related Work

Application support in the control of devices. There have been several previous proposals for giving applications greater control of power states [4, 15, 13, 17, 1, 9]. Carla Ellis [4] articulated the benefits of involving applications in energy management, but did not study specific techniques or policies. Lu *et al.* [15] suggested an architecture for dynamic energy management that encompassed application control of devices, but did not evaluate this aspect of the architecture. In a more recent paper, Lu *et al.* [13] studied the benefit of allowing applications to specify their device requirements with a single operating system call. Microsoft's OnNow project [17] suggests that applications should be more deeply involved, controlling all power state transitions. Flinn and Satyanarayanan [5] first demonstrated the energy benefits of application adaptation.

Our work differs from these previous approaches in that we propose a different form of application support: one in which the application is transformed to increase its run-lengths and informs the operating system about each upcoming run-length, after a device access. This strategy allows us to handle short-run-length and irregular applications. Our approach also simplifies programming/compiler construction (with respect to OnNow) without losing any energy conservation opportunities.

Delaluz *et al.* [1] and Hom and Kremer [9] are developing compiler infrastructure for similar approaches to application support. Delaluz *et al.* transform array-based benchmarks to cluster array variables and conserve DRAM energy, whereas Hom and Kremer transform such benchmarks to cluster page faults and conserve wireless interface energy. Both groups implement their energy management policies in the compiler and use simulation to evaluate their transformations.

In our approach, the compiler or programmer is responsible for performing enabling transformations to increase the possible energy savings, but the actual management policies are implemented in the operating system for two reasons: (1) the kernel is traditionally responsible for managing all devices; and (2) the kernel can actually reduce any inaccuracies in the run-length information provided by the application, according to previously *observed* run-lengths or current system conditions; the compiler does not have access to that information. Nevertheless, our work is complementary to theirs in that we *experimentally* demonstrate the effect of a different form of application transformation and determine their effect under several energy conservation policies.

Direct deactivation and pre-activation. As far as we know, only recently have application-supported policies for device deactivation and pre-activation been proposed [1, 9]. Other works, such as [3], simulate idealized policies that are equivalent to having perfect knowledge of the future and applying both direct deactivation and pre-activation. Rather than simulate, we implement and experimentally evaluate direct deactivation and pre-activation.

Conserving disk energy. Disks have been a frequent focus of energy conservation research, e.g. [21, 12, 3, 2, 8, 15, 10, 6, 14]. The vast majority of the previous work has been on history-based, adaptive-threshold policies, such as the one used in IBM disks. Because our application-supported policies can use information about the future, they can conserve more energy and avoid performance degradation more effectively than history-based strategies. Furthermore, in contrast with previous studies, we focus on non-interactive applications and application-supported disk control.

3 Application Transformations

As mentioned above, non-interactive applications exhibit diminishing opportunities for energy savings and increasing reactivation overheads as processors become faster. To maximize our ability to conserve energy without performance degradation, we need to be able to increase run-lengths and inform the operating system about them. We propose that this can be done by modifying the applications' source codes.

To make the description of these transformations more concrete, the following discussion assumes that the transformations are applied to the control of a disk device. Note however that the transformations are general and can applied to a variety of other devices.

3.1 An Example: Control of a Disk

For a disk, the applications should be modified to cluster disk read operations, so that the processor can process a large amount of data in between two clusters of accesses to disk. If the reads are for consecutive parts of the same file, a cluster of reads can be replaced by a single large read. Disk writes are usually performed in the background, i.e. not in the critical path, so they can be performed whenever the disk is active.

Intuitively, one might think that the best approach would then be to increase run-lengths to the extreme by grouping all reads into a single cluster. However, one must realize that increasing run-lengths in this way will correspondingly increase buffer requirements. Given this direct relationship between run-length and buffer space, we propose that applications should be modified to take advantage of as much buffer space as possible, as long as that does not cause unnecessary disk activity, i.e. swapping. Unfortunately, this approach does not work well for all applications. Streaming applications should have the additional restriction of preventing human-perceptible delays of the stream content. Therefore, a cluster of reads (or large read) should take no longer than this threshold. Here we assume that the threshold is 300 milliseconds.

To determine the amount of memory that is available to an application, we propose the creation of a system call. The kernel can then decide how much memory is available for the application to consume and inform the application.

The following example illustrates the transformations on a canonical (non-streaming) application based on explicit I/O. Assume that the original application looks roughly like this:

```
i = 1;
while i <= N  {
    read chunk[i] of file;
    compute on chunk[i];
    i = i + 1;
}
```

After we transform the application to increase its run-length:

```
// ask OS how much memory can be used
available  = how_much_memory();
num_chunks = available/sizeof(chunks);
i = 1;
while i <= N  {
    // cluster read operations
    for j = i to min(i+num_chunks, N)
        read chunk[j] of file;
    // cluster computation
    for j = i to min(i+num_chunks, N)
        compute on chunk[j];
    i = j + 1;
}
```

A streaming application can be transformed similarly, but the number of chunks of the file to read (`num_chunks`) should be `min(available/sizeof(chunks), (disk_bandwidth` $\times$ `300 millisecs)/sizeof(chunks))`. Regardless of the type of application, the overall effect of this transformation is that the run-lengths generated by the computation loop are now num_chunks times as long as the original run-lengths.

As a further transformation, the information about the run-lengths can be passed to the operating system to enable the policies we consider. The sample code above can then be changed to include the following system call in between the read and computation loops:

```
    next_R(appl_specific_func(available));
```

Note that for regular applications, such as streaming audio and video, the operating system itself could predict run-lengths based on past history, instead of being explicitly informed by the programmer or the compiler. However, the approach we advocate is more general; it can handle these applications, as well as applications that exhibit irregularity. Our image smoothing application, for instance, smooths all images under a certain directory. As the images are fairly small (can be loaded to memory with a single read call) and of different sizes, each run-length has no relationship to previous ones. Thus, it would be impossible for the operating system to predict run-lengths accurately for this application. In contrast, the compiler or the programmer can approximate each run-length based on the image sizes, which is what we do in our experiments.

3.2 Compiler Framework

Restructuring the code as described in the previous section is a non-trivial task, since it requires an understanding of the performance characteristics of the target platform, including its operating system and disk subsystem. This knowledge is needed to allocate buffers of appropriate size (for streaming applications) and to approximate run-lengths. In addition, manual modifications of source code may introduce bugs and are tedious at best. We have developed a compiler framework that takes the original program with file descriptor annotations as input, and performs the discussed transformations automatically, allowing portability across different target systems with different disk performance characteristics.

In the current compiler framework, a user may declare a file descriptor to be streamed or non-streamed. If no annotation is specified, I/O operations for the file descriptor will not be modified by the compiler. Our current prototype implementation is based on the SUIF2 compiler infrastructure [19] and takes C programs as input. Declarations of the form

FILE *streamed_fd and FILE *non-streamed_fd

are supported, which specify the file descriptor fd as streamed or non-streamed. Extending the compiler infrastructure to include the keywords streamed and non-streamed is currently underway. The compiler propagates file descriptor attributes across procedure boundaries, and replaces every original I/O operation of the file descriptor in the program with calls to a corresponding buffered I/O runtime library. The current runtime library contains versions of read and lseek calls. The library calls preserve the semantics of the original I/O operations, and in addition:

- Measure the performance characteristics of the disk through user-transparent runtime profiling,
- Implement buffered I/O through allocation and management of buffers of appropriate sizes, and
- Notify the operating system about the expected idle times of the disk (run-lengths).

In cases where procedures have file descriptors as formal parameters, different call sites of the procedure may use streamed and non-streamed actual parameters, making a simple replacement of the original file I/O operation within the procedure body impossible. Our current prototype compiler introduces an additional parameter for each such formal file descriptor. The additional parameter keeps track of the attribute of its corresponding file descriptor. Using the attribute parameter, code is generated that guards any file I/O operation of the formal parameter, resulting in the correct I/O operation selection at runtime. We are currently investigating the benefits of procedure cloning as an alternative strategy.

The runtime profiling strategy has been designed to be robust across different disk architectures and prefetching strategies. It involves profiling the first few iterations of loops that include read operations to determine averages for the bandwidth of the disk and the run-lengths. Since the profiling is done during actual program execution time, the results may be more precise as compared to runtime profiling as part of a manual translation process, which is typically done off-line and only once, with the resulting parameters "hard-coded" into the transformed program. The runtime overhead of the profiling strategies is negligible and does not affect the user-perceived application performance. Experimental results comparing hand-transformed and compiled program versions are presented in Section 5.3.

Since the source code of the runtime library is available to the compiler, advanced interprocedural compiler transformations such as procedure inlining and cloning can enable further code optimizations. For instance, instead of copying data from the compiler-inserted buffer into the buffer specified in an application-level read operation, the compiler may eliminate the copying by just using a pointer into the compiler-inserted buffer. The safety of these optimizations can be checked at compile time. Manually transformed programs can also take advantage of advanced compiler optimizations.

However, a purely operating system-based, "buffered" I/O approach would require expensive system calls for each original application-level I/O operation. In addition, such an approach may not work well if the files are accessed with a large stride, or accessed irregularly. We are currently investigating compile-time analyses and optimizations to prefetch "sparse" file accesses into a "dense" buffer, and to determine a working set of active file blocks that should be buffered for the non-sequential file accesses.

4 Device Management Policies

We study these transformations in the context of five different device control policies: Energy-Oblivious (EO), Fixed-Thresholds (FT), Direct Deactivation (DD), Pre-Activation (PA), and Combined DD + PA (CO).

For the purpose of terminology, we define the power states of a device to start at number 0, the active state, in which the device is being actively used and consumes the most power. The next state, state 1, consumes less power than state 0. In state 1, there is no energy or performance overhead to use the device. Each of the next (larger or deeper) states consumes less power than the previous state, but involves more energy and performance overhead to re-activate. Re-activations bring the device back to state 1.

Transformations that increase run-lengths have the potential to increase energy savings, because they change the fraction of run-lengths that fall in the different groups we just defined. Increasing the length of run-lengths increases the fraction of run-lengths in larger numbered groups, with a corresponding decrease in the fraction of run-lengths in smaller numbered groups.

Energy-Oblivious. The EO control policy keeps the device at its highest idle power state, i.e. state 1, so that an access can be immediately started at any time. Thus, this policy promotes performance, regardless of energy considerations.

Fixed-Thresholds. The FT control policy recognizes the need to conserve energy in battery-operated computers. It determines that a device should be sent to the consecutive lower-power states after fixed periods of inactivity. We refer to these fixed periods as the their inactivity thresholds. For example, the device could be put in state 2 from state 1 after an inactivity period of 4 seconds (the inactivity threshold for state 1), and later be sent to state 3 after another 8 seconds (the inactivity threshold for state 2), etc. Thus, after 12 seconds the device would have gone from state 1 to state 3. Given that the FT policy can conserve energy in a fairly straightforward fashion, we use it as the baseline when evaluating the other policies.

Direct-Deactivation. The FT policy is based on the assumption that if the device is not accessed for a certain amount of time, it is unlikely to be accessed in the near future. If we knew the run-lengths *a priori*, we could save even more energy by simply putting the device in the desired state right away. This is the idea behind the DD policy, i.e. use application-level knowledge to maximize the energy savings.

Pre-Activation. In both the FT and DD policies the time overhead of bringing the device back from a low-power state to state 1 is exposed to applications, as the transition is triggered by the device access itself, i.e. on demand. However, with run-length information from the application, we can try to hide the re-activation overhead behind useful computation. This is the idea behind PA, i.e. to allow energy savings (through FT or DD) while avoiding performance degradation. For maximum energy savings, the pre-activated device should reach state 1 "just before" it will be accessed. The specific version of PA that we study uses the FT policy to save energy. PA should achieve the same performance as the EO policy, but with a lower energy consumption.

Combined Direct-Deactivation + Pre-Activation. We can achieve the greatest energy savings without performance degradation by combining the PA and DD policies. This is the idea behind the CO policy.

Implementations. We implemented the FT, DD, PA, and CO policies in the Linux kernel. FT is implemented with a kernel timer that goes off according to inactivity thresholds. When the timer goes off, the kernel sends the disk to the next available lower power mode. DD, PA, and CO were implemented by creating a system call that can be called by applications to inform the kernel about the run-length that is about to start. With that information, the kernel can implement DD by determining the best power state according to a model we developed in [7] and putting the disk in that state. The kernel can also implement PA by starting a timer to go off when the disk should be re-activated, again according to our model. Recall that the PA policy assumes FT for energy conservation. The kernel implements the CO policy by combining PA with DD, rather than FT.

5 Experimental Evaluation

This section applies our transformations and policies in the control of the Fujitsu MHK2060AT laptop disk.

5.1 The Fujitsu Disk

The Fujitsu disk we study is a 6-Gbyte, 4200-rpm drive with ATA-5 interface. This particular disk only implements four power states, according to its manual:

0. Active – the disk is performing a read or a write access. The power consumed at this state varies between 1.9 W and 3.25 W.
1. Idle – all electronic components are powered on and the storage medium is ready to be accessed. This state is entered after the execution of a read or write. This state consumes 0.92 W.

Parameter	Explanation
P^s	Average power consumed at state s
T^s	Inactivity threshold for state s
E^s_{act}	Average device energy to re-activate from state s
$E^{s,s'}_{deact}$	Average device energy to transition from state s to lower power state s'
T^s_{act}	Average time to re-activate from state s

Table 1: **Main characteristics of the Fujitsu disk.**

Parameter	Measured Value
P^1	0.92 W
P^2	0.22 W
P^3	0.08 W
T^1 (FT)	9.222 secs†
T^2 (FT)	16.429 secs†
T^1 (PA)	8.712 secs†
T^2 (PA)	17.276 secs†
T^3 (FT and PA)	*Not applicable*
E^1_{act}	0 J
E^2_{act}	1.4 J
E^3_{act}	3.7 J
$E^{1,2}_{deact}$	5.0 J
$E^{1,3}_{deact}$	5.0 J
$E^{2,3}_{deact}$	∼0.0 J
T^1_{act}	0 ms
T^2_{act}	1.600 secs
T^3_{act}	2.900 secs

Table 2: **Characteristics and measured values for the Fujitsu disk. Values marked with "†" were picked so that $(P^s \cdot T^s) + E^s_{act} = E^{s,s+1}_{deact} + (P^{s+1} \cdot T^s) + E^{s+1}_{act}$.**

2. Standby – the spindle motor is powered off, but the disk interface can still accept commands. This state consumes 0.22 W.

3. Sleep – the interface becomes inactive and the disk requires a software reset to be re-activated. This state consumes 0.08 W.

However, our experiments with the disk demonstrate that there are two hidden transitional states. The first occurs before a transition from active to idle. Right after the end of an access, the disk moves to the hidden state. There it consumes 1.75 W for at most 1.1 secs, regardless of policy. The second hidden state occurs when we transition the disk from idle (FT and PA) or the first hidden state (DD and CO) to standby or sleep state. Before arriving at the final state, the disk consumes 0.74 W at this hidden state for at most 5 secs. We do not number these extra states.

Table 1 lists the time and energy characteristics of the disk. Table 2 lists the value we experimentally measured for each of these characteristics. The measurements include

Application	Input	Modified RLs {s1,s2,s3}
MP3 player	11.9-MByte song	{0, 0, 1}
MPEG player	12.75-MByte movie	{0, 0.5, 0.5}
Image smoother	30 images, 2.46 MBytes each	{0, 0, 1}
MPEG encoder	800 files, 115 KBytes each	{0, 0.5, 0.5}
Secure ftp (sftp)	60-MByte file over 10-Mbit Ethernet	{0, 0, 1}
GNU zip (gzip -9)	357-MByte file	{0.07, 0.33, 0.6}

Table 3: **Applications, their inputs, and the grouping of run-lengths in their modified versions (assuming FT states). We consider two streaming (top) and four non-streaming applications (bottom).**

the hidden states, obviously. The values marked with "†" were picked assuming that the disk should stay at a higher power state only as long as it has consumed the same energy it would have consumed at the next lower power state, i.e. $(P^s \cdot T^s) + E^s_{act} = E^{s,s+1}_{deact} + (P^{s+1} \cdot T^s) + E^{s+1}_{act}$. The rationale for this assumption is similar to the famous competitive argument about renting or buying skis [11].

5.2 Methodology

We experiment with real non-interactive applications running on a Linux-based laptop. Unmodified non-interactive applications usually exhibit very short run-lengths; all of them so short that the disk can never be put in low-power state. To achieve greater energy gains, we need applications with longer run-lengths and that call the kernel informing it about their approximate run-lengths. To evaluate the full potential of these transformations, we transformed our applications manually at first.

To determine a reasonable read buffer size for a laptop with 128 MBytes of memory, we determined the amount of memory consumed by a "common laptop environment", with Linux, the KDE window manager, 1 Web browser window, 1 slide presentation window, 1 emacs window, and 2 xterms. To this amount, we added 13 MBytes (10% of the memory) as a minimum kernel-level file cache size. The remaining memory allowed for 19 MBytes of read buffer space.

The transformed streaming and non-streaming applications exhibit the run-length distributions listed in the third column of table 3. The si groups in the table are defined with respect to the run-lengths delimited by the inactivity thresholds of FT. From left to right, each of the values within the braces represents the percentage of run-lengths corresponding to states 1, 2, and 3. For example, a distribution of run-lengths of {0,0,1} means that we measured all run-lengths to be long enough to take the disk to the sleep state. The run-

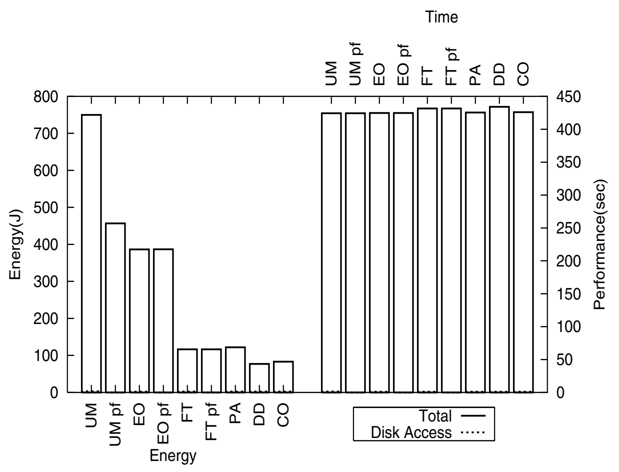

Figure 1: **Energy (left) and time (right) for MP3 player.**

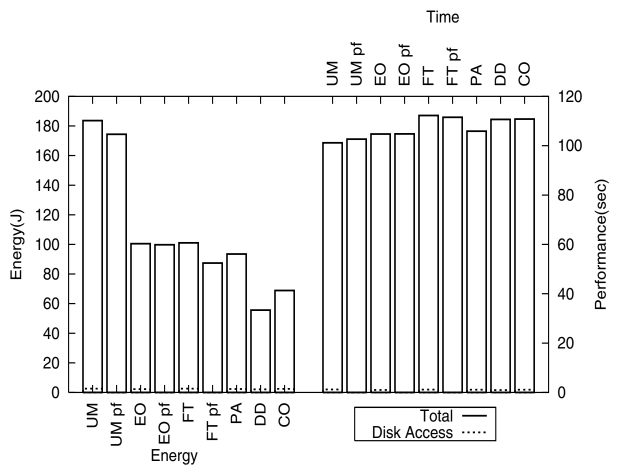

Figure 2: **Energy (left) and time (right) for MPEG player.**

length distributions for the original versions of our applications are always $\{1,0,0\}$.

To understand the effect of operating system prefetching, we execute experiments with and without this optimization. We use the standard prefetching policy of Linux. More specifically, Linux has a variable-size prefetch buffer per open file. The buffer grows up to 128 KBytes, if the file is being read sequentially. It shrinks if reads are not sequential. Prefetching is done synchronously if, upon a read, the file pointer is outside of an already prefetched block and the block being requested is not in memory. Prefetching is done asynchronously if, upon a read, the file pointer is on an already prefetched block. In this case, Linux will request the next window of blocks, up to the 128 KBytes limit.

The disk energy consumed by the applications is monitored by a multimeter directly connected to the disk device. The multimeter collects instantaneous power measurements 3-4 times per second and sends these measurement to another computer, which stores them in a log for later use.

5.3 Results

Figures 1 to 6 present the measured disk energy and performance results for our applications. Each figure plots two groups of bars, disk energy (left) and CPU time (right), with results for all policies. The rightmost bar in each group (labeled "UM") presents the results for the original, unmodified versions of the applications. The bottom part of this bar represents the energy/time associated with actual disk accesses. Next to this bar, we plot the behavior of the unmodified application with operating system prefetching (labeled "UM pf"). The other bars follow this same labeling convention. Note that for the prefetching executions we do not present disk access energy/time because our current low-level instrumentation is not accurate in the presence overlapped communication and computation.

All results besides those for unmodified applications assume application transformations. The EO and FT results assume that run-lengths are extended. The DD, PA, and CO results assume that run-lengths are extended and informed to the operating system. Note that we do not present prefetching results for all policies. We will discuss prefetching later.

Energy. We can make several interesting observations from these figures. Let us start by considering the results without prefetching. These results demonstrate that the application support we propose indeed conserves a significant amount of energy in all cases. The transformation to increase run-lengths reduces energy consumption even under EO, an energy-oblivious policy. When the modified applications are run under FT, energy consumption is further reduced in most cases. The exception here is the MPEG player application, for which two run-lengths are exactly in the range where FT performs worse than EO, namely between 9 and 18 seconds. PA conserves either a little more or a little less energy than FT, as one would expect.

Exploiting run-length information to conserve energy provides even more gains, as shown by the DD and CO results. Modified applications under DD and CO can consume as much as 89% less energy than their unmodified counterparts, as in the case of the MP3 player. Our worst result is for gzip, for which the energy savings is 55% (CO). On average, the disk energy savings we accrue is 70%. The CO policy usually consumes a little more energy than DD. The main reason is that run-length mispredictions may cause the disk to be idle longer than necessary under CO. The same problem is not as severe for PA (percentage-wise) because re-activations under this policy usually come from a shallower state than in CO for these applications.

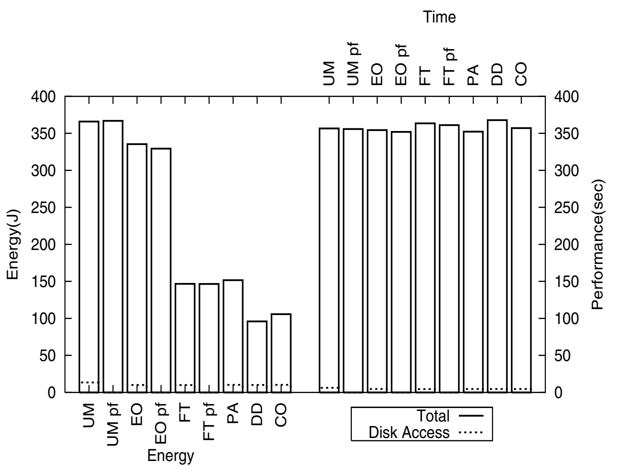

Figure 3: **Energy (left) and time (right) for image smoother.**

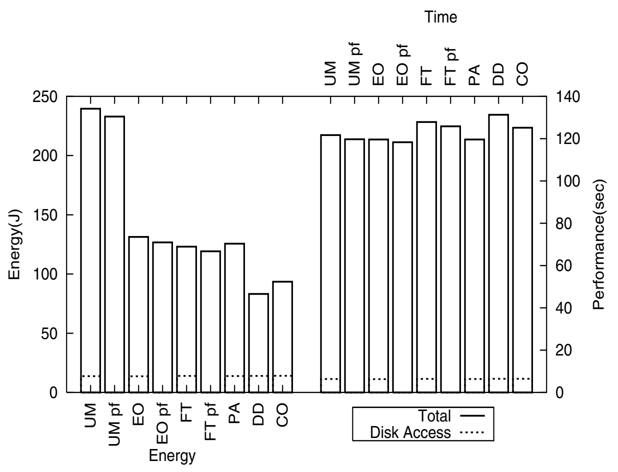

Figure 4: **Energy (left) and time (right) for MPEG encoder.**

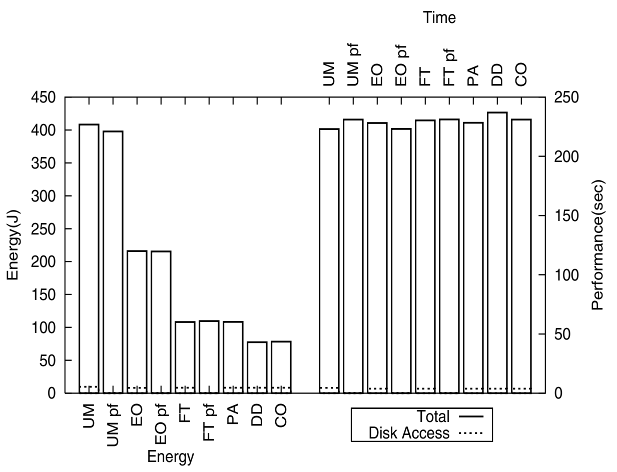

Figure 5: **Energy (left) and time (right) for sftp.**

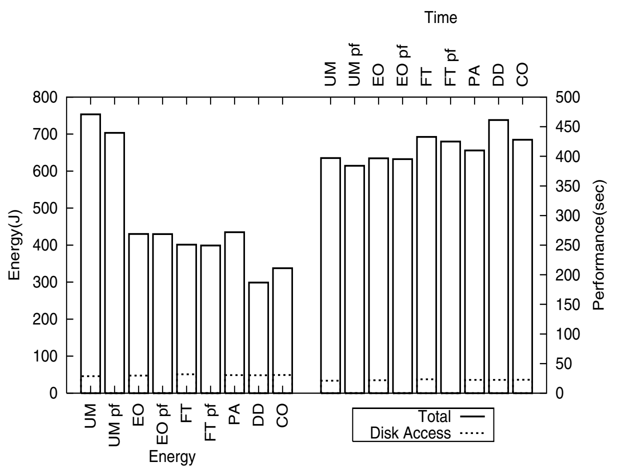

Figure 6: **Energy (left) and time (right) for gzip.**

Putting these energy results in perspective, note that the average power consumed by the disk under UM is between 1 and 2 Watts for all applications. This range is comparable to that of low-power microprocessors (1-2 Watts) [20] and low-power LCD displays (2.5-3.5 Watts) [18, 16]. Given that the display, the microprocessor, and the disk usually consume most of the energy in a laptop, *being able to save an average 70% of the disk energy is highly beneficial even in terms of the energy consumed by the system as a whole.*

Execution time. Still assuming no prefetching, we observe that UM and EO exhibit roughly the same performance, showing that the overhead of extending run-lengths in the way we propose is negligible.

We also observe that FT and DD usually exhibit the worst performance, as one would expect. The disk re-activations are the main cause for the performance degradation under these policies. Furthermore, the figures show that PA and CO are effective at limiting performance degradation. Performance under these policies is always within 5% of that under EO, except in the case of gzip for which the performance degradation is 8%. This discrepancy is a consequence of a few run-length mispredictions that cause disk re-activations in the critical path of the computation. This problem can be alleviated by informing slightly shorter run-lengths than we predict to the operating system or having the operating system itself provide the slack. The amount of slack cannot be too significant however, to avoid increasing the energy consumption excessively.

Prefetching. In terms of performance, prefetching has a negligible effect. The reason is that, for non-streaming applications, our transformations operate on a large (19-MByte) buffer so any additional prefetching the operating system

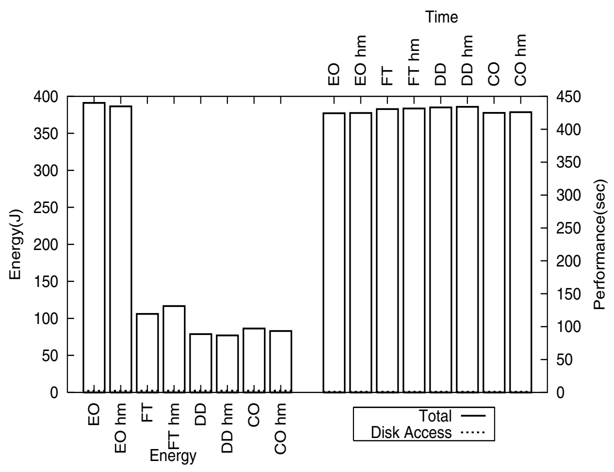

Figure 7: **Energy (left) and time (right) for compiler versions of MP3 player.**

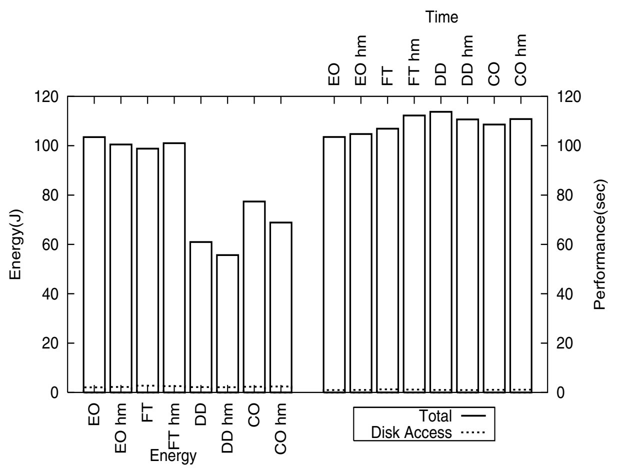

Figure 8: **Energy (left) and time (right) for compiler versions of MPEG player.**

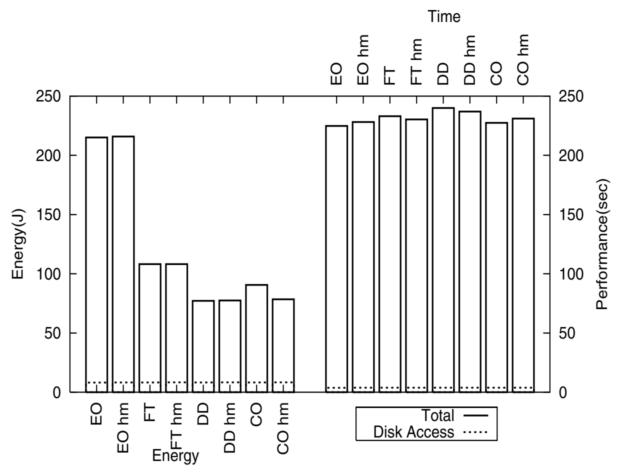

Figure 9: **Energy (left) and time (right) for compiler versions of sftp.**

can perform is unlikely to make a noticeable difference. For streaming applications, the performance is mostly determined by the stream rate which is virtually independent of the performance of disk accesses. Finally, the disk access time for all applications corresponds to a small fraction of the total execution time.

In terms of energy, prefetching does have an effect on the unmodified version of three applications: MP3 player, MPEG player, and gzip. These three applications access data sequentially and in small chunks in their original form. Because prefetching brings larger chunks into memory, it reduces the number of accesses that actually reach the disk, thereby reducing the energy consumption. Recall that a disk access consumes significantly more energy than any low-power state. This effect is not as clearly pronounced for other applications. The limited implications of prefetching is the reason why we do not present all prefetching results in our figures.

Compiler framework. Figures 7, 8, and 9 present a comparison of the hand-modified ("hm") and compiler-based results for the MP3 and MPEG players (both streaming applications), and sftp (a non-streaming application). The results show that the energy consumption and performance of the hand-modified and compiler-based versions are nearly identical in the vast majority of cases. Overall, the compiler framework is able to accurately measure the disk performance without significant energy or runtime overhead, effectively manage the read buffers, and accurately predict the run-lengths.

Summary. Overall, the experimental results demonstrate that the application transformations we propose are extremely effective at conserving energy. Operating system prefetching is not enough to produce significant energy gains. Furthermore, when our transformations are applied, prefetching has no effect on energy or performance. Our compiler framework achieves results that are as good as when we hand-modify the applications. Finally, our results confirm that CO is the best policy in that it conserves significant energy without degrading performance noticeably in all but one case. The main difficulty with CO (and PA) is coming up with accurate run-length predictions.

6 Conclusions

This paper studied the potential benefits of application-supported device management for optimizing energy and performance. We proposed simple and general application transformations that increase device idle times and inform the operating system about the length of each upcoming period of idleness. We also proposed a compiler framework that can transform applications automatically. Using real implementations and physical measurements, we demonstrated the gains achievable by performing the proposed transformations for a laptop disk. Overall, we found that our proposed transformations can achieve disk energy savings ranging from 55% to 89%.

Although this paper considered one particular disk, the transformations and compiler framework we propose should be directly applicable to a wide variety of disks. In fact, the runtime profiling in our compiler was designed to be used transparently across different disks. The operating system modifications we performed are also directly applicable to any disk, provided that the disk parameters are made available to the kernel.

In the future, we plan to confirm these claims by performing the same study with a different laptop disk. The actual energy savings we will accrue depend mostly on the specific ratio between the power consumed in the various disk states. For disks that exhibit similar ratios to our Fujitsu disk, such as the IBM Travelstar, we expect the gains to be comparable to those described in this paper. We also plan to extend our compiler framework.

References

[1] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Dram energy management using software and hardware directed power mode control. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, January 2001.

[2] Fred Douglis and P. Krishnan. Adaptive disk spin-down policies for mobile computers. *Computing Systems*, 8(4):381–413, 1995.

[3] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *Proceedings of the 1994 Winter USENIX Conference*, 1994.

[4] Carla Ellis. The case for higher level power management. In *Proceedings of Hot-OS*, March 1999.

[5] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 48–63, 1999.

[6] Paul Greenawalt. Modeling power management for hard disks. In *Proceedings of the Conference on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Januray 1994.

[7] T. Heath, E. Pinheiro, and R. Bianchini. Application-Supported Device Management for Energy and Performance. In *Proceedings of the Workshop on Power-Aware Computer Systems*, February 2002.

[8] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd International Conference on Mobile Computing*, pages 130–142, 1996.

[9] Jerry Hom and Uli Kremer. Energy management of virtual memory on diskless devices. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.

[10] Chi-Hong Hwang and Allen C.-H. Wu. A predictive system shutdown method for energy saving of event-driven computation. *ACM Transactions on Design Automation and Electronic Systems*, 5(2):226–241, April 2000.

[11] A. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[12] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the 1994 Winter USENIX Conference*, pages 279–291, 1994.

[13] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Requester-aware power reduction. In *Proceedings of the International Symposium on System Synthesis*, September 2000.

[14] Yung-Hsiang Lu, Eui-Young Chung, Tajana Simunic, Luca Benini, and Giovanni De Micheli. Quantitative comparison of power management algorithms. In *Proceedings of the Design Automation and Test Europe*, March 2000.

[15] Yung-Hsiang Lu, Tajana Simunic, and Giovanni De Micheli. Software controlled power management. In *Proceedings of the IEEE Hardware/Software Co-Design Workshop*, May 1999.

[16] *NEC NL8060BC31-01 TFT Color LCD Module*. http://www.lcdspecifications.com/nec.html.

[17] OnNow and Power Management. http://www.microsoft.com/hwdev/onnow/.

[18] *Sharp LM121SS1T53 Color STN-LCD Module*. http://www.lcdspecifications.com/sharp.html.

[19] SUIF. Stanford University Intermediate Format.

[20] Transmeta corporation. http://www.transmeta.com/.

[21] J. Wilkes. Predictive Power Conservation. Technical Report HPL-CSP-92-5, Hewlett-Packard, May 1992.