Execution Context Optimization for Disk Energy

Jerry Hom
jhom@cs.rutgers.edu

Ulrich Kremer
uli@cs.rutgers.edu

Department of Computer Science
Rutgers University
Piscataway, NJ 08854

ABSTRACT

Power, energy, and thermal concerns have constrained embedded systems designs. Computing capability and storage density have increased dramatically, enabling the emergence of handheld devices from special to general purpose computing. In many mobile systems, the disk is among the top energy consumers. Many previous optimizations for disk energy have assumed uniprogramming environments. However, many optimizations degrade in multiprogramming because programs are unaware of other programs (execution context). We introduce a framework to make programs aware of and adapt to their runtime execution context.

We evaluated real workloads by collecting user activity traces and characterizing the execution contexts. The study confirms that many users run a limited number of programs concurrently. We applied execution context optimizations to eight programs and tested ten combinations. The programs ran concurrently while the disk's power was measured. Our measurement infrastructure allows interactive sessions to be scripted, recorded, and replayed to compare the optimizations' effects against the baseline. Our experiments covered two write cache policies. For write-through, energy savings was in the range 3–63% with an average of 21%. For write-back, energy savings was in the range -33–61% with an average of 8%. In all cases, our optimizations incurred less than 1% performance penalty.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Frameworks

General Terms

Languages, Experimentation, Measurement

Keywords

Multiprogramming, synchronization, runtime adaptation, user study

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-469-0/08/10 ...$5.00.

1. INTRODUCTION

Processing speed has maintained a growing trend at an exponential pace. A complementary trend is the shrinking physical size of transistors. These trends allow for expanding the functionality of dedicated systems, consolidating functionality, or fitting off the shelf general purpose parts into embedded systems. Previously dedicated systems, such as mobile phones, personal digital assistants (PDA), and personal entertainment devices have gained significant computing capacity to run a variety of general purpose applications. For example, the Openmoko, iPhone, and Android projects [22, 4, 21] have made great strides toward a general purpose computing platform on mobile phones.

Energy, power, and thermal density are increasingly important design constraints. A system's display, processor, and disk are generally recognized as the top power and energy consumers. Flash memory is a popular storage choice for mobile devices mainly for energy considerations. Yet the choice between flash memory or magnetic disk reduces to a product design tradeoff for performance, energy, and cost. For example, the iPod [5], which functions mainly as a media player, opts for disk storage up to 160 GB. Mobile phone platforms could feasibly use disk instead of flash storage, combining general purpose computing with large capacity storage. Although this research focuses on disk, the energy optimizations and analysis may be applied to other resources which support multiple low power operational modes for energy conservation.

Previous research for disk energy management have generally focused on uniprogramming models of execution. Yet most general purpose systems actually use multiprogramming to run multiple programs concurrently. Operating systems use short time slices to give the illusion of simultaneous execution. The OS mediates among programs when they access various resources, such as the memory and network, giving the illusion of a virtual computing system. The computing paradigm allows a programmer to develop a program without worrying about interference from other programs. In turn, compilers apply optimizations on a program without worrying about how they affect other programs.

An energy-aware compiler can reshape a program to enable and exploit idle periods. Disk requests should be clustered to maximize idle time and provide opportunities for hibernating the disk. The strategy works well in uniprogramming [10, 16] but may lead to poor overall results in multiprogramming. A physical resource can hibernate only when there are no queued requests for a sufficient amount of idle time. Suppose two programs are accessing a resource.

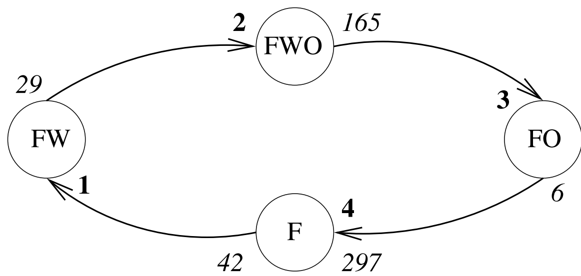

Figure 1: Sample user session trace. Bold numbers indicate order of transitions. Italic numbers indicate the time in seconds spent in that state. Note that F is the start (42) and end (297) state.

In the worst case, their accesses alternate such that the resource repeatedly wakes up and hibernates. They would do better by adapting the hibernation policy threshold [12] or coordinating their accesses. A limited notion of optimizing programs together [18] was introduced to cluster disk requests.

This paper explores the opportunities for disk energy optimizations in execution contexts. We define an execution context as a set of running programs. If people mostly use a single program at a time, then the uniprogramming model suffices. If people regularly use ten programs at a time, then maybe systems are pushed to the limits and offer no opportunity at all. We conducted a user study to gain insight into actual program usage profiles and serve as a guideline for optimization. The usage profiles are modeled as a finite state machine where a state represents an execution context. A transition between states means a program was started or exited. The opportunity for saving disk energy is closely associated with a particular execution context and the interaction between programs. Figure 1 illustrates a sample session from our user study traces. Bold numbers indicate order of transitions, while italic numbers indicate the time spent in that state. The user started by launching Sftp (F), Firefox (W), and OpenOffice (O) in that order. After nearly three minutes in the execution context FWO, the user exited W and O in quick succession but remained in F for another five minutes before exiting. The popular states, FWO and F, dominate the overall active time and represent the opportunity.

Having opportunity is not enough. Realizing the benefits of execution context optimization depends on people's actual activity. If a user is idle in any execution context, then maximum energy is saved without anything special. Hence, programs within an execution context may have large opportunities, but the benefits vary according to user activity. We evaluate the benefits by examining a variety of common applications and their interactions within a diverse set of execution contexts.

Our framework for execution context optimizations combines compiler and runtime elements. We recognize that some techniques could be implemented within the OS. Indeed, the runtime elements would likely be more efficient as OS services rather than user-space mechanisms. Investigating OS approaches combined with our compiler techniques is beyond the scope of this work but certainly an avenue for future work. The contributions of this paper are:

- A user study to identify program usage patterns, quantify the available opportunity for optimization, and provide evidence in support of execution context optimization.
- Language extensions {`STREAMED`, `BUFFERED`, `SYNC_RECV`, `SYNC_SEND`} to categorize program disk request interactions.
- A measurement and evaluation infrastructure using a compiler and runtime system approach to implement execution context optimizations. The infrastructure allows for repeatable, interactive experiments.
- Physical measurements and evaluation of eight programs in ten execution contexts. Measured results from a Hitachi disk showed a range from -33% to 63% energy savings. An energy model estimates disk energy savings based on disk access patterns.

2. RELATED WORK

Heath et al. showed that streaming applications can utilize large disk buffers to save energy [16]. The disk buffers are allocated per program. Prefetching data into the buffer clusters many disk requests. Hence, physical disk access is transformed to a bursty pattern while increasing the idle period between bursts. This optimization works well in uniprogramming environments. However in multiprogramming environments, even just two programs can negate the benefits if their bursty access patterns are unaligned [18]. The interfering disk accesses resembles the original problem of unclustered disk requests in uniprogramming. A new scheduling technique, *inverse barrier*, was introduced to cluster disk accesses across applications [17]. The technique is a variation on barrier scheduling for parallel processing [23] and combines ideas from implicit co-scheduling for distributed systems [6] with the slotted ALOHA protocol [2, 25].

We performed a user study which was similar in scope to Roselli et al.'s efforts to characterize file system workloads [26]. They instrumented the kernel to capture file system events and collected months worth of activity traces from multiple user groups representing different workload patterns. They gained several insights to describe different file system workloads. We also share the same goal with Gurun and Krintz's work to evaluate energy saving techniques on real world workloads [15]. They instrumented the Linux kernel to capture and replay fine-grain event traces while investigating real-time and interactive applications. They then evaluated their dynamic voltage scaling technique on the traces.

Considerable research has been applied to hinting mechanisms which involve the OS and compiler. For example, hints may help ascertain the CPU demand in order to adjust voltage and frequency levels [1]. Hints have also been useful to aggressively prefetch and cache data from disk while comparing different cache policies [24, 29]. Another strategy, called Coop-I/O [28], classifies disk operations as deferrable or abortable. By deferring operations, the OS may batch schedule them at a later time when necessary. However, applications must be manually updated for each I/O function call. Our technique also involves programmer effort to use the new keywords, but source code modifications are minor and limited to tagging file descriptors once when they are declared rather than modifying every I/O function call.

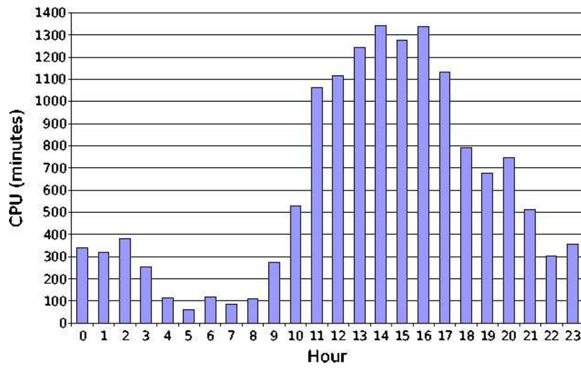

Figure 2: Cumulative user activity by hour.

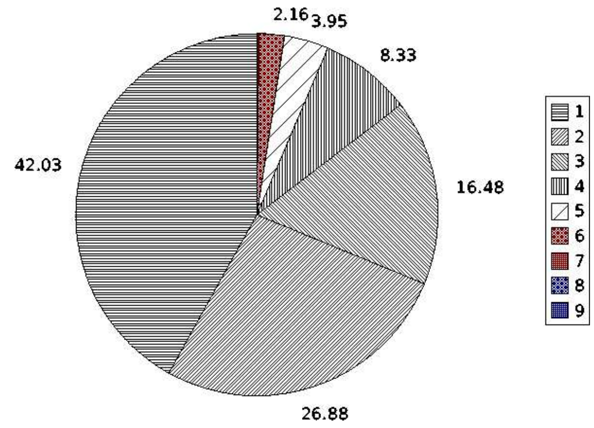

Figure 3: Percentage of time spent in states according to number of concurrent programs.

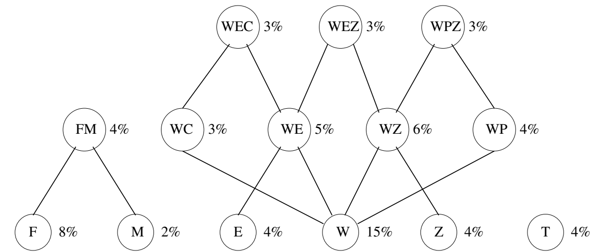

Figure 4: Most popular execution contexts by percentage (at least 2%) of total active time. Edges between nodes indicate a program was started or exited.

3. OPPORTUNITY

Execution context optimization is a promising new research area, and the opportunities for saving disk energy were unknown. Our conjecture is that people tend to use a small number of programs concurrently for a given task. Most people do not keep large numbers of programs running due to resource constraints of the system such as memory. The opportunities for execution context optimizations are inherent in the particular combination of programs. For example, applications which access the disk sparingly, such as a calculator, will have little opportunity. Conversely, streaming applications are characterized by periodic disk accesses and receive the greatest benefits from these optimizations. We want to investigate how often these opportunities occur.

With the help of department system administrators, 40 machines were instrumented in a style similar to Roselli et al.'s setup to trace file systems [26]. The systems are connected to the Computer Science graduate student network; hence, the users are graduate students. The systems are Pentium 4 class machines running between 2.8–3.4 GHz with 512–1024 MB main memory and a standard installation of Fedora Core 3 Linux. The Linux Trace Toolkit next generation (LTTng) [11] was used for tracing. LTTng is a kernel module which monitors events (e.g., file system read and write, signals, process fork and exit). Overall system performance impact is less than 2%. Monitoring the process events `exec` and `exit` was sufficient because they mark when a program starts and exits, effectively describing program lifetimes. Screensaver activity indicates user idleness, and its aggregate time is separated from actual activity time.

We first examined one week's worth of user login sessions to determine when peak activity occurs. Figure 2 shows the cumulative user activity sessions over one week, divided into one hour blocks. We then decided to use a ten hour period (10:00–20:00) on a daily basis for the actual tracing facility. The ten hours capture, on average, 73% of a day's activity. We then traced for four weeks across all machines. The total traced user time was over 860 hours, 12% of which was idle time, leaving over 760 hours of active time by 73 unique users. There were over 60 different programs observed.

From the process events, we identified execution contexts and reconstructed a state graph of program usage. It turns out that indeed, many people run a small number of programs at a time. The contexts with 1–3 programs accounted for 85% of all active time as shown in Figure 3. We observed a general pattern that programs were used as necessary. For example, a group of programs would be launched in succession. The user remains in this execution context until a task is done. Then the programs are exited in succession. Another common pattern is having one or two programs running throughout, such as a web browser or email client.

Table 1 lists the most popular programs by percentage of total active time. The symbols are a shorthand to represent an application in an execution context as shown in the figure below. A handful of applications dominated the usage time. Figure 4 illustrates the most popular execution contexts as a lattice. The contexts shown represent 70% of all active time. Each context is labeled with the percentage of time spent in that state. The states shown are those with at least 2% active time. Lowering the threshold to 1% would cover 80% of all active time and reveal nine more contexts. The rows of the lattice partition the execution contexts into those which contain the same number of concurrent programs. Contexts containing Z represent a program not listed in Table 1. Some programs and execution contexts are clearly more popular than others and should be candidates for optimization.

4. COMPILER / RUNTIME FRAMEWORK

The compiler aspect of our framework consists of new keywords in the form of file descriptor attributes. The attributes categorize a program's disk requests, and the interaction of these requests across programs can be used to synchronize disk accesses. The runtime component includes a synchronization mechanism as well as a file level buffer. The buffer

Table 1: Most popular applications by percentage of total active time. All others individually were less than 4% but combined were 31%.

Application	%	Symbol
web browser	62	W
email	34	E
PDF viewer	23	P
text editor	15	T
file transfer	14	F
internet chat	13	C
matlab	7	M
DVI viewer	5	D
openoffice	5	O
other	31	Z

Table 2: Test programs studied.

Program	Category	Description	Symbol
mpg123	STREAMED	MPEG audio	A
mpeg_play	STREAMED	MPEG video	V
sftp	STREAMED	secure FTP	F
gv	BUFFERED	PS viewer	G
emacs	SYNC_RECV	text editor	T
ooffice	SYNC_SEND	spreadsheet	O
firefox	SYNC_SEND	web browser	W
xpdf	SYNC_SEND	PDF viewer	P

enables clustering of disk requests within a program while synchronization clusters disk requests across programs.

4.1 File Descriptor Attributes

ANSI C specifies two file type abstractions: streams for buffered I/O (`FILE *`) and direct I/O (`int`). We propose new keywords to be optionally used before either type. A file descriptor may be tagged with one of four attributes: `SYNC_SEND`, `SYNC_RECV`, `BUFFERED`, and `STREAMED`. The ordering has a hierarchical meaning — each attribute carries all the semantics of the previous attribute. The keywords inform the compiler as to how the program intends to access the file and the level of interaction with other programs.

The `SYNC_SEND` attribute may be used by practically all programs which have any amount of disk activity. Participation in the synchronization protocol is optional but helpful to other applications. For example, a web browser can notify other programs about disk activity by sending a message after it has read from or written to its disk cache. The `SYNC_RECV` attribute is special among the keywords because it needs extra code by the programmer to take full advantage. This attribute means a program can take useful action when receiving a synchronization message. For example, a text editor with auto-save may decide to save early, in combination with another program's disk access, when receiving a message. The programmer is tasked with adding a handler routine, similar to a signal handler, which reacts to receiving a message. The handler routine is program specific since only the programmer knows an appropriate action to take. The `BUFFERED` attribute is useful for programs which access large data files sequentially but not periodically in time. For example, a postscript document can be reasonably large, and a user may read through the document page by page. As an interactive application, the amount of time before accessing the next page is entirely up to the user. When the buffer's data is consumed, the buffer prefetches data to refill itself. Disk accesses are effectively clustered at buffer refill points. The `STREAMED` attribute is useful for programs which also access large data files sequentially and do so periodically. Some streaming applications, such as audio and video, are also referred to as real-time, meaning that they try to meet minimum performance goals by skipping ahead if computation lags too much. Buffering helps ensure data is ready for computation, but each time the buffer refills itself, there is a significant delay waiting for the disk to wakeup on demand before fetching new data. The delay may be avoided by knowing the program's data consumption rate and activating the disk early such that prefetching occurs just-in-time. Disk accesses are still clustered and application performance is preserved.

The studied test programs are categorized by keyword and listed in Table 2. The PDF viewer cannot take advantage of `BUFFERED` because the portable document format [3] specifies an indexed, pointer-based data layout. The data for a PDF document is stored non-sequentially in the file. All test programs are written in C and C++. OpenOffice and Firefox also have extra modules written in Java, but they were disabled.

4.2 Runtime

Each keyword corresponds to distinct mechanisms for implementing pieces of the runtime framework. The keywords provide a way for programmers to add cooperation between programs with minimal effort. One common aspect is a synchronization policy. The synchronization point should occur after a program has accessed the disk. A group of accesses may correspond to a single program event such as saving a file or advancing to the next page of a document. The compiler must determine where a program's logical I/O operation ends, but finding the end of a logical operation via static analysis is an undecidable problem.

A first attempt would look at the Definition-Use chain of I/O calls and mark a synchronization point after each call. However, a logical I/O operation often encompasses several physical accesses in a loop pattern. A second attempt might add the synchronization function outside the loop, but the first loop block may be enclosed within another loop. Searching through higher nesting levels eventually leads back to the top-level main function and no closer to finding the logical end point. However, there are two problematic cases to consider as illustrated in Figure 5. The pseudo-code is representative of the programming structure for logical I/O operations. They generally conform to a loop pattern and continue until a delimiter has been reached. If a loop iteration takes longer than the disk's hibernation threshold, then Case 1 is appropriate. If a loop iteration takes shorter, then a second attempt would look at the loop surrounding the I/O calls and then decide Case 2 would be better. Yet Case 2 misses the perspective that the loop may be enclosed within another loop. Searching through higher nesting levels eventually leads back to the top-level main function and no closer to finding the logical end point. A heuristic must be used, and Section 5.3 describes one.

If a file descriptor is tagged with `SYNC_SEND`, code for sending synchronization messages is inserted at the end of logical I/O operations. The message is understood by other programs that the disk has just been accessed. If a file descrip-

```
while (NOT FINISHED) {        //
    read (file, buffer, n);   //
    SYNCHRONIZE ();           //  CASE 1
    process (buffer);         //
}                             //

while (NOT FINISHED) {        //
    read (file, buffer, n);   //
    process (buffer);         //  CASE 2
}                             //
SYNCHRONIZE ();               //
```

Figure 5: Synchronization points should be placed at the end of a logical I/O operation. Finding such points is an undecidable problem.

tor is tagged with SYNC_RECV, the compiler gives the same treatment as with SYNC_SEND and also inserts code to receive messages. The code will dispatch to a handler function (provided by the programmer). If a file descriptor is tagged with BUFFERED, the analysis for finding synchronization points is changed slightly. The buffer implementation is provided as a library and mediates between the program and disk for data requests. The I/O calls using the file descriptor are trapped and redirected to the buffer. Physical disk access are transformed to now occur when refilling the buffer. The buffer has clustered the logical I/O operations. In this case, the synchronization point is well known. Lastly, if a file descriptor is tagged with STREAMED, the compiler supplies the enhanced buffer implementation with the added feature of determining the disk's bandwidth and the program's data consumption rate.

If there are n available programs, there are $(2^n - 1)$ possible execution contexts. Our user study has shown that optimizing a small subset of these contexts can sufficiently cover the most popular contexts. Transitions between contexts must also be handled by the runtime component so programs may adapt to their context. For example, in a context with three programs using BUFFERED, each program should cooperatively share the available memory, perhaps by grabbing only a one-third portion. However, in a context where only one program uses BUFFERED while two programs use SYNC_SEND, the BUFFERED program may use all available memory.

The execution contexts can be encoded in each program as a state diagram. A transition in states corresponds to a program either starting or exiting. Thus, after a transition, the extant programs should identify the new state and adapt to it. For instance, the programs may share a memory segment, much like a central whiteboard, to indicate the current execution context. When a program starts or exits, it reads the old state, computes and writes the new state, and notifies other programs of the state change.

5. SETUP AND IMPLEMENTATION

The evaluation infrastructure consists of several hardware and software pieces to automatically control the measurement recording and experiment test sessions. The idea is to run the baseline and optimized versions of the test programs while measuring the disk energy. The tests must be repeatable to assure a fair comparison. However, testing interactive applications is particularly challenging since interactive implies human intervention. While a human could follow a script of actions, the precision and timing would be too poor to reliably repeat the tests. A robot who could emulate a human would be more suitable. Indeed, I will describe such a robot which greatly improved the precision of the experiments.

Table 3: Specifications, operation modes, and power levels for two Hitachi Travelstar disks. The E7K60 was used in our experiments while the C4K60 is listed for comparison. Sleep mode was not utilized during testing. The times and power levels for transitioning between standby and idle modes are reported as physically measured averages with standard deviations. Transition times can vary widely. Specification sheet lists Standby→Idle time as 3.0 seconds (typical) and 9.5 seconds (maximum).

	E7K60	C4K60
Form Factor	2.5"	1.8"
Capacity	40 GB	40 GB
Cache	8 MB	2 MB
Speed	7200 RPM	4200 RPM
Supply Voltage	5.0 V	5.0 V
Active	2.5 W	1.5 W
Idle	2.0 W	0.68 W
Standby	0.25 W	0.12 W
Sleep	0.10 W	0.11 W
Standby→Idle	1.6 s (0.4)	—
	3.1 W (1.0)	—
Idle→Standby	0.6 s (0.2)	—
	2.9 W (2.0)	—
Break-Even	3.5 s	—

5.1 Hardware

The host computer has an AMD Athlon processor at 1.2 GHz with a default workstation installation of Red Hat 9 Linux. We installed a Hitachi Travelstar E7K60 as our target disk. The Travelstar product line is commonly referred to as laptop class disks. They are designed for smaller form factors and better energy efficiency than desktop class disks. If we had tested on a desktop disk, energy savings from our techniques would have been even greater. Although our user study tracked program workloads on desktop machines, we expect to see similar workloads on laptops and emerging general purpose handheld systems. Disk hibernation strategies calculate a hibernation or break-even threshold as a function of a disk's physical characteristics. The E7K60, along with the C4K60 for comparison, specifications are listed in Table 3. The C4K60 model belongs to the 1.8" form factor family which are commonly found in iPods. We physically measured the transition times and power levels. A Tektronix TDS3014 oscilloscope with a Hall effect current probe is attached to the wire supplying current to the disk. The host computer supplies power at a constant five volts; therefore, measuring the supply current readily translates into the disk's power consumption. The TDS3014's maximum sampling rate is 1.25 giga-samples per second, and we chose a reporting resolution of 20 milliseconds. That is, each data point represents an average of the past 20 milliseconds. Most disks have an onboard cache for data with

a small portion reserved for control instructions. The data portion is further divided between reads and writes, both with their own policies. It turns out that the read cache has negligible impact for the workloads in our experiments. Programs exhibiting spatial locality would have the most benefit because read caches typically perform sequential prefetching, except when the working set sizes are greater than the cache size, such as with multimedia files. Our optimizations add file level read buffers which are significantly larger than the cache. The write cache has a choice of policies, write-through or write-back, to affect data safety and performance. The write-through policy uses synchronous writes meaning that data is written to disk immediately and ensures safety at the cost of performance. The write-back policy uses asynchronous writes by buffering data and flushing to disk only when necessary. Disk performance, response time, and energy consumption is improved because of the clustering effect. If working set sizes are greater than the cache size, then the situation is similar to the read case. The policy's downside is the risk of losing data if the supply power is cut before the cache flushes its buffer. User-level write buffers could have been added as an extension of our optimizations. Our synchronization optimization achieves an effect similar to the write-back policy by clustering data accesses, whether read or write, from multiple programs. We tested our techniques under both write cache policies.

5.2 Software

The OS maintains a disk cache per open file via its virtual file system buffer cache. However the default maximum size at 128 KB is an order of magnitude smaller than the disk's onboard cache. For read performance, such a relatively small cache has negligible impact in our experiments. Any write cache policies will depend upon the file system type. The default file system, Second Extended File System [7], supports asynchronous and synchronous writes which are analogous to the disk cache's write-back and write-through policies. Asynchronous writes improve performance and energy in the exact same style as the disk cache's write-back policy. In both read and write cases, any performance and energy effects of the small file system cache are masked by the larger disk cache. Thus, we set the file system to use synchronous writes which matter only when testing with the disk's write-back cache policy.

Our software test suite includes interactive programs such as a web browser. The key for such programs is user interaction such as clicking on a link or typing a web address. Unfortunately, user interaction lends poorly toward repeatability for experiment testing as noted by Crowley [9]. Fortunately, the `X Window System` contains extensions, Record and XTEST [30, 13], which provide hooks to record and replay all keyboard and mouse events. One implementation to utilize the extensions is GNU Xnee. Xnee is dubbed "Not an Event Emulator" because it records actual X11 protocol events, such as keyboard and mouse, to a file which may be used as a script for replay. The replayed events appear to the `X Server` as though coming from the physical keyboard and mouse. Xnee is described by its author as a robot and suitable as a testing infrastructure [27].

Xnee records events with a time stamp precision of one millisecond. As a robot, Xnee can be expected to consistently replay events within a margin of ten milliseconds — the default length for a process time slice. Such precision is useful for timing sensitive programs, but two issues reduce the usefulness, particularly for interactive programs. First, programs which communicate over the network are subject to latency which varies at any given moment. A repeated experiment must allow for a range of latency delay between program events. For example, typing a new website address comprises several keyboard events, but keyboard events have insignificant delays due to buffering. Higher-level program events such as loading a web page require waiting several seconds before the next program event. Second, CPU response also has a variable delay due to background processes, overhead, disk latency, and network packet traffic. Xnee does have a synchronization feature, but this feature was not yet robust enough to reliably repeat our experiments. We found that allowing a margin of five seconds before the next significant event covers most latency issues during replay. The timings may be fine-tuned afterward to suit unusual latencies. The coarse granularity also helps to distinguish between events when analyzing results.

We wrote a control program, running on the data acquisition computer, to coordinate the oscilloscope and the experiments running on the host computer. When the host computer signals ready, the control program signals the oscilloscope to start recording, then runs Xnee with the next session script. When the session script has finished, the host computer signals the experiment has ended, and the control program signals the oscilloscope to stop recording. The host computer will then flush its disk, file, and memory caches before signaling ready again.

5.3 Optimization Stages

We used the Low Level Virtual Machine (LLVM) compiler infrastructure [19] to implement code transformations in various passes. Among its many features, LLVM extends gcc with a new intermediate representation, an analysis and optimization framework, and several back-ends including portable C. Although LLVM robustly compiles many source codes to binary form, LLVM did not completely handle our eight test programs. Using LLVM exclusively would have required adjustments to the various makefile scripts. Since LLVM can emit C source code, we preferred to perform source to source code transformations, then compile normally without changing the build system. However, this approach also limits our implementation for inter-procedural analysis. LLVM is designed to perform inter-procedural analyses during the linking phase when all object code is available for inspection. We opted to build Definition-Use chains by hand and direct the compiler at those source files containing the interesting points. The overall compilation will be described in two stages as depicted in Figure 6.

For the `SYNC_SEND` or `SYNC_RECV` attributes, synchronization points must be located. As discussed in Section 4.2, finding such points is an undecidable problem via static analysis. Therefore, we developed a heuristic, similar to the work in [15] for distinguishing interactive sessions, to identify logical I/O operations. Our Stage 1 pass instruments the program with profiling code to generate a runtime control flow trace. The program runs a training phase with a training set to induce logical I/O operations. In the trace, we group consecutive disk accesses into the same logical I/O operation if the inter-arrival time between accesses is less than a threshold. We used a threshold of ten seconds. The profiling results identify the synchronization points which are fed to

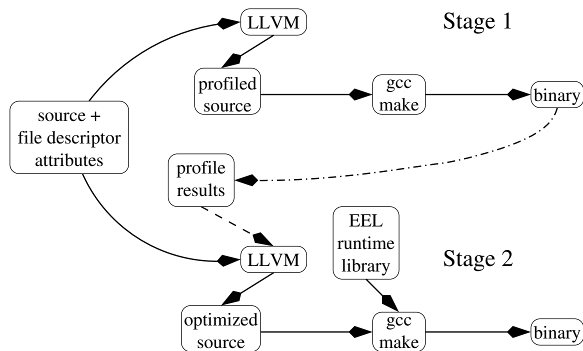

Figure 6: Implementation steps of optimization stages. Stage 1 is used only for SYNC_SEND and SYNC_RECV attribute types. Stage 2 can accept results from Stage 1 to mark synchronization points. Stage 2 then inserts code to implement program synchronization, buffers, and disk profiling.

Stage 2. A program using BUFFERED or STREAMED will bypass Stage 1 because the synchronization points are known.

Stage 2 contains three components which are implemented as separate passes. Pass 1 uses the profiling results from Stage 1 and inserts code for sending disk synchronizations. We used the inverse barrier synchronization policy. Inverse barrier notification was implemented with semaphores to mimic multicast. A true multicast mechanism with message queues would have been more elegant. For programs using BUFFERED or STREAMED, Pass 2 is guided by the Definition-Use chain to replace I/O calls with our library of I/O functions. Our library implements the self-managed, user-level buffer. The function signatures and interfaces are compatible with ANSI C I/O functions and transparent to the original program. Pass 2 also inserts the code for sending notifications since the synchronization points are explicitly marked in the buffer code. Programs using STREAMED will also need to learn a disk's bandwidth and data consumption rate to enable just-in-time disk wakeup. Pass 3 inserts code for that purpose. We developed a profiling mechanism to sample the program's actual execution and pass the results to the buffer. The code inserted from all passes resulted in less than 1% runtime performance delay. Any performance overhead was due to processing synchronization messages.

One final mention goes to a non-optimizing pass we wrote. The SYNC_RECV attribute implies that a program will supply a handler to respond when receiving a message. The text editor, Emacs, is categorized under SYNC_RECV, and we must therefore provide the handler. Our handler responds by invoking the auto-save routine immediately. The default criteria in emacs is to count 30 seconds of idleness or 300 keystrokes before auto-saving. Our aggressive auto-save action is based on research which models user activity patterns as self-similar or bursty. Several researchers have found self-similar patterns in ethernet traffic, web traffic, and file systems [20, 8, 14]. This pattern characterizes inter-arrival times of user input as either very short or long. If the next input is short, then immediate auto-save is negligible. If the next input is long, then immediate auto-save allows maximum idle time for disk hibernation; otherwise auto-save will occur at most 30 seconds from now and incur the energy overhead of waking up and then hibernating the disk.

6. EXPERIMENTS

Our set of experiments investigate the energy and performance impact of execution context optimizations on disk accesses from concurrently running programs. The experiments were devised by using Xnee to record a session and replay it repeatedly while measuring disk energy consumption. The program binaries could be swapped between the baseline and optimized versions. Thus the same session script could be replayed with different program versions. The sessions do not represent all possible real world user behaviors but illustrate a range of common disk activity interactions from a mix of popular programs.

We implemented the previously published buffering optimizations [16] for the baseline programs. Our optimized programs are the result of applying all optimizations described in Section 5.3. Previous results have shown that buffer optimizations (with the uniprogramming model) in multiprogramming environments do save energy vs. unoptimized programs because unoptimized programs never hibernate the disk. Hence, an optimization with at least one profitable disk hibernation will save energy.

6.1 Results

From the eight programs under investigation, we formed ten combinations to represent a diverse range of execution contexts. From a disk energy standpoint, the contexts are important in providing opportunities for savings. The actual program interactions according to the keyword optimizations employed are the direct factors towards realizing energy savings. The ten execution contexts are labelled in Figure 7 using compositions of the program symbols listed in Table 2. Each recorded session was replayed while the disk energy was measured on the oscilloscope. Figure 7 shows the results of our experiments when using the write-through and write-back cache policy. The results in either policy compare the total energy consumption for each session in two pairs. The first pair includes the start-up phase while the second pair contains only the steady state phase. Each pair represents the baseline vs. optimized version. The results are averaged over nine runs.

We first emphasize that the possible disk energy savings depends highly upon the actual activity workload. These ten experiments are synthetic traces to illustrate disk activity interactions between programs. Our user study does not delve deeply enough to characterize user activity and formulate representative traces. A more detailed user study is a monumental task left for future work. The experiments show that with mostly interactive programs such as WP, the opportunity for energy savings is small. Conversely, streaming programs such as AF offer plenty of opportunity for savings. Some contexts have moderate opportunity for savings, such as AWT, though the actual activity did not demonstrate all of the potential savings. We avoided biasing the choice of experiments towards those contexts offering the largest potentials. Using the write-through policy, the average savings was 21% with a range of 3% to 63%. Using the write-back policy, the average savings was 8% with a range of -33% to 61%.

With the write-back policy, there were three sessions in which our techniques performed poorly. Of the three, two had small negative effects while one was greatly affected by consuming 33% more energy. Those three sessions all involved small writes. There were three other sessions with

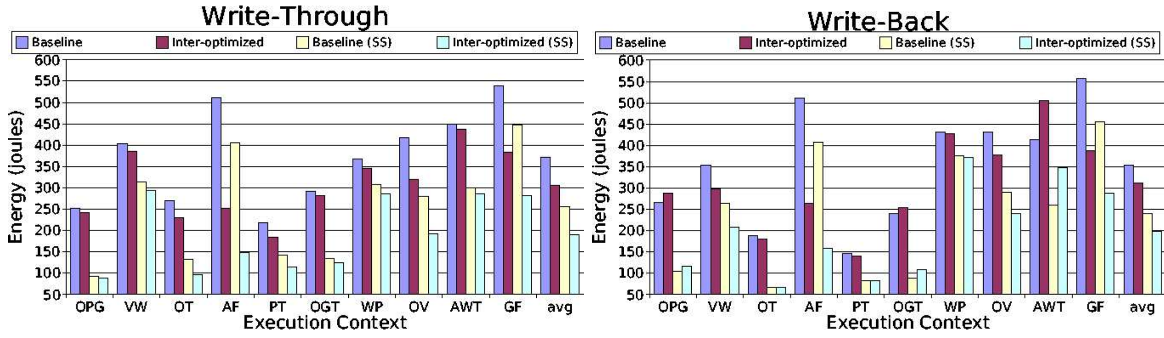

Figure 7: Experiment results when using write-through or write-back cache policies. An execution context represents the set of active programs. Each experiment group compares between the Uniprogramming and Multiprogramming versions. The steady state (SS) results trim away start-up transition phases.

small writes, but the results showed little or no appreciable energy savings. Of the remaining four sessions, the results are similar to the write-through policy. The negative results under the write-back policy are due to the write semantics our techniques employ. Synchronizing disk accesses across programs enforces a synchronous write operation whereas the write cache allows for asynchronous operation. However, if working set sizes were larger than the write cache, then the write cache would have behaved as in a synchronous manner. In addition, our techniques could be extended to add a file level write buffer which may be more efficient than the disk's write cache. The flexibility of these options can be extended further by potentially adjusting the disk's memory allocation between read and write caches. That is, depending on workload and execution context optimizations, the disk's cache may allocate more toward the read cache instead of the traditional 50%.

Secondly, these optimizations performed well overall in reducing disk energy consumption yet with negligible performance costs. The most harmful cases occur when extra disk accesses are triggered via prefetch and the data is not used, when the disk is repeatedly accessed at short intervals, or when small writes trigger early accesses. The last case is harmful only when using a write-back policy. An adversary could perform such activity, but our user study indicates that this is not typical behavior. User activity does follow a Pareto or bursty distribution. In all experiment runs, the overhead of synchronization communication between programs delayed total session execution times by no more than 1%. For real-time programs with buffering, just-in-time wakeup is necessary to maintain performance and the quality of result; previous uniprogramming optimizations are insufficient. Overall, these optimizations are beneficial for saving disk energy with insignificant performance penalty.

6.2 Energy Model

Our optimizations save disk energy via two forms of clustered disk accesses. User level file buffers cluster accesses from each program while synchronization clusters accesses across multiple programs. Both forms appear identical to the physical disk and are modeled as operational power modes over time. Figure 8 illustrates the disk activity of a typical access. Intuitively, the energy savings comes from eliminating wakeup ($E_{\nearrow}$) and hibernate ($E_{\searrow}$) transitions and combining the active periods together. The energy

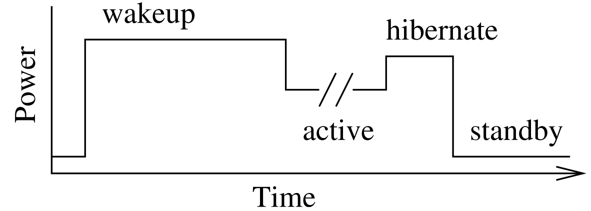

Figure 8: Disk power behavior during typical access request from standby and returning to standby.

saved is proportional to the transition costs of a given disk. Laptop class and smaller disks are designed for fast transitions with lower energy costs then desktop or server class disks. The transition costs are

$$E_{transition} = E_{\nearrow} + E_{\searrow}$$

The baseline energy consumption with M accesses is

$$E_{base} = M \times E_{transition} + E_{active} + E_{standby}$$

The energy reduction by eliminating N transitions is

$$\Delta E = N \times (E_{transition} - E_{standby})$$

Suppose we have an activity trace and an estimate of which disk accesses can be clustered. We choose a time period over which to compute ΔE. In this short trace with seven accesses, we choose the entire trace. With a longer trace, we may choose some activity patterns to represent average activity in the trace. Finding representative patterns is challenging because E_{active} and $E_{standby}$ can have wide variances. Longer time periods will increase accuracy, particularly since M and N are restricted to integral numbers.

A complete energy model would account for times when the disk is in idle mode or has different activity patterns than the typical pattern in Figure 8. For instance, a web browser may encounter network latency such that its logical I/O operation (writing to cache after loading all page objects) stalls, keeping the disk in idle mode. We used the parameters from Table 3 to analyze two traces, OPG and AF. These traces had the least variation in activity patterns.[1]

[1]Measured current flow reveals large fluctuations and error margins, particularly when considering low power disks. Measured activity is in line with manufacturer's measurements.

For OPG and AF, the model estimates energy savings of 2.6% and 55% compared to measured savings of 3.1% and 63%, respectively. The error rates are within one standard deviation.

Lastly, we used our energy model to estimate energy savings of our user study traces. Only some of our experiment contexts can be compared with the user study contexts. Referring back to Figure 4, the contexts F and FM are characterized by streaming applications and are similar to AF from our experiments. The opportunity for energy savings from streaming applications is high and limited mainly by the buffer size. Our experiments with AF used moderate buffer sizes — not so large to buffer the entire file and not so small that the disk must immediately wakeup to refill the buffer. The optimized AF was able to save 63% energy. The contexts WP and WPZ of Figure 4 are characterized as a mix of reading PDF documents and web pages. They correspond most directly to WP from our experiments where the activity consisted of visiting a web page, reading an eight page PDF document (scrolling at regular intervals) linked from that web page, and visiting another web page. This context carries little opportunity for synchronizing disk requests but was able to cluster two accesses into one and save 7% energy. The four contexts {F, FM, WP, WPZ} accounted for 21% of the active time, and if optimized, may have saved about 9% energy over the 730 hours. If the workloads of emerging mobile systems include more multimedia applications, they may find greater advantage from streaming execution context optimizations.

7. SUMMARY

Mobile systems are typically resource limited and rely on battery power. With increasingly multiprogramming environments, execution context optimizations are beneficial to prolong battery life and the usability of the device. This paper has demonstrated that execution context optimizations can be an effective technique to save disk energy for many workloads including a mix of interactive and streaming applications. We introduced a disk request classification system in the context of a compiler and runtime framework to implement optimizations for disk energy consumption. Taking advantage of these optimizations requires the programmer to tag file descriptors, run the code profiling phase, and if using `SYNC_RECV`, supply a synchronization handler routine. The runtime mechanisms for buffering and synchronization are then automatically supplied. The programming model for implementing large file buffering and inter-process synchronization communication is thus simplified for the programmer.

The opportunities to save disk energy are related to the execution contexts and the program interactions within a context. We used LTTng to collect traces (760 hours) from graduate students to validate observations about program usage profiles and explore the value in these optimizations. We used Xnee to develop a robust measurement infrastructure suitable for repeatable experiments on interactive programs. Ten experiment sessions show that execution context optimizations are applicable across a wide range of programs. The measured energy savings are in proportion to the opportunities available (up to 63%) and can be substantially greater over previous best optimizations. The overhead from these optimizations impacts performance by less than 1%. Lastly, we formulated a simple model to estimate energy savings based on a disk's specifications, a disk access profile, and expected program interactions. Applying our model to compare our synthetic traces with execution contexts from the user study, at least 9% disk energy could have been saved. This work is a first analysis at real world multiprogramming workload opportunities and benefits for disk energy execution context optimizations.

8. REFERENCES

[1] N. AbouGhazaleh, D. Mossé, B. Childers, R. Melhem, and M. Craven. Collaborative operating system and compiler power management for real-time applications. In *Real-Time and Embedded Technology and Applications Symposium*, pages 133–143, Los Alamitos, CA, May 2003.

[2] N. Abramson. The ALOHA system — another alternative for computer communications. In *Proceedings of the Fall Joint Computer Conference*, pages 281–285, 1970.

[3] Adobe. *PDF Reference*. Adobe Systems Incorporated, sixth edition, Oct. 2007.

[4] Apple Incorporated. iPhone. <http:// www.apple.com / iphone>.

[5] Apple Incorporated. iPod. <http:// www.apple.com / ipodclassic>.

[6] A. Arpaci-Dusseau, D. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 233–243, Madison, WI, June 1998.

[7] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the Dutch International Symposium on Linux*, Amsterdam, Netherlands, Dec. 1994.

[8] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, Dec. 1997.

[9] C. Crowley. TkReplay: Record and Replay for Tk. In *USENIX Annual Tcl/Tk Workshop*, pages 131–140, Toronto, Canada, July 1995.

[10] V. Delaluz, M. Kandemir, N. Vijaykrishnan, M. Irwin, A. Sivasubramaniam, and I. Kolcu. Compiler-directed array interleaving for reducing energy in multi-bank memories. In *Proceedings of the Conference on VLSI Design*, pages 288–293, Jan. 2002.

[11] M. Desnoyers and M. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the Ottawa Linux Symposium*, volume 1, pages 209–223, Ottawa, Canada, July 2006.

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

[13] K. Drake. *XTEST Extension Protocol*. X Consortium Standard, 1994. Version 2.2.

[14] S. Gribble, G. Manku, D. Roselli, E. Brewer, T. Gibson, and E. Miller. Self-similarity in file systems. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 141–150, Madison, WI, June 1998.

[15] S. Gurun and C. Krintz. AutoDVS: An automatic, general-purpose, dynamic clock scheduling system for hand-held devices. In *Proceedings of the Conference on Embedded Systems Software*, Jersey City, NJ, Sept. 2005.

[16] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Code transformations for energy-efficient device management. *IEEE Transactions on Computers*, 53(8):974–987, Aug. 2004.

[17] J. Hom and U. Kremer. Inter-program compilation for disk energy reduction. In B. Falsafi and T. Vijaykumar, editors, *Power-Aware Computer Systems*, volume 3164 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 2004.

[18] J. Hom and U. Kremer. Inter-program optimizations for conserving disk energy. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 335–338, San Diego, CA, Aug. 2005.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, CA, Mar. 2004.

[20] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. In D. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, CA, 1993.

[21] Open Handset Alliance. Android. <http:// www.openhandsetalliance.com>.

[22] Openmoko Incorporated. Openmoko. <http:// openmoko.org>.

[23] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the Conference on Distributed Computing Systems*, Miami, FL, Oct. 1982.

[24] A. Papathanasiou and M. Scott. Energy efficient prefetching and caching. In *USENIX Annual Technical Conference*, pages 255–268, Boston, MA, June 2004.

[25] L. Roberts. ALOHA packet system with and without slots and capture. *Computer Communications Review*, 5:28–42, Apr. 1975.

[26] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000.

[27] H. Sandklef. Testing applications with xnee. Linux Journal online, Jan. 2004. <http:// www.linuxjournal.com / article / 6660>.

[28] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O — a novel I/O semantics for energy-aware applications. In *Proceedings of the Conference on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[29] Q. Zhu, F. David, C. Devaraj, Z. Li, Y. Zhou, and P. Cao. Reducing energy consumption of disk storage using power-aware cache management. In *Proceedings of the Symposium on High Performance Computer Architecture*, pages 118–129, Madrid, Spain, Feb. 2004.

[30] M. Zimet. Record extension library specification: Version 1.10 public review draft. *The X Resource*, 14(1):177–193, Feb. 1995.