Inter-program Optimizations for Disk Energy Reduction[⋆]

Jerry Hom[1] and Ulrich Kremer[1]

Rutgers University, Department of Computer Science
96 Frelinghuysen Rd, Piscataway, New Jersey 08854, USA
{jhom, uli}@cs.rutgers.edu

Abstract. Previous work has shown that intra-program optimizations, i.e., optimizations performed on individual programs in isolation, can be very effective in reducing disk energy in streaming applications. This paper investigates the potential additional benefits of inter-program optimizations where sets of programs are optimized together. Experimental results on different subsets of three streaming applications show that 7–49% additional energy savings (27.3% on average) can be obtained with negligible performance penalties using two novel inter-program optimizations, namely execution context sensitive buffer size selection and inverse barrier synchronization. These figures were obtained via physical measurements on two laptop disks.

Keywords. execution context, inverse barrier

1 Introduction

Power dissipation and energy consumption have become crucial design constraints for mobile, laptop, and desktop computers since they impact several aspects of a system, including packaging costs due to cooling requirements, operating costs, battery life time, and the overall weight of the device. Hardware, operating systems, and compiler techniques have been successful in reducing power and energy, but more work needs to be done in order to keep up with users' increasing demand for faster CPUs, faster and larger disks, and higher networking speeds.

Resource hibernation exploits the ability of devices to switch between different activity states, ranging from high activity (active and operational) to low activity (deep sleep and not operational) states[1]. As a rule of thumb, the lower the activity state, the more power and energy may be saved, but the longer it takes to transition between the low power and fully operational, active state. Each transition between activity states has a penalty, i.e., overhead, both in terms of performance and power/energy. Resource hibernation strategies identify intervals in a program's execution where a resource is not in use and therefore can be put into a low power state. For a given hibernation interval, the most

⋆ This work was partially supported by NSF CAREER award #9985050.

effective hibernation mode should be selected, and the transition into this mode should be initiated as early as possible, i.e., at the beginning of the interval. The transition out of the selected hibernation state back to the active state should be done just in time before an upcoming use, i.e., just before the end of the hibernation interval. It may not always be possible or profitable to utilize the deepest hibernation mode due to the length of the hibernation interval and the overhead of required state transitions. Most hibernation strategies have a "break even" point which typically is specified by the minimal length of the hibernation interval for which transitioning into and out of the state is profitable. Resources that have been targets for the hibernation optimization include the disk, display, memory banks, cache lines, and wireless network cards.

An energy-aware compiler can reshape a program such that the idle times between successive resource accesses are maximized, giving opportunities to hibernate a device more often, and/or in deeper hibernation states. This compilation strategy has been shown to work well in a single process environment[2–4], but may lead to poor overall results in a multiprogramming environment. In a multiprogramming setting, one program may finish accessing a resource and may direct the resource to hibernate during some time of idleness. During this time, another program may need to access the resource. In the worst case, each program alternately accesses a resource such that the resource never experiences significant amounts of idleness. In effect, one program's activity pattern interferes with another program's idle periods and vice versa. To alleviate this problem, some inter-program or inter-process coordination is necessary.

Operating systems techniques such as batch scheduling coordinate accesses to resources across active processes. Requests for a resource are grouped and served together instead of individually, potentially delaying individual requests for the sake of improved overall resource usage. In contrast to operating systems, compilers often have the advantage of knowing about future program behavior and resource requirements. Instead of reacting to resource requests at runtime, a compiler can insert code into a set of programs that will proactively initiate resource usage across the program set at execution time. This is typically beyond the ability of an operating system since it requires program modifications and knowledge about future resource usage.

In this paper, we discuss the opportunities for power and energy optimizations based on the idea of optimizing applications not in isolation, but as groups of active programs that share common resources. The disk is a primary example of such a shared resource. The original contributions of this paper are

1. The implementation of an inter-program optimization strategy through *inverse barriers* that use semaphores for inter-process communication under Linux to synchronize disk accesses. The implementation uses prefetching when profitable, assuming that disk and CPU activities may be overlapped,
2. Application-level buffer size allocation policies that consider the execution context of an application, i.e., the knowledge of other applications running at the same time in order to dynamically choose the best buffer sizes, and

3. The evaluation of the entire compiler / runtime system optimization framework through physical measurements for two commercial disk drives (4200 rpm Fujitsu MHK2060AT and 7200 rpm Hitachi E7K60) and subsets of three streaming applications (MPEG audio, MPEG video, and ftp) that were executing at the same time. The test system was a default installation of Red Hat 9 Linux, and OS-based disk prefetching remained enabled.

Relative to the intra-program optimized versions of the applications, our new inter-program optimizations save an additional 21–49% (34% on average) of disk energy on the Hitachi disk, and 7–32% (21% on average) on the Fujitsu disk. Relative to the unoptimized applications, the energy savings are 49–82% (68% on average) across both disks. Therefore, inter-program optimization is a successful and promising new optimization strategy that may be implemented effectively through a compiler / runtime library approach. These results were obtained without any user observable performance or quality of result penalties.

Although the discussed inter-program optimization strategy is based on a compiler/runtime library framework, an operating system only or a combined OS and compiler approach is also possible. A direct comparison with these other approaches is beyond the scope of this paper and is currently under investigation. Our results show that inter-program optimization is feasible and can result in significant additional disk energy savings over intra-program optimization alone.

2 Related Work

Previous work has shown that applications which read data from disk in a streamed fashion (i.e., periodic access) can utilize large disk buffers to save energy[2]. These disk buffers are local to each application and serve to increase the idle period between disk accesses. Hence each application has a unique disk access interval associated with the size of its buffer. Having longer intervals between disk accesses creates opportunities to hibernate the disk. This intra-program optimization works well for applications running in isolation, but when multiple such applications execute simultaneously, some of the intra-program optimization's effects are negated. That is, the disk idle period of one application is interrupted by a disk access from another application. This will occur whenever the intervals between accesses by multiple applications are different.

A scheduling technique, *inverse barrier*, was proposed to synchronize disk accesses across active applications[5]. This mechanism is similar to implicit co-scheduling for distributed systems[6]. Arpaci-Dusseau et al. introduce a method for coordinating process scheduling by deducing the state of remote processes via normal inter-process communication. The state of a remote process helps the local node determine which process to schedule next. The inverse barrier applies this idea to coordinate resource accesses by multiple processes on a single system.

Program cooperation can be accomplished in at least two ways: (1) delay resource access until all group members wish to use it or (2) inform all group members to use the resource immediately. The first method is similar to a *barrier* mechanism in parallel programming and can be used by programs which lack deadlines. The second method is the notion of an *inverse barrier* and can be used by programs with deadline constraints such as real-time software. For example, having "gaps" in a video stream application of more than 300 milli-seconds will reduce the overall perceived quality of the video. For audio streams, the tolerance for such "gaps" is even lower. Programs using a barrier cooperate in a passive fashion. When a program wants to access a resource, it will pause and wait until all members in its group also wish to access the resource. When all members have reached the barrier, they all may access the resource consecutively. To avoid starvation, each waiting process has a timer. If the timer expires, the process will proceed to access the resource. Programs using an inverse barrier cooperate actively to synchronize resource accesses. When a program needs to access a resource, it will notify all members in its group that the resource is in use. Other group members may decide whether accessing the resource early is benefitial. For programs with disk buffering, this has the effect of refilling a program's disk buffer earlier than necessary. In conjunction with a prefetching mechanism, this strategy can ensure that deadlines are satisfied with negligible performance impact.

There is a significant body of work with respect to scheduling processes that share resources. We are only able to discuss what we consider the most closely related works in the remainder of this section.

Weissel et al. developed Coop-I/O to address energy reduction by the disk[7]. Coop-I/O enables disk operations to be deferrable and abortable. By deferring operations, the OS may batch schedule them at a later time until necessary. The research also shows that some operations may be unnecessary and hence the abortable designation. However, the proposed operations require applications to be updated by using the new I/O function calls. In contrast, our technique utilizes compiler analysis to determine which operations should be replaced. The modification cost is consolidated to the compiler optimization and a recompile of the application.

In terms of scheduling paradigms, our work resembles basic ideas from the slotted ALOHA system[8, 9]. The essential idea is to schedule access between multiple users to a common resource (e.g. radio frequency band) while eliminating collisions or when multiple host transmit on the same frequency at the same time. For our purposes, a collision takes on almost the opposite notion of a disk request without any other requests close in time. Rather than scheduling for average utilization of the disk, optimizing for energy means scheduling for bursts of activity followed by long periods of idleness.

A form of inter-program compilation has been applied to a specific problem of enhancing I/O-intensive workloads[10]. Kadayif et al. use program analysis to determine access patterns across applications. Knowledge of access patterns allows the compiler to optimize the codes by transforming naive disk I/O into collective or parallel I/O as appropriate. The benefit manifests as enhanced I/O performance for large, parallel applications. We aim to construct a general framework suitable for developing resource optimizations across applications to reduce energy and power consumption.

3 Compiler / Runtime System Framework

In this paper, we start with the basic compilation framework as proposed by Heath, et al.[2] for intra-program optimizations. All applications are assumed to fit into main memory, avoiding any additional disk activities due to swapping. In contrast to their approach, our compiler framework initiates disk power state transitions directly through appropriate system calls, i.e., the operating system is not involved in making decisions with respect to disk hibernation for the set of optimized applications. In addition, the compiler performs inter-program optimizations by inserting code to implement inverse barriers for disk access synchronization, and to perform user-level data buffer prefetching for applications that allow overlapped CPU and disk activities. In such applications, the physical disk accesses are performed by a child process that writes into the buffer, while the corresponding application (parent) process reads from the buffer. Communication between parent and child processes is performed through semaphores. It is important to note that user-level buffer prefetching is not always possible. For example, the use of the ANSI C language STREAM I/O data type prohibits concurrent processing of and reading from a file stream. As a result, a performance penalty would be observed during the time of a buffer refill. Execution time constraints may specify the maximal length of such a “gap” in terms of milli-seconds in order to preserve the QoR (quality of result) guarantee of the application.

In the compiler framework, a user may declare a file descriptor to be `buffered` or `non-buffered`. If no annotation is specified, I/O operations for the file descriptor will not be modified by the compiler. 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. If programs use file descriptors as formal parameters, a static replacement of the orignal I/O call by a buffered I/O call is not always legal. In this case, the compiler will generate a guarded expression that selects the appropriate type of I/O operation at runtime.

To apply the buffering optimization, some characteristics of the disk must be known. This information can be obtained through runtime profiling. The goal of the profiling is to determine read and write performance characteristics of the disk, and application characteristics such as data production and/or data consumption rates. The values of these parameters are used to calculate the maximal buffer size that can be read and/or written without violating an existing performance constraint. In addition, disk speed and data consumption rate are used to determine the best placement of operations to refill the buffer with negligible performance impact on the application.

The buffer size should be maximal in order to allow the longest possible disk hibernation time between successive disk accesses. However, when a set of applications are running, the available memory for each application is restricted. The selected buffer sizes should not lead to any swapping. When compiling this set of applications, a conservative approach might divide the available memory equally among each application. This will have a poor result when only a single application is actually running. Including execution context knowledge allows the applications to truly use the available resources rather than stick to a conservative assumption. In our framework, all interesting execution contexts are known at compile time and modeled as states of a finite state machine[5]. At runtime, process communication is necessary to inform active programs about changes in their execution context. These changes are due to programs starting or ending their execution.

The profiling mechanism has two phases. The first phase measures the data consumption rate of the application's unchanged execution behavior. The unchanged behavior typically reads only the next needed block of data, processes it, then loops. This rate is used to estimate the amount of time taken to consume a buffer of a given size. The measurement also provides a lower bound estimate on the disk bandwidth. The second phase measures the observed disk bandwidth while reading a large block of data. The observed bandwidth is useful because it may be affected by the existing load on the system. The lower bound estimate is used to allocate a small buffer which will supply data to the application. This allows a forked, child process to profile the disk without interrupting the main process. Finally, a buffer size can be calculated considering parameters such as disk bandwidth, quality of result performance guarantees, available memory, execution context, and consumption rates. The actions of each phase are discussed in more detail in [2]. The overhead of our profiling strategies is negligible and does not affect the user-perceived application performance.

In this work, user-level prefetching has been added to those applications which can support it. Applications which implement disk I/O using raw file I/O operations (i.e., `read`()) are candidates for prefetching. For the applications we studied, MPEG audio and ftp can utilize such prefetching. A child process is created which sleeps until its parent's buffer is nearly consumed. The child must refill the buffer and update the buffer's new end point before the parent reaches the previous end point. The prefetch point is calculated according to the estimated time of waking up the disk, time to read the disk, and the number of other applications in its execution context. Therefore, execution context becomes necessary when disk accesses are synchronized because each application must consider all other applications which are also in queue to access the disk.

Data prefetching then relaxes execution time constraints, and accordingly the buffer sizes. In the previous model where computation and I/O were not overlapped, buffers were sized according to an execution time constraint to refill the buffer[2]. In the current model, buffers may be sized up to the available memory or allocated based on some policy of fairness. While memory sizes may vary from system to system, this work investigates interesting available memory sizes, excluding the cases of extremely large or small available memory.

The discussed approach compares favorably against a pure operating system-based, "buffered" I/O approach, in that the latter would require expensive system calls for each original application-level I/O operation. Existing OS techniques for disk hibernation use a fixed threshold of idleness before transitioning to a power saving mode. 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 Experiments and Results

This study examines three streaming applications *mpeg_play*, *mpg123*, and *sftp*. The MPEG video and audio decoders are examples of real-time applications with the need for low latency access to the disk. The audio and ftp applications use direct disk reads, which allows an overlap of CPU and disk activity, making prefetching feasible. The video application uses file descriptors of type stream I/O, prohibiting the overlap of CPU activities in the parent process and disk reading activities in the child process. We assume that the video application has a time constraint of 300 milli-seconds, i.e., can tolerate “gaps” in displaying the frames of no more than 300 milli-seconds. This becomes the amount of available time to refill the buffer, and consequently, the size constraint of the buffer itself. All three applications in the experiments have overall execution times in the range of 6.5 - 8.0 minutes. Each experiment was run three times, and the energy results reported in Figure 3 are averages.

4.1 Prototype Framework

The existing framework consists mainly of runtime libraries which implement the profiling, buffer allocation policies, disk buffering, and synchronization. Using the annotations described in Section 3, the compiler can, for example, replace the `read()` calls with `EEL_read()`, which is part of our runtime system. Currently, this replacement is done by hand. The profiling phase requires a handful of parameters about the disk such as cache size, power modes, and time to transition between modes. Some of these parameters are readily available from the disk, and some were determined through physical measurement. Disk manufacturers could trivially include all these parameters on the disk. The existing buffer allocation policies include SIZE and TIME, which are discussed in Section 4.3. SIZE is easily implemented in our system as a “divide-by-n” policy. However, TIME requires data dependent information from the profiling phase, which is only available at execution time. Since the data streams are known to us, this information is derived and hard-coded into those experiments. The disk buffering provides a virtual representation of the disk, and our runtime system mediates between the program and the physical disk. Disk reads by the program are satisfied by the disk buffer, and the runtime system will refill the buffer as necessary. So far, the only synchronization policy implemented is inverse barrier. The runtime system of each application share a semaphore as a means of communicating a “broadcast notification”. The runtime system uses a child process to monitor these notifications and alert the parent as necessary. All of these optimizations are transparent to the original program.

There are several parts which are in the process of being automated within the runtime system. The TIME allocation policy also requires execution context knowledge (e.g., consumption rate) from all running applications within a set. A communication mechanism to exchange this context will be implemented as part of the state transition module. When an application transitions to a new state, it must communicate its context information as well as determine the context information from all other applications.

Table 1. Power levels of disk states

	Disk States				
	average power (W)				
	read	idle	hibernation	standby	wakeup
Fujitsu	1.8	0.9	0.7	0.2	3.0
Hitachi	2.5	2.0	2.7	0.25	3.0

Table 2. Times and energy for disk state transitions

	State Transitions (secs / joules)	
	hibernation	wakeup
	time / energy	time / energy
Fujitsu	5.0 / 3.5	1.6 / 4.8
Hitachi	0.6 / 1.6	3.0 / 6.9

4.2 Setup

A 4200 rpm Fujitsu and 7200 rpm Hitachi laptop disk were used for the experiments. The built-in data buffer sizes (disk cache) are 0.5 MBytes for the Fujitsu and 8 MBytes for the Hitachi. The hibernation states together with their power dissipation levels are listed in Table 1. The transition costs of hibernating and waking up are listed in Table 2. The break-even point for hibernation in terms of energy savings is 17 seconds for the Fujitsu and 5.2 seconds for the Hitachi. That is, the energy consumed would be the same if either the Hitachi disk was left in idle mode for 5.2 seconds, or the disk was immediately directed to standby mode, hibernated for some seconds, and then reactivated such that it was in ready or idle mode by 5.2 seconds.

The OS on the host PC was a default installation of Red Hat 9 Linux. Linux has a disk prefetching feature, which remained enabled, but its effect on our

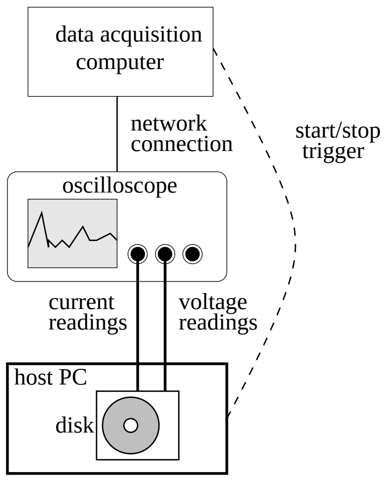


Fig. 1. Power measurement infrastructure for disk power dissipation

experiments was insignificant. This is largely because the default prefetch buffer on Linux is on the order of one hundred kilobytes whereas our runtime system's disk buffer is on the order of megabytes. Figure 1 shows the basic measurement infrastructure. Each disk was installed in the host PC, and the supply current and voltage were measured using a Tektronix TDS3014 oscilloscope with a Hall effect current probe. Measurements were reported by the oscilloscope every 20 milli-seconds and communicated to the data acquisition computer. In other words, each data point represents the average current reading for a 20 milli-second interval based on the TDS3014 sampling rate of 1.25 Giga samples per second.

4.3 Results

Experimental results are based on three streaming applications, MPEG audio (**A**), MPEG video (**V**), and ftp (**F**), and their subsets (**AV, AF, AVF**). We use the compilation strategy as proposed by Heath et al.[2] coupled with user-level buffer prefetching as the base line for our comparison. In each application set, the individual programs were optimized independently. However, Heath, et al.'s algorithm does not perform prefetching, nor does it synchronize disk accesses across the running applications. We refer to this version as the INTRA strategy, and our new proposed version as the INTER approach.

Figure 2 shows the disk current/power profile of the application set **AV** under different optimization strategies on the Hitachi disk. This figure illustrates the impact of the different optimizations on the power dissipation behavior of the sample application set. A summary across all application sets and the two disks is given in Figure 3. The disk has a supply voltage of 5 volts, and the graphs in Figure 2 show the measured supply current in amperes along the y-axes. Hence,

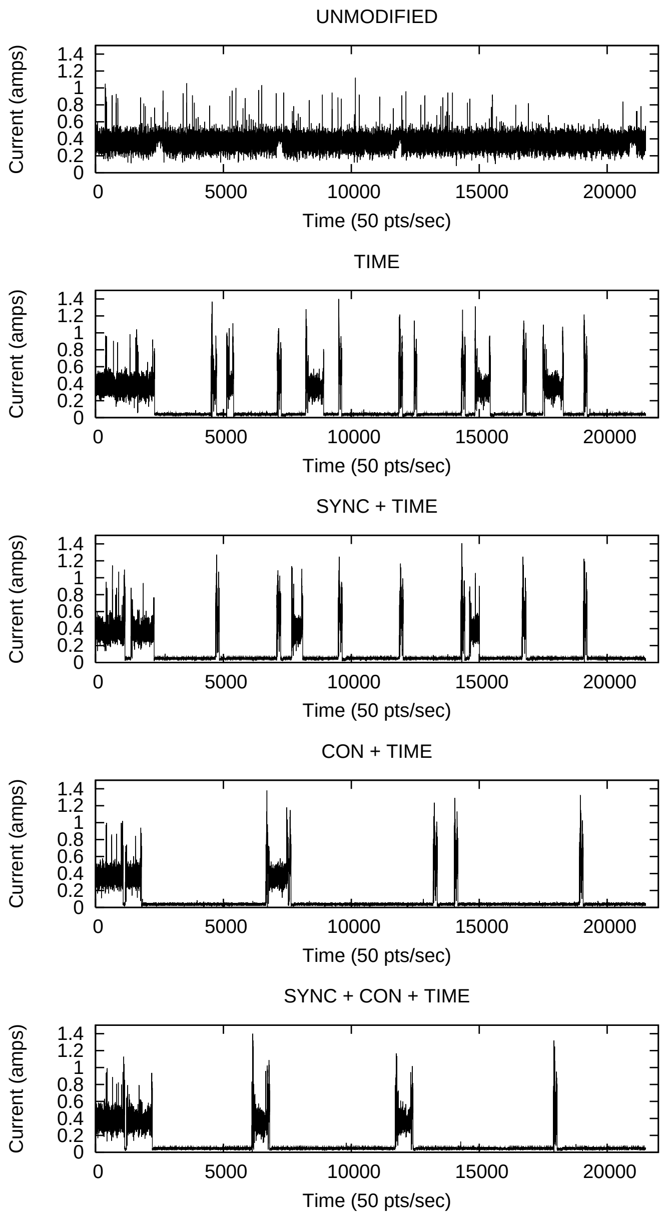
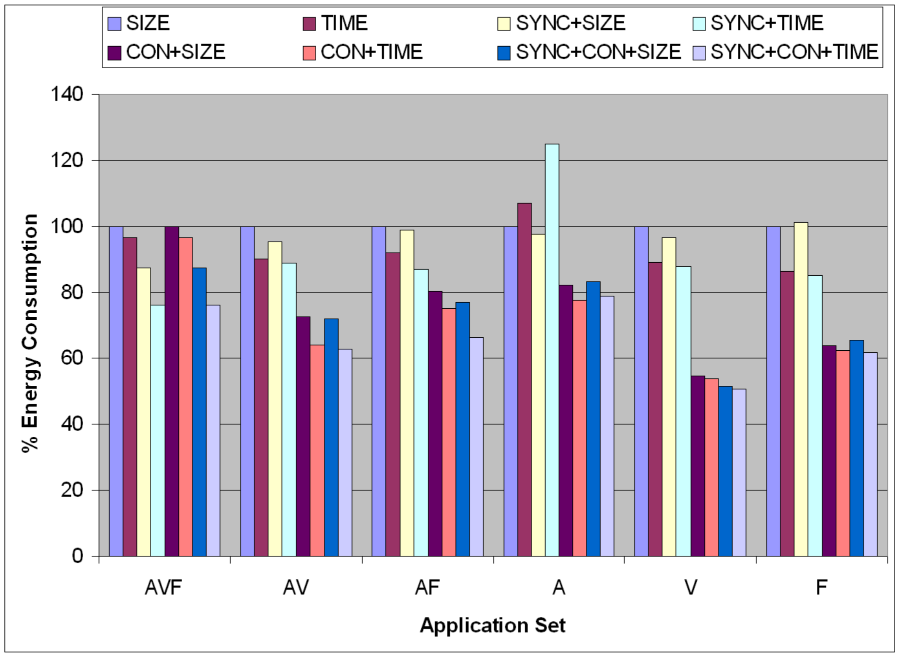


Fig. 2. Application set **AV** on the Hitachi disk (from top to bottom): UNMODIFIED — original unmodified code; TIME — with buffers allocated proportionally to each programs' data consumption rate; SYNC + TIME — adding inverse barrier synchronization to TIME; CON + TIME — adding execution context knowledge to TIME; SYNC + CON + TIME — using all three optimization strategies. The performance impact relative to the UNMODIFIED version is negligible

a 1 amp supply current results in 5 watts of power dissipation. Programs without any modifications typically exhibit the profile shown under UNMODIFIED. The disk is nearly constantly utilized and is never idle for more than a few seconds. TIME, SYNC + TIME, CON + TIME, and SYNC + CON + TIME show the effects of applying additional optimization techniques. In particular, TIME means each application in a set will have buffers allocated proportionally to its data consumption rate, resulting in all applications exhausting their buffers at the same time. TIME has no synchronization nor execution context knowledge. SYNC + TIME and CON + TIME add synchronization and execution context, respectively, to the TIME strategy. Finally, SYNC + CON + TIME uses all three optimizations.

The energy benefits of hibernation are clear when comparing UNMODIFIED and TIME. Using available memory to buffer the disk allows sufficiently long idle periods to save energy through hibernation. The effect of synchronization across applications joins together disk accesses, as shown by SYNC + TIME, which would have been non-uniformly dispersed over time. When adding execution context information, both **A** and **V** no longer use the worst case, conservative assumption that all three applications are running, but instead may use larger, proportional shares of the available memory. The essential effect of larger buffers can be seen in CON + TIME. Lastly, SYNC + CON + TIME appears to have little benefit with the additional optimization of resource synchronization, but this is actually dependant on the data streams. It turns out that the bitrate of the video stream is almost an even multiple of that of the audio stream. Hence, the buffer refill points happen to very nearly coincide. If the data streams were longer, CON + TIME would show a pattern of disk accesses starting close together and then drifting apart as time extends because the acceses are never synchronized.

Figure 3 gives a broad comparison on both the Hitachi (left) and Fujitsu (right) disks of all combinations of optimizations relative to SIZE. The first bar, SIZE, is considered a baseline optimization based on previously established results[2]. That is, applications compiled in a set will have disk buffers which are sized equally across the applications in the set. Furthermore, this implementation improves upon the established optimization by including data buffer prefetching. Related to this baseline is TIME, which assumes that the data consumption rates for each application is known. Each program's buffer size is allocated proportionally to its data consumption rate without violating the overall memory constraint. Against these baselines, applying all optimizations (SYNC + CON + SIZE, SYNC + CON + TIME) results in up to 50% additional energy savings. If synchronization is added to the baselines (SYNC + SIZE, SYNC + TIME), comparing these against all optimizations shows that up to 40% energy savings can be attributed to context knowledge. Comparing against the optimization of execution context, CON + SIZE and CON + TIME, we see that synchronization can provide up to 20% savings.



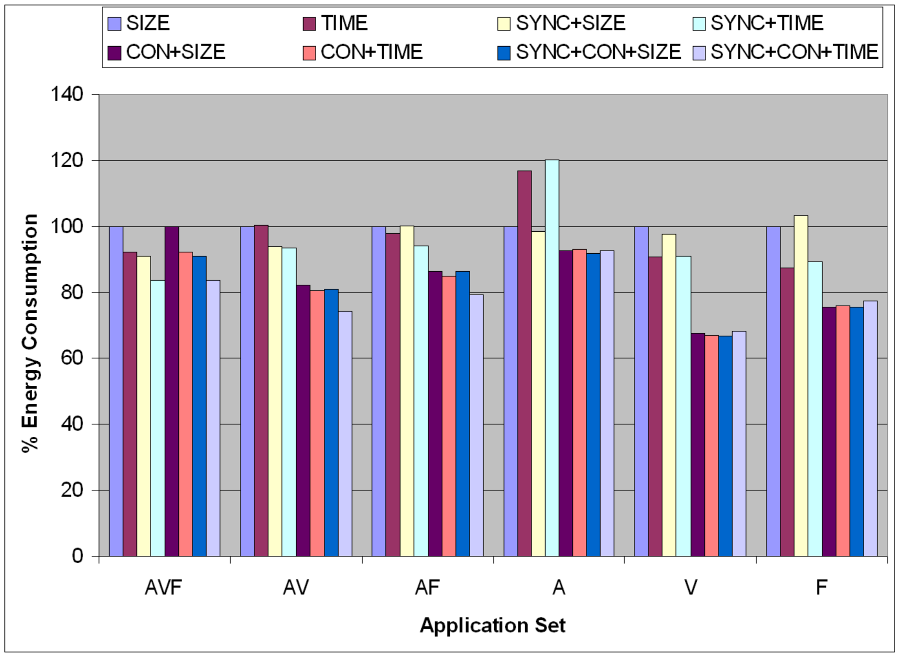


Fig. 3. Comparison of energy savings between optimization combinations and across application sets on Hitachi (top) and Fujitsu (bottom) disks. All values are % energy consumption relative to SIZE

Discussion of Results In Figure 3, there are a few significant trends to observe. In general, TIME should have better results than SIZE because the allocated buffers are proportionally maximal for all applications. Under SIZE, the application with the fastest consumption rate will dominate in terms of disk accesses, and most likely result in greater overall energy consumption. The notable exceptions occur in the application set, **A**. This is actually showing the significance of execution context. Without context knowledge, the conservative assumption meant that SIZE allocated 33% of available memory for its buffer. However, it turns out that TIME allocated only 10% of the available memory because **A**'s consumption rate is only 10% of the overall consumption rate of **AVF**. If context knowledge was used, **A** could have known it was running by itself and then used 100% of the available memory.

AVF shows the most benefits coming from synchronization. As the number of applications in a set increases, resource accesses will also increase. This application set is already the most conservative assumption for execution context, so the context results within **AVF** are identical to those without context. On the right half of the graphs, single applications show the most benefit from execution context. They are allocated 100% of available memory as buffer space. Conversely, synchronization has no use with single applications. There is an overhead associated with synchronization, but the performance penalty is usually hidden because the CPU is never overloaded. Sets consisting of two applications show the modest, cumulative energy saving effects of synchronization and context knowledge.

These trends appear in both the Hitachi and Fujitsu disks. The Hitachi results turn out better mainly because the threshold for hibernation benefit is lower (5.2 vs. 17 seconds). Hence, the Hitachi allows greater opportunities for hibernation, and our optimizations exploit it. These similar trends indicate that our profiling mechanism and optimization techniques are equally applicable among disks with widely different specifications.

A key opportunity to save power and energy is due to the fact that the available memory for buffering varies and may depend on concurrently running applications. If applications know about each other, i.e., if they know their execution context, an inter-program optimization allows the choice of the best buffer sizes across all applications for the given available memory. For example, assume that for each of three application programs there is a combination of applications and buffer sizes that will allow them to allocate a buffer size of at most 33% before inducing disk swapping. Without the execution context, each application makes a conservative assumption, leading to buffer sizes of 33%. However, if only a single application is running, context knowledge would allow that application to use 100%. The effects of context knowledge are most pronounced in the sets with a single application as shown in the right halves of Figure 3; compare the bars with and without CON.

Our experiments showed significant energy reductions of the inter-program optimization approach over an optimization approach that considers data accesses only for individual programs in isolation. Using execution context knowledge across applications provides up to 40% disk energy savings. Adding inverse barrier synchronization also contributes a potential 20% energy savings. The effect of prefetching serves chiefly to reduce or remove any performance penalties incurred by the runtime system's buffer management or the communication overhead of synchronization. These optimizations are orthogonal to each other and can be used in combination for greater energy benefits. The degree of energy savings from each optimization depends on the application set while performance is unchanged.

5 Summary and Future Work

Inter-program optimization is a promising compilation strategy for sets of programs that are expected to be executed together. The program's resource usage can be coordinated across all programs in the set, allowing additional opportunities for resource hibernation over single program, i.e., intra-program, optimizations alone. This paper discusses the potential benefits of inter-program optimizatios using the disk as the shared resource. Using 48 separate experiments, we have shown energy savings in the range of 7–49% over the intra-program optimization approach when the most aggressive optimization strategies were applied. The discussed optimization strategies included different policies for assigning buffer sizes, policies that utilize execution context knowledge, and inverse barrier synchronisation for disk access. As a point of reference, although not shown in Figure 3, energy savings over unmodified applications range from 49–82%.

Significant work is left to be done. This includes the evaluation of different strategies to identify promising sets of applications that may benefit from inter-program optimization. We are planning to instrument a collection of target systems potentially including cell phones, PDAs, laptops, and desktop systems in order to record users' typical program usage over time. We expect these results to illuminate specific usage patterns and perhaps guide the development of mobile devices. In addition, we are currently implementing dynamic context awareness in programs that are part of promising application sets. Programs use a shared interface to indicate their arrival and departure. In response, active applications may adjust their resource allocation or even change their allocation policies.

The compiler and OS have unique perspectives on key parts of the entire resource management scheme. We will experimentally explore the strengths of each strategy, resulting in the development of a resource-aware, combined compiler, runtime system, and operating system approach. A current study is trying to assess the advantages and disadvantages of a compiler-only; compiler and runtime system; OS-only; and compiler, runtime system and OS approach to inter-program resource management. The integration of the discussed inter-program optimization strategy as part of a fully automatic compiler and corresponding runtime library is currently underway.

References

1. Intel Corp., Microsoft Corp., Toshiba Corp.: ACPI implementers guide. Draft (1998)
2. Heath, T., Pinheiro, E., Hom, J., Kremer, U., Bianchini, R.: Code transformations for energy-efficient device management. IEEE Transactions on Computers **53** (2004) 974–987
3. Delaluz, V., Kandemir, M., Vijaykrishnan, N., Irwin, M., Sivasubramaniam, A., Kolcu, I.: Compiler-directed array interleaving for reducing energy in multi-bank memories. In: Proceedings of the Conference on VLSI Design. (2002) 288–293
4. Hom, J., Kremer, U.: Energy management of virtual memory on diskless devices. In Benini, L., Kandemir, M., Ramanujam, J., eds.: Compilers and Operating systems for Low Power. Kluwer Academic Publishers, Norwell, MA (2003) 95–113
5. Hom, J., Kremer, U.: Inter-program compilation for disk energy reduction. In: Workshop on Power-Aware Computer Systems, Springer-Verlag (2003)
6. Arpaci-Dusseau, A., Culler, D., Mainwaring, A.: Scheduling with implicit information in distributed systems. In: Proceedings of the Conference on Measurement and Modeling of Computer Systems. (1998) 233–243
7. Weissel, A., Beutel, B., Bellosa, F.: Cooperative I/O — a novel I/O semantics for energy-aware applications. In: Proceedings of the Conference on Operating Systems Design and Implementation. (2002)
8. Abramson, N.: The ALOHA system — another alternative for computer communications. In: Proceedings of the Fall Joint Computer Conference. (1970) 281–285
9. Roberts, L.: ALOHA packet system with and without slots and capture. Computer Communications Review **5** (1975) 28–42
10. Kadayif, I., Kandemir, M., Sezer, U.: Collective compilation for I/O-intensive programs. In: Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems. (2001)