Inter-Program Optimizations for Conserving Disk Energy*

Jerry Hom Ulrich Kremer
Department of Computer Science
Rutgers University
Piscataway, New Jersey
{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.

Categories and Subject Descriptors: D.3.3 [Frameworks]: Programming Languages—language constructs and features

General Terms: Design, Measurement

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[3]. Each transition between activity states has an overhead in terms of both 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 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. Most hibernation strategies have a "break even" point which is typically specified by the minimal length of the hibernation interval for which transitioning in to and out of the state is profitable. Primary targets for hibernation optimization include the disk, display, 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[5, 4, 6], but may lead to poor overall results in a multiprogramming environment. In a multiprogramming setting, one program may finish accessing a resource and then direct the resource to hibernate over its idle period. 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 programs that share common resources. The disk is a primary example of such a shared resource. Although the discussed inter-program optimization strategy is compiler/runtime library based, 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. However, one advantage of our framework is its transparency to the OS, which was unchanged in our experiments. The original contributions of this paper are

1. The implementation of a compiler-based, inter-program optimization strategy with *inverse barriers* that use signals and semaphores for inter-process communication 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

*Partially supported by NSF CAREER award #9985050.

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.
ISLPED'05, August 8–10, 2005, San Diego, California, USA
Copyright 2005 ACM 1-59593-137-6/05/0008 ...$5.00.

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. These results were obtained without any user observable performance or quality of result penalties. Therefore, inter-program optimization is a successful and promising new optimization strategy that may be implemented effectively through a compiler / runtime library approach.

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[5]. 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[7]. When a program accesses the disk, all other programs within its group are notified that the disk has been used, and they may decide to also use the disk. This mechanism is similar to implicit co-scheduling for distributed systems[2]. 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 coordinates resource accesses from multiple processes on a single system.

Weissel et al. developed Coop-I/O to address energy reduction by the disk[12]. Coop-I/O enables disk operations to be deferrable and abortable. By deferring operations, the OS may batch schedule them at a later time. 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 ideas from the slotted ALOHA system[1, 11]. The essential idea is to schedule accesses among multiple users to a common resource (e.g., radio frequency band) while eliminating collisions (i.e., when multiple hosts transmit on the same frequency at the same time). For our purposes, a collision takes on the opposite notion — a disk request without any other requests close in time. Optimizing the disk schedule 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[9]. 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 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.

There is an independent, simultaneous effort by Mesut et al. to study low power benefits of OS-level disk buffering and scheduling for streaming applications[10]. The implementation requires adding some kernel modules and daemons to support a low power disk scheduler which provides an interface for applications to request a buffer size and bandwidth. Their work performs almost exactly the same operation as ours, though they do not report any sense of energy savings. They have experimentally measured break-even thresholds for when their technique is beneficial.

3. COMPILER / RUNTIME SYSTEM FRAMEWORK

Inter-program optimization is based on the assumption that specific groups of programs are executing together, giving the opportunity to optimize them together. Each such group defines an execution context of applications running on the target machine at a particular time. Switching between contexts occurs when an application terminates or a new application begins. One key premise for this benefit study is considering known execution contexts. If other programs, compiled outside of the application set, are also executing, their resource access patterns may negate the benefits of the optimized application set. Alternatively, the application set may switch to a more conservative execution mode.

Our work extends the intra-program compilation framework as proposed by Heath et al.[5]. 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. Our compiler/runtime system ensures that all applications in the set will fit into main memory, thereby avoiding any additional disk activities due to swapping. In addition, the compiler performs inter-program optimizations by inserting code to implement inverse barriers for disk access synchronization, to allocate buffers with execution context, 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 main (parent) process reads from the buffer. Communication between parent and child processes is performed through semaphores and signals.

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.

To apply the buffering optimization, some characteristics of the disk must be known, which may 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 when 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 would divide the available memory equally among each application. This will have poor results when only a single application is actually running. Compiling with execution context knowledge allows the applications to truly use the available resources rather than stick to conservative assumptions. In our framework, all execution contexts are determined at compile time and modeled as states of a finite state machine[7]. At runtime, processes may passively or actively communicate about changes, such as programs starting or ending execution, in their execution context.

User-level prefetching has been added to those applications which can support it. When to prefetch 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.

4. EXPERIMENTS AND RESULTS

This study examines three streaming applications *mpeg_play*, *mpg123*, and *sftp*. 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. The data streams in the experiments have overall run times in the range of 6.5 - 8.0 minutes. More details can be found in our technical report[8].

4.1 Prototype Framework

The prototype framework consists 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 `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, while some were determined through physical measurement. The existing buffer allocation policies are SIZE and TIME. SIZE means applications in a set will have buffers which are sized equally among the applications. TIME means each application in a set will have a buffer allocated proportionally to its data consumption rate. However, TIME requires data dependent information from the profiling phase. For the TIME experiments, the information was derived and hardcoded. The disk buffering provides a virtual representation of the disk, and our runtime system mediates between the program and the disk. Disk reads by the program are satisfied by the disk buffer, and the runtime system refills the buffer as necessary. So far, the only synchronization policy implemented is inverse barrier, which is indicated by SYNC in both figures. Optimizing with execution context, labeled as CON in both figures, means that each application within a set has runtime awareness of which other set members are also running and can adjust its execution to adapt (e.g., resizing its buffer). All of these optimizations are transparent to the original program.

Some key parts are in the process of being automated within the runtime system. In particular, the TIME allocation policy 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.

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. In addition to active, idle, and off states, these disks support standby and sleep states. We used only the standby state for the power saving mode, though utilizing the sleep state would allow even greater savings. 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 by the Hitachi disk would be the same if it was either left in idle mode for 5.2 seconds or 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 experiments was insignificant. Each disk was installed in the host PC, and the supply current was measured using a Tektronix TDS3014 oscilloscope with a Hall effect current probe. Measurements were reported by the oscilloscope every 20 milli-seconds and logged 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.

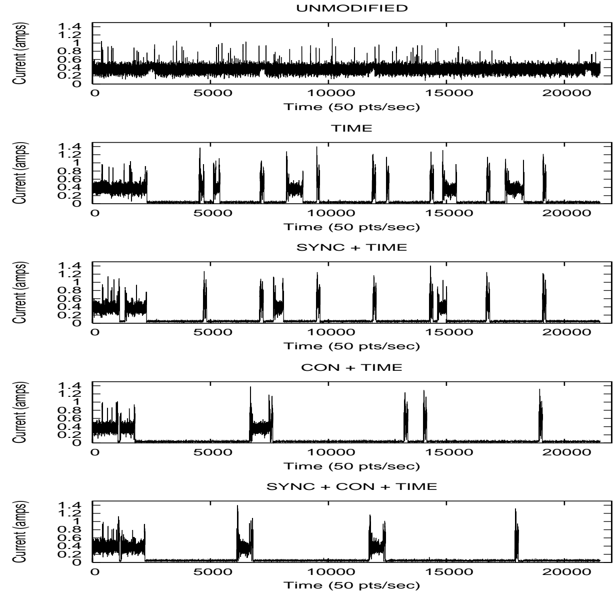

Figure 1: Disk activity profiles of application set AV using different inter-program optimization strategies on the Hitachi disk.

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**). As the base line for our comparison, we started with Heath et al.'s strategy and enhanced it by adding user-level buffer prefetching. Figure 1 contains representative graphs of 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 disk activity behavior. A summary across all application sets on the Hitachi disk is given in Figure 2. Similar results were obtained for the Fujitsu disk[8]. The disk has a supply voltage of 5 volts, and the graphs in Figure 1 show the measured supply current in amperes along the y-axes. The streaming applications without any modifications have disk activity profiles similar to that shown in UNMODIFIED. The disk is nearly constantly utilized and never idle for more than a few seconds.

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. Synchronizing disk accesses across applications, shown in SYNC + TIME, means that one application's disk access does not interfere with another application's idle period. CON + TIME shows the effects of adding execution context information. Both **A** and **V** now use larger, proportional shares of the available memory instead of assuming the worst case, conservative assumption that all three applications are running. SYNC + CON + TIME appears to have little benefit compared to CON + TIME, but this is actually dependent on the data streams. It turns out that the bit rate of our video stream is almost an even multiple to that of our audio stream. Hence, the buffer refill points are very nearly coincident. If the data streams were longer, CON + TIME would show a pattern of disk accesses starting close together and then drifting apart over time because the acceses are never synchronized.

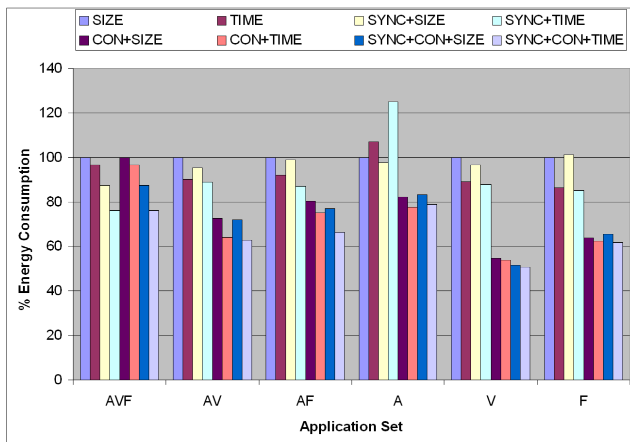

Figure 2: Comparison of energy savings between optimization combinations and across application sets on Hitachi disk. All values are % energy consumption relative to SIZE.

Figure 2 gives a comparison on the Hitachi disk of all combinations of optimizations relative to SIZE. The first bar, SIZE, is the baseline optimization based on previously established results[5]. Related to this baseline is TIME, which assumes that data consumption rates for all applications are 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 only synchronization is added to the baselines (SYNC + SIZE, SYNC + TIME), up to 20% energy savings can be realized. Comparing only the optimization of execution context, (CON + SIZE, CON + TIME), we see up to 40% energy savings.

Discussion of Results

In Figure 2, there are a few significant trends to observe. In general, the TIME optimizations should have better results than SIZE because the allocated buffers are proportionally maximal for all applications. The notable exception occurs in the application set, **A**. This result actually shows 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**. With context knowledge, **A** could know it was the only running application and hence use 100% of available memory.

AVF shows the most benefits from synchronization. As the number of concurrent applications increases, resource accesses will also increase, raising the likelihood of interference between resource accesses and idle periods. However, this application set is 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 graph, single applications show the most benefit from execution context. They are allocated 100% of available memory as buffer space. Conversely, synchronization is useless with single applications. Sets consisting of two applications show cumulative energy saving effects of both synchronization and context knowledge.

These trends also appear in the Fujitsu disk[8]. The Hitachi results turn out somewhat better mainly because the threshold for hibernation benefit is lower (5.2 vs. 17 seconds). Hence, the Hitachi has more 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.

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

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 synchronization for disk access. As a point of reference, although not shown in Figure 2, energy savings over unmodified applications range from 49–82%.

6. REFERENCES

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

[2] 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, June 1998.

[3] Intel Corp., Microsoft Corp., and Toshiba Corp. ACPI implementers' guide. Draft, February 1998.

[4] V. Delaluz, M. Kandemir, N. Vijaykrishnan, M.J. 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, January 2002.

[5] 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, August 2004.

[6] J. Hom and U. Kremer. Energy management of virtual memory on diskless devices. In L. Benini, M. Kandemir, and J. Ramanujam, editors, *Compilers and Operating systems for Low Power*, pages 95–113. Kluwer Academic Publishers, Norwell, MA, 2003.

[7] J. Hom and U. Kremer. Inter-program compilation for disk energy reduction. In *Workshop on Power-Aware Computer Systems*, December 2003.

[8] J. Hom and U. Kremer. Inter-program optimizations for conserving disk energy. Technical Report DCS-TR-578, Dept of Computer Science, Rutgers University, May 2005.

[9] I. Kadayif, M. Kandemir, and U. Sezer. Collective compilation for I/O-intensive programs. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, August 2001.

[10] Ö. Mesut, B. van den Brink, J. Blijlevens, E. Bos, and G. de Nijs. Hard disk drive power management for multi-stream applications. In *Proceedings of the Workshop on Software Support for Portable Storage*, March 2005.

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

[12] 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*, December 2002.