Inter-Program Optimizations for Conserving Disk Energy

Jerry Hom  Ulrich Kremer
Department of Computer Science
Rutgers University
Piscataway, New Jersey
fhom, ulig@cs.rutgers.edu

ABSTRACT
Previous work has shown that inter-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 74.9% additional energy savings (7.3% on average) can be obtained with negligible performance penalties using two novel inter-program optimization techniques, namely execution context sensitive buffer size selection and inverse barrier synchronization. These gains were obtained via physical measurements on two laptop disks.

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 system, 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 demands for faster CPUs, faster and larger disks, and higher networking speeds.

Resource hibernation exploits the ability of devices to switch between high activity (active and operational) to low activity (sleep and not operational) states [6]. 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 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 practical 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 minimum length of the hibernation interval for which transitioning into and out of the state is pro table. 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 minimized, 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 [6, 5, 7], but may lead to poor overall results in a multiprogramming environment. In a multiprogramming setting, one program may wish 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 sign cant amounts of idleness. In one case, a 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 system techniques such as batch scheduling coordinate access 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 system, compiler techniques can 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 prime 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 inter-program entation uses predischarging 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 took place by the use of physical memory in tests for two consumer electronics devices (Fujitsu M H K 2060AT and 7200 rpm Hitachi 7K6) and subsets of three streaming applications (MPEG audio, MPEG video, and Zip) that were executing at the same time. The test system was a default installation of Red Hat 9 Linux, and OS-based disk prefetching was enabled.

Relative to the intra-program optimization versions of the applications, our new inter-program optimization savings an additional 21(49%) from disk energy on the Hitachi disk, and 7(32%) from on the Fujitsu disk. Relative to the un-optimized applications, the energy savings are 48(82%) from (68% on average) across both disks. Therefore, inter-program optimization is a successful and promising new optimization strategy that may be in place either entirely through a compiler/runtime library approach. These results were obtained without any user observable performance or quality of result penalties.

A though the discussed inter-program optimization strategy is compiler/runtime library-based, an operating system only or a compiler/runtime library approach is also possible. A place-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 in-program optimization alone.

2. RELATED WORK

Previous work has shown that applications which read data from disk in a stream-based fashion (i.e., periodic access) can utilize large disk buffer to save energy. 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 inter-program optimization works well for applications running in isolation, but when multiple such applications execute simultaneously, some of the inter-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 barriers, was proposed to synchronize disk accesses across active applications[2]. This mechanism is similar to the plctic co-scheduling for distributed system[3]. An apacilD useuse 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 provides 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 which have deadline constraints such as real-time software. For example, having a gap in a video stream application more than 300 ms will reduce the overall perceived quality of the video. For audio streams, the tolerance for such a gap is even lower. Program cooperation using a barrier mechanism in a passive fashion is when a group 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. Program cooperation using an inverse barrier cooperates 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 is necessary. If not, the group's 'di line' is 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.

Weselewski et al. developed Coop-I/O to address energy reduction by the disk[10]. Coop-I/O enables disk access to be deferred and abortable. By deferring operations, the O.S may batch 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 recomputation of the application.

In terms of scheduling paradigm, our work resembles basic ideas from the sketched ALOHA system[1, 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 a multiple host transmit on the same frequency at the same time. For our purposes, a collision takes on all other requests close in time; rather than deferring for average utilization of the disk, operating 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[8]. Kadayif et al. use program analysis to detect disk access patterns across applications. It now leverages of pattern allows the compiler to optimize the codes by transforming normal disk I/O into collective or parallel I/O as appropriate. The basic aim is to enhance I/O performance for large, parallel applications. We aim to construct a general framework 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.[6] for inter-program optimizations. All applications are assumed to be mapped 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 regard 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 disk physical 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 operation. Execution time constraints may specify the total length of such a gap.*
In term s of milliseconds in order to preserve the QoR (quality of result) guarantee of the application.

In the compiler framework, a user may declare a \texttt{le} descriptor to be buffered or non-buffered. If no annotation is specified, I/O operations for the \texttt{le} descriptor will not be modified by the compiler. The compiler propagates \texttt{le} descriptor attributes across procedure boundaries, and replaces every original I/O operation of the \texttt{le} descriptor in the program with calls to a corresponding bu er I/O runtime library. If program uses \texttt{le} descriptors as \texttt{b}om al param eters, a static replace ent of the original I/O call by a bu er 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 bu er 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 the read and write performance characteristics of the disk, and application characteristics such as data production and/or data consumption patterns. The values of these parameters are used to calculate the bu er \texttt{le} size that can be read and/or written without violating an existing performance constraint. In addition, disk speed and data consumption pattern rate are used to determine the best placement of operations to re-allocate the bu er with negligible performance impact on the application.

The bu er size should be \texttt{max} in order to allow the lowest possible disk hibernation time between successive disk accesses. However, when a set of applications are running, the available \texttt{mem} ory for each application is restricted. The selected bu er sizes should not lead to any swapping. When compiling this set of applications, a conservative approach is tight divide the available \texttt{mem} ory 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 \texttt{machine}(2). At runtime, the compiler 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 pattern rate of the application's unchanged execution behavior. The unchanged behavior typically reads only the next needed block of data, processes it, and then loops. This rate is used to estimate the amount of time taken to consume a bu er 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 bu er which will supply data to the application. This allows a forked child process to probe the disk without interrupting the main process. Finally, a bu er size can be calculated considering parameters such as disk bandwidth, quality of result performance guarantees, available \texttt{mem} ory, execution context, and consumption pattern. The actions of each phase are discussed in more detail in [6]. 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 benefit from it. Applications which rely on disk I/O using raw I/O operations (i.e., \texttt{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 bu er is nearly consumed. The child is used to read the bu er and update the bu er'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, \texttt{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 bu er sizes. In the previous model where compensation and I/O were not overlapping, bu er size were sized according to an execution time constraint to \texttt{min} the bu er size. In the current model, bu er size may be sized up to the available \texttt{mem} ory or allocated based on specific policies. While \texttt{mem} ory sizes in me vary from system to system, this work investigates interesting available \texttt{mem} ory sizes, excluding the cases of extremely large or small available \texttt{mem} ory.

The discussed approach compares favorably against a pure operating system-based, \texttt{bu er} 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 \texttt{les} are accessed with a large stride, or accessed irregularly.

We are currently investigating compiler analyses and optimizations to predict \texttt{les} access into a "dense" bu er, and to determine the working set of active \texttt{le} blocks that should be \texttt{bu er}ed for the non-sequential \texttt{les} accesses.

4. EXPERIMENTS AND RESULTS

4.1 Prototype Framework

The existing framework consists of a library which implements the profiling, bu er allocation policies, disk bu ering, and synchronization. Using the annotations described in Section 3, the compiler can, for example, replace the \texttt{read()} calls with \texttt{SEE} \texttt{read()}, which is part of our runtime system. Currently, this replacement is done by hand. The profiling phase requires a handful of parameter eters about the disk such as cache size, \texttt{mem} ory size, and \texttt{time} to transition between \texttt{mem} ory sizes. Some of these parameters are readily available from the disk, and some eters were determined through physical measurement. D \texttt{isk} \texttt{m} anufacturers could trivially store all of these parameters on the disk circuitry. The existing \texttt{bu er allocation policies} include \texttt{SIZE} and \texttt{TIME}, which are discussed in Section 4.3. \texttt{SIZE} is easily pl emented in our system as a \texttt{divide-by-n} policy. However, \texttt{TIME} requires data dependent information from the profiling phase, which is only available at execution time. Since the data stream are known to us, this information is derived and hard-coded into those experiments. The disk \texttt{bu ering} is 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 \texttt{bu er}, and the runtime system relays the \texttt{bu er} to the program as necessary. So far, the only synchronization policy in pl emented in our system is the \texttt{I/O} or \	exttt{broadcast} notification. The runtime system also uses a \texttt{broadcast} as a means of communicating a "broadcast notification." The runtime system also uses a \texttt{mechanism} as a means of communicating a "broadcast notification."
Table 1: Disk states, their power levels, and wakeup and hibernation times and energy.

<table>
<thead>
<tr>
<th></th>
<th>disk states</th>
<th>average power (W)</th>
<th>state transitions (secs/joules)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>hibernation</td>
<td>wakeup</td>
</tr>
<tr>
<td>Fujitsu</td>
<td>0.7</td>
<td>3.0</td>
<td>1.8</td>
</tr>
<tr>
<td>Hitachi</td>
<td>2.7</td>
<td>3.0</td>
<td>2.5</td>
</tr>
</tbody>
</table>

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, and the costs of hibernating and waking up are listed in Table 1. 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 5.2 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 remains enabled, but its effect on our experiments was negligible. This is largely because the default prefetcher 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 mili-
seconds and communicated to the data acquisition computer. In
other words, each data point represents the average current reading
for a 20 mili-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 DLP (F), and their sub-
sets (A, V, A, A, F, A, V, F). We use the compilation strategy as pro-
posed by Heath et al.[6] coupled with user-level bu er prefetch-
ing as the base line for our comparisons. 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 re-
fer 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 graph illustrates the impact of different optimization tech-
niques on the power distribution behavior of the sample application set. A
sum m ary 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 am-
peres along the y-axes. Hence, a 1 amp supply current results
in 5 watts of power dissipation. Programs with no memory allocation 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 applica-
tion in a set will have its own bu er allocated proportionally to its
data consumption rate, resulting in all applications exhausting
their b uers 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 UN-
MODIFIED and TIME using available memory to buffer the disk
allowing sufficiently long idle periods to save energy through hib-
ernation. 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 exec-
ution 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 b uers 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 dependent on the
data stream s. It turns out that the bitrate of the video stream is
almost an even multiple of that of the audio stream. Hence,
the bu er re quirement happens to very nearly coincide. If the data
stream s were longer, CON + TIME would show a pattern of disk
accesses starting close together and then drifting apart as the e
extends because the accesses 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 bar, SIZE, is considered a baseline optimization
based on previously established results[6]. That is, applications con-
tained in a set will have disk b uers which are sized equally
across the applications in the set. Furthermore, this implies that
optimization is proved upon the established optimization by including
data bu er prefetching. Related to this baseline is TIME, which
assumes that the data consumption rate for each application is
known. Each program’s bu er size is allocated proportionally to its
data consumption rate within a disk utilization constraint. Against these baselines, applying optimizations
\( \text{SYNC + CON + SIZE, SYNC + CON + TIME} \) results in up
to 50% additional energy savings. If synchronization is added to
the baselines \( \text{SYNC + SIZE, SYNC + TIME} \), comparing these
against all optimizations shows that up to 40% energy savings can
be attributed to context knowledge. Comparing against the opti-
mization of execution context \( \text{CON + TIME} \), we see that
clock synchronization can provide up to 20% savings.

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 bu er s are proportionally more aligned for all applications.
Under SIZE, the application with the highest consumption rate
will dominate in terms of disk accesses, and thus most likely result
in greater overall energy consumption. The notable exceptions
occur in the application set, A. This is actually showing the sig-
nificant impact of execution context. Without context knowledge, the
conservative assumption that SIZE allocated 33% of available
memory for each program. However, it turns out that TIME allo-
cated only 10% of the available memory because A’s consumption
rate is only 10% of the overall consumption rate of AVF.

AVF shows the most benefit coming from synchronization. As
the number of applications in a set increases, resource accesses will
also increase. This application set is already the most consen-
trating application set for execution context, so the consumption
reads 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 synchron-
zation, but the performance penalty is usually hidden because the
CPU is never overloaded. Sets consisting of two applications show the most benefit, cumulative energy savings e ects of synchroniza-
tion 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 (6.2 vs. 17 seconds). Hence, the Hi-
tachi allows a greater opportunity for hibernation, and our opti-
mizations exploit this. These similar trends indicate that our pro-
ning 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 buffer varies and may depend on
currently 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 each of the three application programs has a
combination of applications andbuffer sizes that will allow them
to allocate a buffer size of 33% before inducing disk swapping.
Without the execution context, each application makes a conser-
vative 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 e ects of context knowl-
dge are more pronounced in the sets with a single application as
shown in the right half of Figure 3; compare the bars with and
without CON.

Our experiments showed significant energy reductions of the inter-
program optimization approach over the optimization approach
that considers data accesses only for individual programs in iso-
lation. 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 e ects of prefetching serve chiefly to reduce or remove any
performance penalties incurred by the run-time system’s buffer
management or the communication overhead of synchronization.
The optimization is orthogonal to each other and can be used
in combination for greater energy benefits. The degree of energy


The discussed optimization strategies included different policies 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 allocation over single program, i.e., intra-program, optimizations alone. This paper discusses the potential benefits of inter-program optimization 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 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 ranging from PDAs, laptops, to desktop systems in order to record their program usage over time. 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 system. 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.

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 allocation over single program, i.e., intra-program, optimizations alone. This paper discusses the potential benefits of inter-program optimization 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 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 ranging from PDAs, laptops, to desktop systems in order to record their program usage over time. 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 system. 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.

6. REFERENCES

[2] A anonym ous. Inform at ion w i th held t o al low bl ind r evi ews.