Inter-program Compilation for Disk Energy Reduction[⋆]

Jerry Hom and Ulrich Kremer

Rutgers University, Department of Computer Science,
Piscataway, NJ 08854, USA,
{jhom, uli}@cs.rutgers.edu

Abstract. Compiler support for power and energy management has been shown to be effective in reducing overall power dissipation and energy consumption of individual programs, for instance through compiler-directed resource hibernation and dynamic frequency and voltage scaling (DVS). Typically, optimizing compilers perform intra-program analyses and optimizations, i.e., optimize the input program without the knowledge of other programs that may be running at the same time on the particular target machine. In this paper, we investigate the opportunities of compiling sets of programs together as a group with the goal of reducing overall disk energy. A preliminary study and simulation results for this inter-program compilation approach shows that significant disk energy can be saved (between 5% and 16%) over the individually, disk energy optimized programs for three benchmark applications.

1 Introduction

Handheld computers have come a long way from just being a sophisticated address book and calendar tool. Handheld computers or pocket PCs feature rather powerful processors (e.g. 400MHz XScale), 64MB or more of memory, wireless Ethernet connections, and devices such as cameras, speakers, and microphones. They are able to run versions of standard operating systems such as Linux and Windows. However, as compared to their desktop PC counterparts, pocket PCs have significantly less resources, in particular power resources, and less computational capabilities. While pocket PCs will evolve further in terms of their resources and capabilities, they will always be more resource constrained than the comparable desktop PCs, which will evolve as well. As a result, users of pocket PCs have to be more selective in terms of the programs that they wish to store and execute on their handheld computer. Such a program set may include a web browser, an mpeg player, communication software (e.g. ftp), a voice recognition system, a text editor, an email tool, etc. Typically, a user may only run a few of these programs at any given point in time. Figure 1 shows an example of possible subsets of programs executing simultaneously on a handheld PC or PDA. Only program combinations that occur frequently or are considered important are represented as explicit states. For example, running a web browser,

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

an audio player, and an email program at the same time may occur frequently enough to promise a benefit from inter-program analysis and optimizations. State transitions are triggered by program termination and program activation events. The state marked "???" is a catch-all state that allows internal transitions and represents all combinations of simultaneous program executions that are not considered interesting enough to be analyzed and optimized as a group. The graph shows the underlying assumption of our presented work, namely that a typical usage pattern of a handheld PC or PDA can be characterized by a limited number of program subsets where the programs in a subset are executed together. This makes optimizing particular states or program combinations feasible. The graph can be determined through program traces or other means. Techniques and strategies to determine such a graph are beyond the scope of this paper.

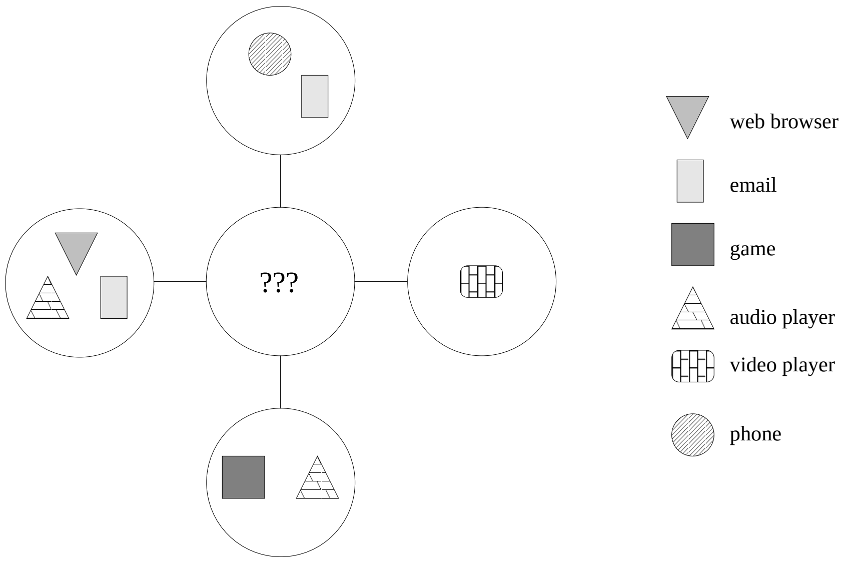

Fig. 1. Example finite state machine. Nodes represent subsets of applications executing simultaneously. Transitions are triggered by program activation and termination events. The catch-all state marked "???" allows internal transitions and represents all non-interesting program combinations

A compiler is able to reshape a program's execution behavior to efficiently utilize the available resources on a target system. Traditionally, efficiency has been measured in terms of performance, but power dissipation and energy consumption have become optimization goals in their own right, possibly trading-off power and energy savings for performance. One effective technique to save power and energy is resource hibernation which exploits the ability of devices to switch between different activity states, ranging from a high activity (active) to a deep sleep state. As a rule of thumb, the lower the activity state, the more power and energy may be saved, but the longer it will take to bring back the device into the fully operational, active state. Each transition between an activity state has a penalty, both in terms of performance and power/energy.

An energy-aware compiler optimization 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[1, 2], but may lead to poor overall results in a multiprogramming environment. In a multiprogramming setting, one program may finish accessing a resource and direct the resource to hibernate during some time of idleness. During this time, a second program needs 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 have often 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 investigate the potential benefits of an inter-program optimization strategy for disk power and energy management. This paper focuses on a compiler/runtime library based approach, although an OS only, or a combined OS and compiler approach is also possible. An initial study of a compiler only vs. OS only strategy for inter-program optimizations is currently underway.

By considering multiple programs, the compiler applies a synchronization optimization which we call *inverse-barrier*. 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[1]. 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.

Simulation experiments using physical traces of three intra-program optimized applications show significant energy savings when applying the inverse-barrier optimization. The inverse-barrier also proves more effective at saving energy and maintaining performance than using barrier synchronization.

2 Related Work

This research is related to a few OS level techniques. In order for inter-program compilation to be useful, it should apply optimizations which span across applications. The initial idea began from the notion of a programming mechanism called a barrier to delay disk accesses in order to cluster them as well as increase the idle time between accesses. To be useful, the OS must support such a programming paradigm with a scheduling policy. Indeed, co-scheduling is one example and a well-known technique for scheduling processes in a distributed group at the same time[3]. One aspect is to schedule associated processes at the same time thereby letting processes make progress within their scheduled timeslot. Since our work relies on idleness to save energy, we desire processes to synchronize by scheduling their resource accesses together and maximizing idleness.

Our mechanism for synchronizing accesses, inverse-barrier, is similar to implicit coscheduling for distributed systems[4]. 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. Inverse-barrier applies this idea to coordinate resource accesses by multiple processes on a single system.

More recently, Weissel et al. developed Coop-I/O to address energy reduction by the disk[5]. 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 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, this work resembles basic ideas from the slotted ALOHA system[6, 7]. 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[8]. 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 Intra- vs. Inter-program Optimizations

Handheld computing devices may be designed as general purpose, yet each user may desire to run only a certain mix of applications. If this unique set of applications remains generally unchanging, compiling the set of applications together with inter-program scheduling can enhance performance and cooperation by synchronizing resource usage. A further goal is to show how new energy optimizations may be applied for resource management.

Consider a scheduling paradigm across programs on a single processor. The proposed optimizations augment the paradigm with user-transparent barrier and *inverse*-barrier mechanisms to resemble thread scheduling. Barrier semantics enforce the notion that processes or threads (within a defined group) must pause execution at a defined barrier point until all members of the group have reached the barrier. The notion of an inverse-barrier applies specifically to resources. That is, when a process or thread reaches an inverse-barrier (e.g. by accessing a resource), all members of the group are notified to also access the resource. Synchronizing resource accesses eliminates any pattern of random access and allows longer idle periods where the resource may be placed in a low power hibernation mode.

An example of an intra-program optimization is a transformation to create large disk buffers in memory thereby increasing the disk's idle time for hibernation. While application transformations have been shown to benefit applications executed in isolation[1], running such locally optimized programs concurrently squander many of the benefits because the access pattern from each process disrupts the idle time of the resource. This intra-program optimization considers each program by itself while an inter-program optimization now considers all programs in a group and augments them to cooperate in synchronizing accesses to a resource.

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 has the notion of an inverse-barrier and can be used by programs with deadline constraints such as real-time software.

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 reach the barrier, then 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 to also access the resource immediately. This has the effect of refilling a program's disk buffer earlier than necessary which ensures that deadlines are satisfied. This synchronization mechanism can be communicated via signals among all processes. Only those processes with an appropriately included signal handler will follow suit in accessing the resource. Uninterested processes may simply ignore the signal.

The signal mechanism is also used in our current compilation framework to inform active programs about other active, simultaneously executing programs. The compiler generates signal handling code within each program that implements state transitions between interesting groups of applications. Each time a program is about to terminate, it sends a signal to inform other active processes about its termination event. The appropriate signal handlers in the remaining active programs will then make the corresponding state transition. Each time a program begins execution, it sends a signal to inform other active programs about its presence. In return, active programs will send a signal informing the ”new” process about the state that they are in, i.e., inform the program about its current execution group. This way, a program is aware of its group execution context, and can perform appropriate optimizations in response to inverse-barrier signals.

4 Experiments and Results

This benefit analysis builds upon previous work and examines three streaming applications *mpeg_play*, *mpg123*, and *sftp*. The MPEG video and audio decoders are examples of real-time applications where they must have low latency access to the disk. They cannot afford to wait for other applications before accessing the disk. On the other hand, ftp is a silent process, mostly invisible to the user, and can tolerate pauses with the understanding that throughput performance is traded off with energy savings.

From these three applications, there are three experiments with interesting results: 1) all three applications, 2) video with audio, 3) audio with ftp. Combining video with ftp is expected to produce similar results as (3). Although the original experiments operated on the same file, each run produced slightly different traces because of the dynamic nature of the disk profiling at program startup. However, the variance from each set of runs was minor and demonstrates the stability of the profiling strategy. All experiments used disk traces from Heath et al.[1] for hand simulating the behavior of these programs executing at the same time with and without inter-program optimizations applied.

The disk traces were modified to better simulate the more interesting, steady state conditions while the applications are running simultaneously. The duration of the traces from the three programs vary considerably. For example, the trace for *mpg123* lasted 425 seconds while *mpeg_play* and *sftp* were 106 and 232 seconds, respectively. The shorter traces were extended to be roughly time equivalent to *mpg123*. Since each program was optimized to produce periodic disk accesses, extending the execution time is merely a matter of using larger data files. Hence, the traces were extended by "copying and pasting" multiples of disk access periods.

A second modification deals with the buffer sizes. The buffer size is calculated at runtime after some profiling steps. A prudent calculation would divide the buffer size by n where n is the number of applications compiled with this disk buffer optimization. Otherwise, if all applications used its maximums buffer size, thrashing may occur when such applications are actually executed together. Thus, the disk access intervals for each application was divided by either 2 or 3 for the experiments.

These particular applications lightly stress the CPU, and the experiments assume that the CPU meets all deadlines (e.g. decoding frames) for all applications running simultaneously. The CPU can decode all frames of video and audio while copying blocks of data for file transfer without degrading performance. Degraded performance might result in dropped frames. However, the physical disk is constrained to serving one process at a time. Thus, if more than one process issues a disk request at the same time, they will be queued and interleaved. In effect, disk access time by processes cannot be overlapped and hidden.

The results of these experiments are closely tied to system parameters. A different disk will change the mix of thresholds in determining when to switch power states, but the essential premise is the potential to save energy by utilizing low power states. Table 1 summarizes the parameters measured on a real disk[1]. When transitioning from *idle* to *standby*, the disk spends 5.0 seconds in the *transition* state. When waking up from *standby* to *idle* or *read*, the disk takes 1.6 seconds. The idleness threshold at which transitioning to *standby* becomes profitable is 10.0 seconds. That is, when the system knows the next disk access is greater than 10.0 seconds, the system should tell the disk to transition to *standby*.

Table 1. Disk states and power levels

Disk States	Power (W)
wakeup	3.0
read	1.8
idle	0.9
transition	0.7
standby	0.2

The first experiment combines all three applications. The respective disk access periods are $P_{mpeg\_play} = 11.7$, $P_{mpg123} = 13.7$, and $P_{sftp} = 23.5$; all times in seconds. Figure 2 shows the disk access traces of each application. The top row of Figure 3 shows an overlay of all disk accesses. The bottom row shows the synchronization when using inverse-barrier scheduling. Net energy savings is 5.4%.

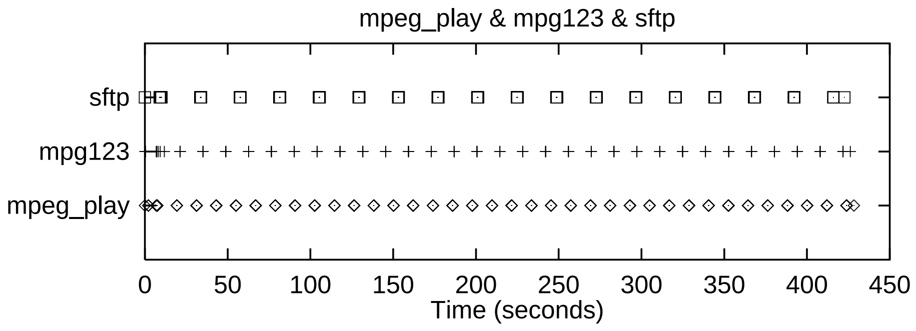

Fig. 2. Disk access traces for *mpeg_play*, *mpg123*, and *sftp*

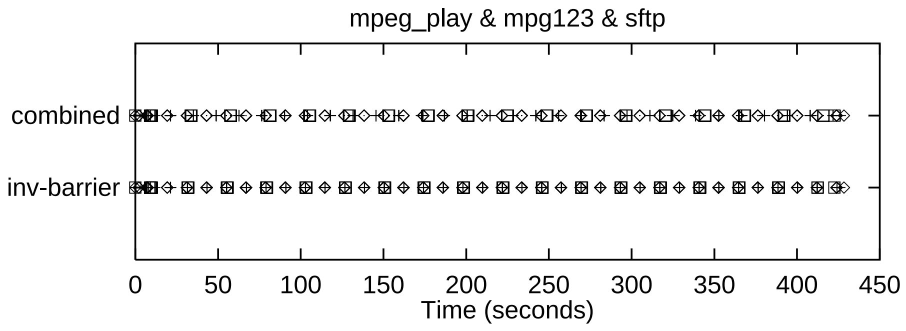

Fig. 3. Running all three applications simultaneously. Comparison of disk access patterns with inverse-barrier optimization for synchronization. Inverse-barrier scheduling saves 5.4% energy

The second experiment combines *mpeg_play* with *mpg123*. The disk access periods are $P_{mpeg\_play} = 17.6$ and $P_{mpg123} = 20.6$. Figure 4 illustrates the idleness interference patterns of just two applications vs. inverse-barrier synchronization. A similar pattern can be seen with the third experiment in Figure 5. Here, the disk access periods are $P_{mpg123} = 20.6$ and $P_{sftp} = 35.2$. The energy saved in these two experiments are 15.9% and 9.8%, respectively.

There is a key difference between the inverse-barrier and barrier mechanisms. An OS may employ a barrier mechanism to delay resource accesses for appli-

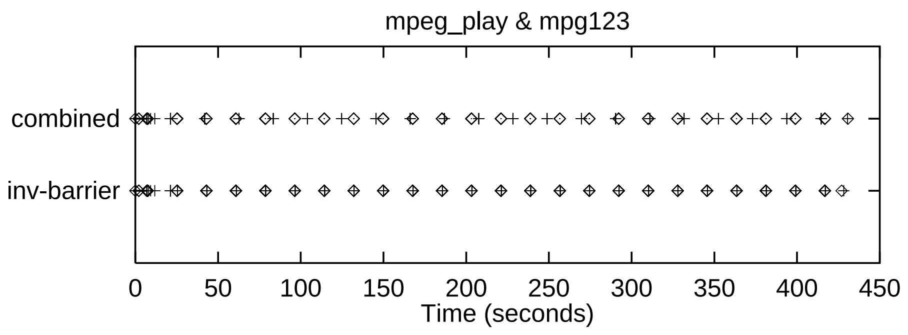

Fig. 4. Comparison of two applications' (*mpeg_play*, *mpg123*) disk access patterns. Inverse-barrier scheduling saves 15.9% energy

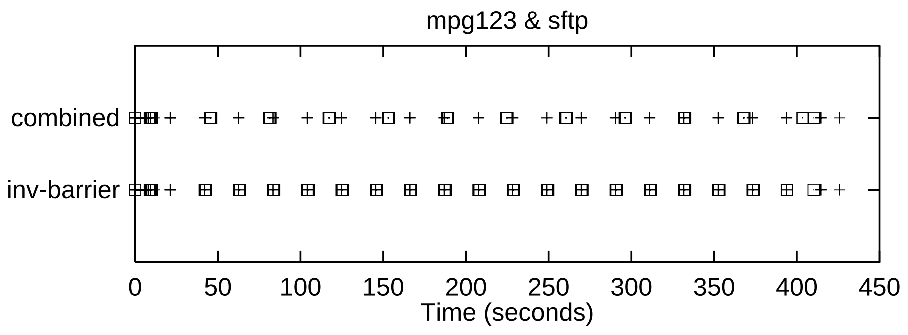

Fig. 5. Comparison of two applications' (*mpg123*, *sftp*) disk access patterns. Inverse-barrier scheduling saves 9.8% energy

cations which can tolerate such latencies. For example, *sftp* has few constraints about deadlines since it operates by best effort semantics over an unreliable network. If an OS uses this assumption to schedule *sftp* with barriers at disk accesses (i.e., delaying until next access by another application), there will certainly be a performance delay. This mode of operation can still save energy by batching the disk access but also depending on how much delay is involved. The difference with inverse-barrier is the pre-emptive action to ensure that buffers are always sufficiently full. At every synchronized disk access, each process can check the data capacity of its buffer and decide whether to read more data. Another optimization during the buffer check might compute the differential between resource access periods. For instance, if a process has a resource access period more than twice as long as the current period, it can afford to skip every other resource access and maintain a non-empty buffer.

The last experiment explores the behaviors of barrier scheduling; Figure 6 illustrates the difference. In terms of execution time, *sftp* finishes over two minutes later using the barrier vs. the inverse-barrier. Compared to the baseline of running all applications together under normal scheduling, the barrier method expends 2.4% more energy. Barrier scheduling is only slightly more expensive in energy yet can impact performance. In this case, *sftp*'s performance is delayed by 31.4%. Under inverse-barrier scheduling, there is no performance loss while showing modest energy savings.

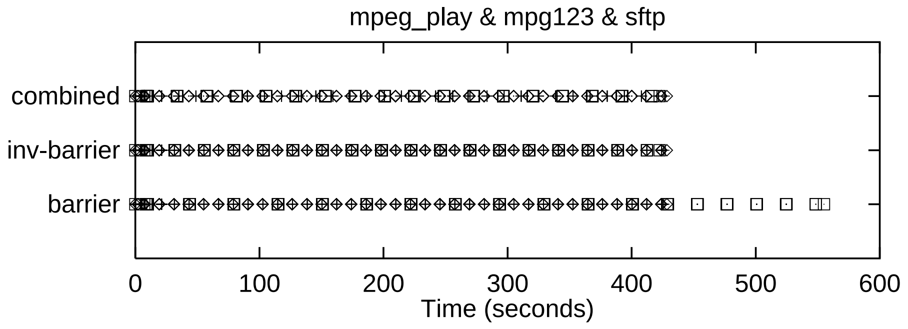

Fig. 6. Comparison of intra-program buffered I/O optimizations, with inter-program inverse-barrier scheduling, and with barrier scheduling on *sftp*. Barrier scheduling causes *sftp* to finish 132 seconds later, a 31.4% performance delay, while using 2.4% more disk energy overall

5 Analysis of Potential Energy Savings

Investigating the upper bounds on energy savings gives an indication whether this avenue of research is worthwhile. Exploring the involved parameters can provide insights into how this technique is beneficial. Towards that end, consider a situation of two programs, A and B, optimized to exhibit periodic resource access behavior. Inter-program compilation with inverse-barrier scheduling results in our optimized execution, but how much can it possibly save over the previous intra-program optimization, which has already shown large disk energy savings? We can approximate this with an analytical examination of the cases where the intra-program execution deviates from the optimal case where all programs access a resource at the same time (i.e., in batch mode). Our inter-program optimization results in such an optimal case. Thus, the difference represents the potential energy savings.

Refer to Table 2 for a list of involved parameters. The following description of these parameters are illustrated in Figure 7. Hence, P_A and P_B represent the period between resource accesses by programs A and B; assume $\frac{P_A}{2} < P_B < P_A$ and let $\Delta P = P_A - P_B$. Each program accesses the resource for an amount of time, R_A and R_B. The rise and fall in the graphs of programs A and B merely indicate a resource access. Only one resource is considered, so its corresponding hibernation threshold time will be designated simply H. If a resource will be idle for at least H, then hibernation will be beneficial and assumed to be initiated. Consequently, we assume $min(P_i) > H$; otherwise any chance for hibernation is gone.

Table 2. Analytical parameters

Variable	Description
P_i	resource access Period of program i
R_i	length of access (Read) time by program i
H_i	Hibernation threshold of resource i
E_i	average Energy use in case i

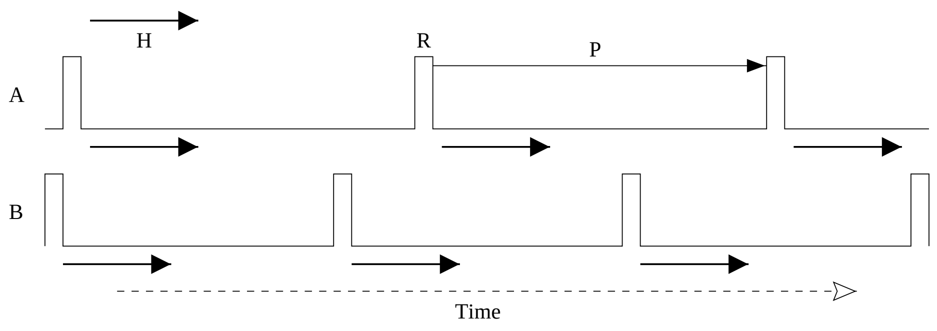

Fig. 7. Resource access patterns of programs A and B

There are three ways to categorize the resource access patterns, demonstrated by the three accesses of A (A^1, A^2, A^3) along with the four accesses of B (B^1, B^2, B^3, B^4). A^1 is *optimal* because it is clustered with B^1. Since the resource will not be used again within H of A^1, hibernation may be initiated immediately. A^2 is *sub-optimal* because it occurs within H of B^2. The accesses are mildly offset, and the resource consumes extra energy by remaining in the idle power state. A^3 is *out-of-phase* because it occurs after H of B^3, and B^4 occurs after H of A^3. The effect is that B^3's and A^3's hibernation periods are immediately interrupted. There is little opportunity to save energy during the respective hibernation periods. The next question is, what percentage of time do each of the three cases occur?

The optimal case, *opt*, can be expected to occur $\frac{\Delta P}{P_B}\%$ of the time. If A and B have a synchronized access, then each respective access afterward will be offset by ΔP. They will coincide again after $\frac{P_B}{\Delta P}$ accesses. The sub-optimal case is expected to occur $\textit{sub-opt} = (max(\frac{2H}{P_B}, 1) - \textit{opt})\%$ of the time. The first term refers to all accesses within H of an access, including the optimal case. Subtracting the optimal case gives just the sub-optimal case. Lastly, the out-of-phase case occurs $\textit{out} = (1 - (\textit{opt} + \textit{sub-opt}))\%$ of the time, or simply the remaining percentage of time after subtracting the optimal and sub-optimal cases.

The next step toward estimating potential energy savings is calculating the average energy consumed during the three cases and computing the differences. The energy usage for each case can be obtained from a power consumption profile which is a simple graph showing the amount of time spent in the various power states. In the non-optimal cases, there may be many instances of the graphs corresponding to different timing offsets between accesses. These are averaged to produce one profile graph for each of the sub-optimal and out-of-phase cases. Figure 8 shows what a sample power profile may look like. The average energy usage of each case, E_i, is now a matter of summing the power levels over time.

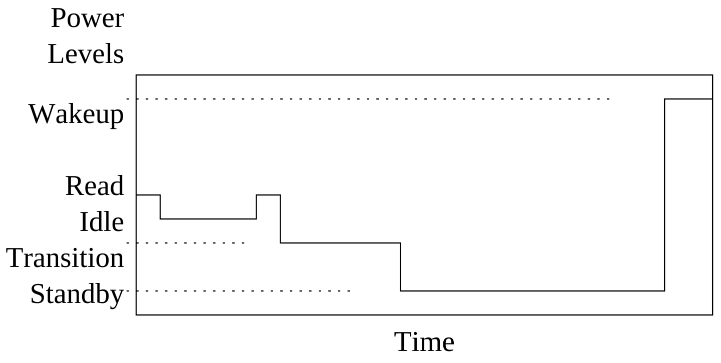

Fig. 8. Sample power profile graph

Finally, with the average energy use of all three cases (E_{opt}, $E_{sub-opt}$, E_{out}), an upper bound on potential energy savings can be computed. Energy savings over the sub-optimal case is $\Delta E_{sub-opt} = E_{sub-opt} - E_{opt}$, and the out-of-phase case is $\Delta E_{out} = E_{out} - E_{opt}$. These cases occur *sub-opt* and *out* percent of the time. Thus, the upper bound is $(sub\text{-}opt * \frac{\Delta E_{sub-opt}}{E_{opt}}) + (out * \frac{\Delta E_{out}}{E_{opt}})$. Applying this analysis to the experiment of running *mpeg_play* and *mpg123* together under inter-program optimization, the upper bound is estimated at 26.2% while actual savings is 15.9%. The upper bound can never be reached because of the small overhead involved during program startup for profiling to initialize the disk buffers. For this experiement, the startup overhead accounted for 5.8% of the total execution time.

6 Summary and Future Work

Inter-program optimization is a promising new compilation strategy for sets of programs that are expected to be executed together. Such sets occur, for instance, in resource restricted environments such as handheld, mobile computers. 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 discussed the potential benefits of inter-program analysis using the disk as the resource. An analysis of energy savings and simulation results for a set of three benchmark programs show that further significant energy savings over intra-program optimizations (between 5% and 16% for the simulations) can be achieved.

The compiler and OS have unique perspectives on key parts of the entire resource management scheme. We hope to experimentally explore and discover the strengths from each, then apply them in developing a resource-aware compiler and OS system. 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. We will also be using physical measurements to guide and validate our development efforts.

References

1. Heath, T., Pinheiro, E., Hom, J., Kremer, U., Bianchini, R.: Application transformations for energy and performance-aware device management. In: Proceedings of the Conference on Parallel Architectures and Compilation Techniques. (2002) Best Student Paper Award.
2. 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
3. Ousterhout, J.: Scheduling techniques for concurrent systems. In: Proceedings of the Conference on Distributed Computing Systems. (1982)

4. 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
5. 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)
6. Abramson, N.: The ALOHA system — another alternative for computer communications. In: Proceedings of the Fall Joint Computer Conference. (1970) 281–285
7. Roberts, L.: ALOHA packet system with and without slots and capture. Computer Communications Review **5** (1975) 28–42
8. 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)